Project Number:	284941	Project Title:	Human Brain Project

Document Title:	Neuromorphic Platform Specification
Document Filename:	HBP_SP9_D9.7.1_NeuromorphicPlatformSpec_56b296e from 13 May 2014
Deliverable Number:	D 9.7.1
Deliverable Type:	Platform Specification
Sub Project:	SP9
Planned Delivery Date:	M6 - 31 March 2014
Actual Delivery Date:	M7
Authors:	The deliverable has been written by the SP9 partners UHEI, UMAN, CNRS-UNIC, TUD and KTH. The complete version history with commit-info is available in the git repository: git@gitviz.kip.uni-heidelberg.de:hbp-sp9-specification-d9-7-1.git
Abstract:	This document provides the technical specifications for the Neuromorphic Computing Platform of the Human Brain Project. For each of the two complementary large-scale hardware implementations, detailed technical descriptions of the architecture, the user view and the electronic components are given. In addition, the support software required for the execution of experiments on the Platform is described and benchmark tasks for neuromorphic computing are proposed. The document closes with a list of key performance indicators and a timeframe for the Platform's construction.
Keywords:	neuromorphic, VLSI, analog, mixed-signal, many-core, brain-inspired computing, PyNN

Neuromorphic Platform Specification

13 May 2014 (git 56b296e)

Human Brain Project

Executive summary

The Human Brain Project will construct and operate a Neuromorphic Computing Platform consisting of two complementary hardware systems and the software infrastructure necessary for their operation. The size and the research opportunities of the HBP hardware systems will be unrivaled. They offer the first and so far only generic and remotely accessible neuromorphic computing facilities to perform research on this new computing paradigm.

This specification document is primarily written for regular consultation by researchers. It provides hardware and software developers and the user community with a technically detailed, comprehensive and quantitative description of the systems under construction. It also enables administrators to monitor progress using a set of high-level "key-performance-indicators (KPIs)".

This document has an introduction and four main parts. It starts with an introduction to neuromorphic computing and a description of the specific implementation and capabilities of the HBP Neuromorphic Computing Platform. For consistency with the other HBP platform specification deliverables, the software tools required to access, configure and operate the neuromorphic computing systems are described first (in part 1). Parts 2 and 3 contain detailed specifications of the two complementary hardware systems. These systems are the "Physical-Model (PM)" system to be installed in Heidelberg (Germany) and the "Many-Core (MC)" system to be installed in Manchester (UK). Part 4 introduces the benchmarks for the systems, and part 5 lists the scientific key performance indicators for monitoring the platform building progress.

The ability to build such a large scale and unique facility on an extremely short timescale during the 30 months ramp-up phase of the HBP builds on 10 years of preceding work, in particular the research, design and development carried out in the SpiNNaker, FACETS and BrainScaleS projects. The scale of the Human Brain Project allows for the aggregation of existing components, and for their assemblage into a user facility. This specification document therefore includes a precise, quantitative description of components developed prior to the HBP.

Contents

The Neuromorphic Computing Platform

What is Neuromorphic Computing?

Neuromorphic computing represents a radically new paradigm for information processing. The underlying concept is a direct mapping of brain architecture and functions on an array of asynchronously communicating, massively parallel computing elements in custom electronic hardware. An essential consequence is that the memory-holding structure and function of the neural circuits and the computing elements themselves are not physically separated as they are in traditional computing. Rather, they are intertwined on the same hardware substrate. This approach offers several advantages of neuromorphic systems compared to the traditional computing approach when simulating brain circuits.

Data and code describing brain activity are not shifted back and forth over large distances during simulation. This leads to a large advantage in energy consumption per basic operation. Such basic operations are the generation of an action potential or a synaptic transmission. On a logarithmic scale, the energy gap between the biological brain and a detailed simulation on a supercomputer is as large as 14 orders of magnitude. Using simplified models in supercomputer simulations reduces this gap to 10 orders of magnitude. Existing operational neuromorphic systems with comparable model complexity operate about 4 to 6 orders of magnitude above the brains energy consumption [71] or with the same distance to traditional computing. There are no known systems or even concepts to reach this performance with traditional supercomputers. Conceptual studies for a future exascale machine may reduce the energy consumption per fundamental operation only by a factor 2-5 [13] to reach a power consumption of 20-30 MW for such system.

Massive parallelism also affects the speed of brain simulations on neuromorphic systems [71]. Traditional very large-scale supercomputer based simulations with cell-level precision execute 100 to 1000 times slower than biological real-time. This makes them unsuitable for interfacing with physical robotic devices, and even more for the study of the dynamics that drives learning and development. Neuromorphic systems simulate brain activity at least at biological real-time. This is a considerable advantage when interfacing them with robotic systems. Specific implementations can even deliver considerable acceleration above real-time, up to a factor 10.000. This provides the only known method to study the dynamics of learning and development, covering time scales from biological milliseconds to years, or to explore large network parameter spaces.

The massive parallelism makes neuromorphic systems tolerant against failures of individual components. Like the brain, which loses about one living cell per second, neuromorphic

systems can cope with failing components through graceful degradation rather than catastrophic failure. This resilience will be a prerequisite for constructing future, very large neuromorphic systems made from unreliable components like memristors.

In addition to the technical advantages described above, there are several fundamental open research questions related to neuromorphic computing. The biological brain operates with noisy and diverse components. It is not deterministic but inherently stochastic. Understanding these features and exploiting them for a fundamentally new way of computing requires a large-scale and fully configurable research platform like the one under construction in HBP. Here, it is particularly important to grant access to scientists that have not contributed to the design and construction, but rather use the platform as a user facility, very much like scientists already use traditional generic computers. This service is arguably the most significant contribution of the Neuromorphic Computing Platform in the HBP.

Finally, there may exist a formal theory of the brain based on fundamental insights from mathematics or theoretical physics. Examples for such insights are analytical topology or topological field theories. Although still rather speculative, such a fundamental theory would need to be validated by controlled experiments. Neuromorphic systems on artificial substrates may well provide the only viable experimental access.

What are the key features of the HBP Neuromorphic Computing Platform?

The HBP delivers neuromorphic computing with key features that are summarized in this section.

Complementarity: The platform provides access to two different and complementary neuromorphic computing technologies.

The mixed-signal PM (physical model) system (figure .1) initially consists of 4 million analog neurons and 1 billion synapses implemented on 20 8-inch silicon wafers. Biological and electronic parameters of the cells, as well as the network topology, are user configurable. The biological model for the neurons is the Adaptive-Exponential-Integrate-and-Fire Model (AdEx), synapses have 4-bit precision weights and feature short-term and long-term plasticity. The system is accelerated and runs at 10.000 times biological real-time.

The digital MC (many-core) system (figure .2) initially consists of 500.000 ARM968 processor cores. A single chip contains 18 cores running integer arithmetics at 200 MHz, a shared system RAM and a router for address and package based spike transmission. Each chip has 6 bi-directional links capable of transmitting 6 million spikes per second. A 128 Mbyte DRAM is stacked on the chip die. The system runs at biological real-time.

Configurability: In view of the exploratory phase of neuromorphic computing it is essential that the systems under construction are as unconstrained as possible given the chosen technological approaches. Both HBP systems offer a very high degree of configurability with respect to the network architecture and the local models used for neurons, synapses and plasticity. The PM system uses cross-bar switches, analog floating gates and SRAM cells for this purpose. The MC system is based on programmable ARM cores connected by bi-directional links. Both systems are capable of performing a wide spectrum of experiments ranging from biological reverse-engineered circuits to highly abstract networks, which may be as extreme as random

Figure .1: Rendered View of the NM-PM1 system (for explanations see page 55)

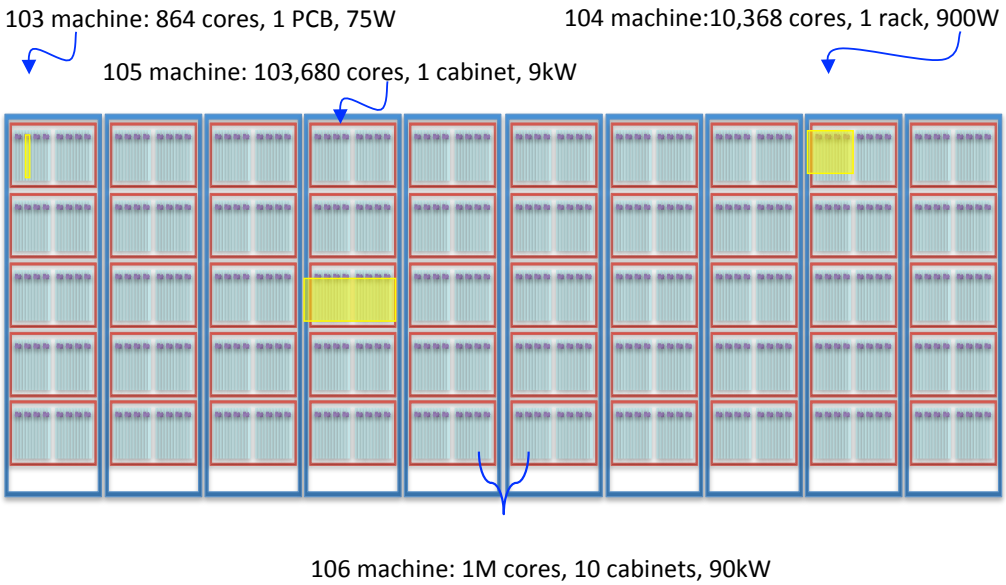


Figure .2: Concept view of the NM-MC1 system

connectivity.

Low Energy and High Speed: Both HBP NM systems offer several orders of magnitude advantages over traditional simulation computers in terms of their energy consumption and simulations time. The energy gap in performing a single synaptic transmission between the

biological brain and a detailed computer simulation is as large as 14 orders of magnitude, corresponding to 10 fJ in the former case and 1J in the latter. Simplified models executed on traditional computers lead to a reduction to 0.1 mJ, which is still 10 orders of magnitude worse than biology. The NM systems of HBP are consuming 10.000 pJ and 100 pJ for the MC and the PM systems, respectively. It is essential to note, that these numbers are not obtained from isolated lab samples but rather from fully functional systems including all overheads from control systems, losses in power supplies and similar effects.

Simulations of large networks on traditional computers typically run 100 to 1000 times slower than biological real-time. This renders a real-time link to physical robots or a study of slow learning and developmental processes impossible. In this respect the complementarity of the two HBP systems is very evident. The MC system operates at biological real-time, making it an ideal candidate to connect to physical robots with vision and sound sensors as well as mechanical moving parts and actuators. The PM system, with the large acceleration factor of 10.000, can compress a day of development into 10 seconds. This provides the only known access to slow learning and developmental processes with an effective biological timing precision in the sub-millisecond regime, where processes like STDP drive the dynamics of synapses. The large acceleration factor even allows to explore evolutionary time-scales in experiments lasting several days or even months.

Scalability: The scale of both phase 1 systems is entirely determined by the financial funds available for their construction. For growth of up to a factor 10 the cost for larger systems will simply scale with the growth factor. No fundamentally new technological approaches would have to be developed. This is an important feature of the massively parallel approach and it should be exploited whenever extra funding becomes available. For even larger systems the costs will start to grow faster than linear because of costs for more advanced infrastructure like space, power and cooling. Also, new assembly technologies like 3D-integration and automated manufacturing would drive the costs. At this point, upgradability will become an important feature (see below).

Hybrid Operation: Although there are early experiments that need to be performed with stand-alone neuromorphic systems, the important new insights will only arise once those systems interact with data or the environment, and once learning and development is driven by those interactions. In the case of the real-time MC system, closed external perception-action loops can be implemented using physical robots. For the accelerated PM system this is not feasible. Here, the external data will be provided by an nearby high performance computer operating in a closed loop with the NM system (figure .3). This so-called hybrid operation of an NM system with a traditional computer is also required for other purposes like functional simulations of larger brain areas for a multi-scale approach, or for performing the mapping and routing of reverse engineered biological networks to the hardware substrate. For this reason the PM system will operate a 5 TFlop machine in close physical proximity to the NM system.

Non-Expert User Access: The application of NM systems has so far been restricted to users with very detailed knowledge about the specific underlying hardware system and the dedicated software package provided to operate the system. This is very different from traditional computers, where established software packages allow efficient use with very little training effort. The HBP NM Platform systems will provide a unified software suite that

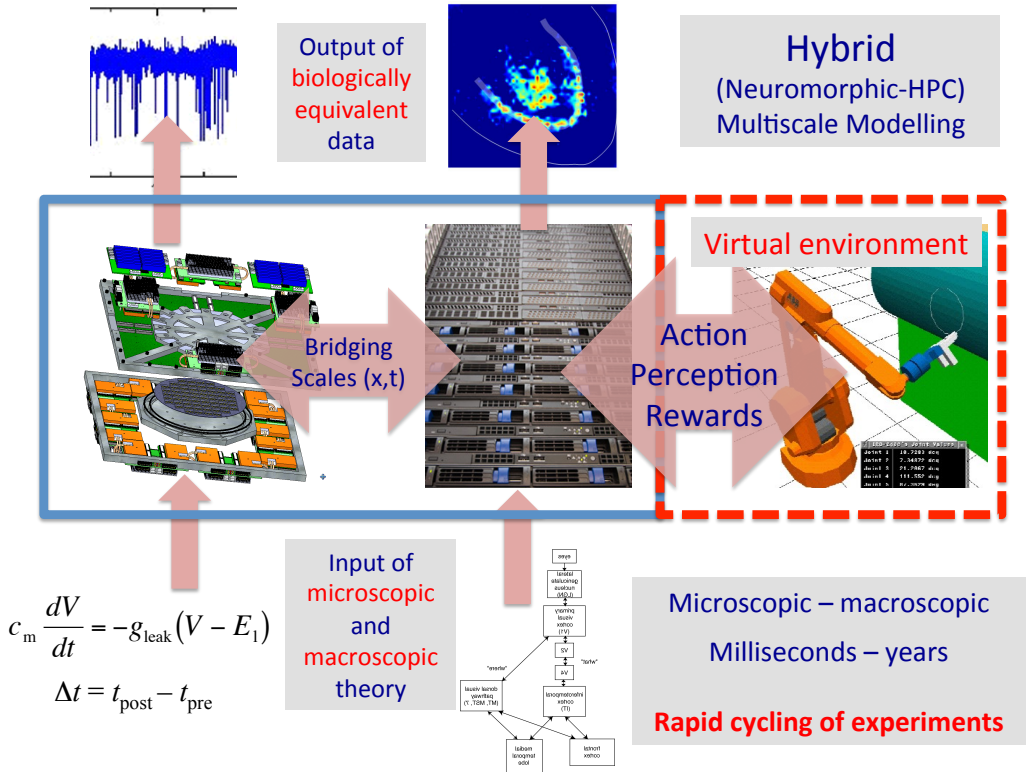


Figure .3: Hybrid operation

enables access by non-expert users. A typical example are neuroscientists running experiments implementing reverse engineered circuits. The software suite contains a description language for networks (PyNN), the mapping and routing from biology or a theoretical model to the hardware substrate, a simulation and verification tool, and tools for the storage and the analysis of the produced data. As a whole, the NM software suite will be integrated into the HBP Unified Portal, allowing for an integration with the Neuroinformatics Platform, the brain simulations and the neurorobotics simulation environment.

Upgradability: It is expected that data integration and simulation in HBP will deliver a clearer idea of which aspects of neural circuits are essential for computation. This new knowledge will most likely require the design of new and improved electronic circuits, including the necessary new chip design. Also, device and VLSI technologies will develop and more advanced process nodes are likely to become accessible to neuromorphic computing. The groups in the NM Subproject are therefore already developing concrete plans to upgrade their systems. In this context, "upgradability" is very important. Infrastructure elements like power supplies, cooling, racks, control boards, readout- and monitoring systems, and the software tools will be transferred to and reused by the new hardware generations in order to reduce the development time.

How will the NM Platform be used?

The high degree of configurability and the requirement to allow for the use by non-experts require the set-up of an integrated user concept. It is planned to provide training session for prospective users. In the training session the hardware architectures and software tools will be described and hands-on exercises will be offered to gain experience with this new type of computing. New users should initially work very closely with the experts in the NM subproject. After gaining some initial experience, users will be able to access the NM systems remotely from their home labs. The operation of the systems will be carried out through a web based interface and a sharing of the system resources by a scheduling system. On-call experts will be available to support remote and local users.

Integration of the NM Platform into the HBP Platform Ecosystem

The NM platform is an integral part of the HBP platform ecosystem. It will be operated through the HBP Unified Portal which offers access to all users of the HBP infrastructure.

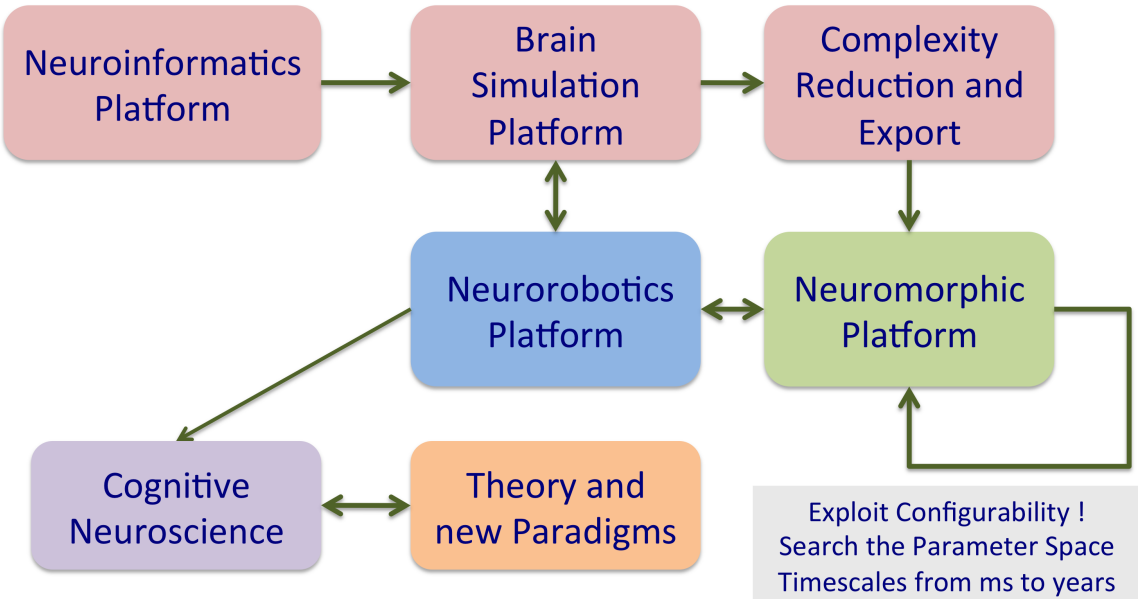


Figure .4: Integration of the Neuromorphic Computing Platform into the HBP Platform Ecosystem

The HBP integration of the NM platform is visualized in figure .4. Neuroscience data is aggregated by the Neuroinformatics Platform and then used as a basis for circuit building and simulation performed by the Brain Simulation Platform. The simulations run on high performance computers, which offer a very high degree of flexibility but are not very energy efficient and operate typically 100-1000 times slower than biological real-time. The simulations do interact with the Neurorobotics Platform, which offers the possibility to run closed loop simulations with virtual sensors, actuators and environments.

The detailed cell models used by the Brain Simulation Platform will then be reduced in complexity. In a first approach, point neurons will be used as an extreme case of complexity reduction. The reduced circuits can be transferred to and executed on the machines of the NM Platform with a large energy and speed advantage. In particular the physical model machine can execute emulations 10.000 times faster than biological real-time. In that system a day of learning and development can be reduced to an effective wall clock time of 10 seconds. Also, the accelerated operation allows to scan large parameter regimes for a systematic study of model variations. The large exploratory power of the NM Platform should also guide theoretical studies. The Platform is therefore closely integrated with the European Institute for Theoretical Neuroscience (EITN) in Paris.

Finally, the NM Platform machines will be part of the overall computing infrastructure in HBP. The high performance computers may be used to perform placing and routing for the neuromorphic machines, and the experience with the construction of the neuromorphic machines can also give guidance to the design of future, energy efficient high performance computers.

The purpose of this document

This specification document is primarily written for regular consultation by researchers. It provides hardware and software developers and the user community with a technically detailed, comprehensive and quantitative description of the systems under construction. It also allows administrators to monitor the progress through a set of high-level “key-performance-indicators (KPIs)”.

As construction of the first phase systems proceeds and upgrade concepts evolve, the document will be continuously updated. It will be available in the HBP document repository as a living document accessible to all developers and users.

Part 1

User interface to the Neuromorphic Computing Platform

1.1 Overall goals

The Neuromorphic Computing Platform will enable users to run simulation/emulation experiments on the two neuromorphic computing systems, the Heidelberg system ("Neuromorphic Computing with Physical Emulation of Brain Models", part 2 and the Manchester system ("Neuromorphic Computing with Digital Many-core implementation of Brain Models", part 3).

This part of the specification addresses the user interface to the Platform, both direct access by users and interactions with other HBP platforms.

1.2 Use cases

We expect that a user may wish to interact with the Platform in one of three ways:

- direct interaction through a web page
- interaction via the HBP Portal
- scripted interaction

For the Heidelberg system, there may be three types of experiments:

1) single runs (potentially long-running, using plasticity);
2) parameter sweeps or other batch-mode experiments;
3) closed-loop experiments involving interaction with a virtual environment.

For the Manchester system, the same three types of experiment are possible, plus closed-loop experiments with a real environment, through interaction with the Neurorobotics platform.

1.2.1 A single run of a simple network model

Primary actor Bill, a computational neuroscientist

Description Bill has created a network model with point neurons and short-term synaptic plasticity using the PyNN API. He has simulated the model using the NEST and NEURON simulators, and now wishes to check that the results from neuromorphic hardware are comparable.

Preconditions The model and experiment description are in a single Python script on Bill's laptop.

Success scenario

1) In a web browser, Bill navigates to the home page for the Neuromorphic Computing Platform and logs in to his user page.
2) Bill can see a list of previous jobs he has run on the Platform.
3) Bill clicks a button to request a new job.

4) Bill copies the content of the Python script from his text editor and pastes it into the appropriate text box.
5) Bill selects the Manchester system.
6) Bill submits the job request.
7) Bill is returned to his user page, where he can see that his new job has been added to the list of jobs with the status "in queue".
8) When the job is complete, Bill receives an e-mail containing a link to the job detail page.
9) Bill clicks on the link, which opens the job detail page in his browser. This page shows that the job has successfully completed, and contains links to download the log and output data files generated by the experiment.
10) Bill downloads the data files and compares the results to his NEST simulations.

Alternate scenarios

1) There is a syntax error in Bill's script.
   a) when Bill submits the job request, he is taken back to the job submission page, where a traceback of the error appears.
   b) Bill corrects the error and resubmits the job.
2) There is an error in the output data-handling section of Bill's script, after the simulation section.
   a) Bill receives an e-mail informing him that the job was unsuccessful, and containing a link to the job detail page.
   b) The job detail page shows the error traceback and contains a link to download the log file, enabling Bill to debug his script.

1.2.2 A scripted run of a complex network model with input data and parameter files

Primary actor Carol, a computational neuroscientist

Description Carol has developed a detailed model of a sensory system, which uses spike-timing-dependent plasticity and receives naturalistic stimulation. Even on a traditional HPC computer, the simulation takes several days to run. Carol wishes to take advantage of the large acceleration factor of the Heidelberg system to bring the run time down to a few minutes, so that she can study the effect of parameter variations. Since she expects to submit many jobs with different parameters, she wishes to script the job submission process rather than click through a website.

Preconditions The model and experiment descriptions are written using the PyNN API and are in separate Python files in a public Git repository. The repository also contains parameter files, a file containing data used to construct the sensory stimuli, and a main script which reads all these files, launches the simulation and then handles the output data processing.

Success scenario

1) Carol downloads a Python client for the Neuromorphic Computing Platform job submission REST API.
2) Using the client library, she writes a short script to submit a job to the Neuromorphic Computing Platform and retrieve the results.
3) The job request script includes the name of the system (the Heidelberg system in this case), the URL of the Git repository, the path to the main script within the repository, and the list of arguments (parameter file name, etc.) required by the script.
4) After submitting the job request, the script receives a URL that returns a document indicating the job status.
5) The script polls the job status URL repeatedly until the job is complete, at which point the job status document contains the URLs of the output data files and the log file.
6) the script downloads the output data files and saves them to the local disk.

Alternate scenarios

1) There is an error somewhere in Carol's code
   a) the job status document indicates there has been an error, and contains the error traceback and the URL of the log file
2) The public Git repository is unavailable
   a) the job status document indicates there has been an error, and indicates the cause of the problem
3) Carol cancels the job submission script, or reboots her computer, after the job has been submitted but before the job has completed.
   a) the job remains in the queue
   b) when the job completes Carol receives an e-mail containing a link to the job detail page.
4) after submitting the job but before it has completed, Carol realizes she has made a mistake.
   a) Carol uses the Python client for the Neuromorphic Computing Platform job submission REST API to cancel the job.

1.2.3 Using the Neuromorphic Computing Platform through the Unified Portal and Brain Simulation Platform

Primary actor Dennis, a neuroscientist.

Description Dennis has used the Brain Builder component of the Brain Simulation Platform to create a network model of a brain region, using point neurons. He has successfully executed a simulation of the model on the HPC Platform using the NEST simulator, and now wishes to execute the model on the Manchester hardware preparatory to beginning a collaboration with the Neurorobotics sub-project. Dennis is not comfortable with Python coding, and wishes to use the Unifying Portal to perform his simulations.

Preconditions Dennis' model is available in the Unifying Portal.

Success scenario Using the Unifying Portal:

1) Dennis selects and executes a task that exports a Brain Builder model in a format suitable for execution on the Neuromorphic Platform (PyNN).
2) He configures a Neuromorphic simulation job, selecting the Manchester hardware.
3) He launches the job, which is then queued and executed when time is available on the hardware.
4) About an hour later, Dennis receives an e-mail telling him his job has completed successfully.
5) Dennis returns to the Unifying Portal, from where he can access the data files generated by his simulation, as well as provenance information about the execution, e.g. what version of the hardware system was used.

Alternate scenarios

1) Dennis' model contains features that are not supported by the Neuromorphic Computing Platform.
    a) The export task fails, with a clear error message indicating which features are not supported.
    b) Dennis consults the documentation for the Neuromorphic Computing Platform and modifies his model so that it will run on Neuromorphic Hardware.
    c) He runs simulations with the modified model on the HPC Platform, and finds that the results are qualitatively unchanged.
    d) He now submits a new job for the Neuromorphic Computing Platform, using the modified model, which successfully runs to completion.

1.2.4 Parameter sweeps

Primary actor Esin, a computational neuroscientist

Description Esin wishes to explore the parameter space of her network model. Due to its long run time, she needs to make use of the large acceleration factor of the Heidelberg system.

Preconditions The model and experiment descriptions are written using the PyNN API in a single Python file in a public Git repository.

Success scenario

1) Esin writes a batch configuration file. This provides values for those parameters that will be varied across runs. She commits this to the Git repository.
2) Esin downloads a Python client for the Neuromorphic Computing Platform job submission REST API.
3) Using the client library, she writes a short script to submit a job to the Neuromorphic Computing Platform and retrieve the results.
4) The job request script includes the name of the system (the Heidelberg system in this case), the URL of the Git repository, the path to the model script within the repository, and the path to the batch configuration file.
5) After submitting the job request, the script receives a URL that returns a document indicating the job status.
6) The script polls the job status URL repeatedly until the job is complete, at which point the job status document contains the URLs of the output data files and the log files from all of the runs in the batch.
7) the script downloads the output data files and saves them to the local disk.

Alternate scenarios

1) One of the parameter sets in the batch run contains values outside the valid range for the Neuromorphic hardware.
    a) The invalid run is skipped, and a warning is written to the log file.

1.2.5 Closed-loop experiment involving a virtual environment

Primary actor Fumiko, a roboticist.

Description Fumiko has developed a robot simulation within a virtual environment. The robot perceives its environment via a model retina, and acts upon its environment through actuators. Communication from the retina to the robot brain model and from the brain to the actuators is via spikes. The retina, actuators and virtual environment are implemented as a C++ application.

Preconditions Working with the developers of the Neuromorphic Computing Platform, Fumiko has successfully installed the virtual environment software on the Platform, working via remote shell access. The Python code for the brain model is in a Git repository, which has been checked out on the platform.

Success scenario

1) Fumiko writes a Python script which connects the brain model with the retina and actuators, using a PyNN extension that connects spike-emitting and spike-receiving ports (for example, using the MUSIC interface).
2) Using the REST API, Fumiko launches the job, which runs until the robot completes a pre-defined task, or until a pre-defined time limit is reached.
3) When the job is complete, Fumiko receives an e-mail that contains a URL for the job status.
4) Fumiko accesses this URL through the REST API and downloads the data and log files generated by the job.

1.3 Functional requirements

1.3.1 Model and experiment descriptions

1) Model descriptions must be written as Python scripts using the PyNN API.
2) To the extent supported by PyNN and the neuromorphic hardware, scripts may read all or part of the model description from NineML or NeuroML files.
3) Model scripts may read parameter values from external files.
4) The name of the simulator or hardware platform to use must be provided as a command-line argument, not within the script.
5) Up until the first internal release of the Platform, PyNN API versions 0.7 (`http://neuralensemble.org/trac/PyNN`) and 0.8 (`http://neuralensemble.org/docs/PyNN/`) shall be supported.
6) After the first internal release, older versions of the API will be deprecated as new versions are released.
7) Experiment descriptions must be written as Python scripts using the PyNN API.
8) The model and experiment descriptions may be combined in the same script, or as separate Python scripts; in the latter case there must be a main script which launches the experiment.
9) Scripts should avoid performing data analysis or visualization; rather the recorded data should be saved to file for later analysis and visualization.
10) The Platform shall provide one or more Tasks for the Task Repository of the Unifying Portal which export a Network level model constructed using the Brain Builder as a PyNN script.

1.3.2 Job control interface

1) Users and other Platforms will access the Neuromorphic Computing Platform by submitting jobs to a job queue server and retrieving results from the server.

2) The job queue server shall provide a REST API so that job submission, monitoring and retrieval of results can be performed by scripts and by other Platforms.

3) The REST API shall provide the following functionality:
   a) submission of jobs to be run on the neuromorphic hardware systems.
   b) the ability to select which neuromorphic hardware system (Heidelberg or Manchester) to use.
   c) the ability to provide model and experiment description scripts directly within the submission or by specifying an external version control repository.
   d) the ability to specify a project to which the job belongs.
   e) the ability to monitor job status (e.g. queued, being processed, completed successfully, incomplete due to errors).
   f) the ability to retrieve information about completed jobs or about errors. The information will include URLs for all files produced by the simulation.

4) During development, the Platform shall provide a web portal for job submission and monitoring. Use of the portal will be phased out once all of its functionality can be provided by the Unifying Portal.

5) The Platform shall provide a Python client library for the job queue server API.

1.3.2.1 Batch mode

1) Where single runs provide one model description and one experiment description, a batch job provides a single model but multiple experiments.

2) Each batch-mode job shall receive a single identifier, and the results shall be transmitted as a whole, rather than separately for each experiment within the batch.

3) Batch jobs shall be controlled by a configuration file, indicating the parameter set to use for each run within the batch.

4) For the Heidelberg hardware, parameters to be varied during parameter sweeps may not affect the network structure, since this would require re-mapping, and the benefit of the time acceleration would be lost.

1.3.3 Data handling

1) The Neuromorphic Computing Platform will not provide long-term file storage, but shall make use of resources provided by the Neuroinformatics Platform (Dataspace) and possibly the HPC Platform.

2) All data files generated by the Neuromorphic Computing Platform shall have a unique URI.

1.3.4 Closed-loop experiments

1) By closed-loop experiments, we refer to experiments in which a neuronal network simulation interacts with an environment, either real or virtual, using sensors and actuators.

2) Sensors must generate, and actuators be controlled by, spike events.

3) An interface shall be defined to connect spike producers/consumers to neuronal network models (an example of such an interface that could be used is MUSIC [Djurfeldt, 2010])

4) This interface shall be accessible through Python, enabling the entire closed-loop experiment to be defined in a single Python script.

5) Closed-loop experiments that use virtual environments, sensors and actuators shall be submitted using the same job submission system as open-loop experiments.

6) Closed-loop experiments that use real robots and real environments shall require reservation of a block of time on the relevant hardware platform, since a real-time, more interactive mode of operation is required.

1.4 Non-functional requirements

1.4.1 Sharing

Unless the results of a job are explicitly deleted, they will continue to be accessible by the user on the server. A mechanism is needed to enable access by someone other than the person who submitted the job. One possibility is to assign each job to a project, and then allow access by any user who is a member of that project.

1.4.2 Authentication and Authorization

1) Only authenticated and authorized users may submit jobs to the Platform.
2) Only the user who submitted a job, or an administrator, may cancel.
3) Access control to in-process and completed jobs shall be based on projects: all users who are members of the project associated with the job may access it.
4) Only an administrator may delete a job; other users with access may hide it.
5) No later than the first public release of the Platform, the Neuromorphic Platform shall use the central HBP user directory and authentication workflow.
6) In the initial, development phase, a local database will be used for authentication and authorization.

1.4.3 Security

Since the model and experiment definition format is Python code, there is an evident security risk. To mitigate this risk:

1) only authenticated and authorized users will be able to submit jobs (see previous section)
2) use of certain Python modules and functions will not be allowed (e.g. detected through static code analysis)
3) scripts will first be executed with a "mock" hardware backend in a sandboxed Python environment before being run on the neuromorphic hardware.

1.4.4 Accounting

The Neuromorphic computing systems are a limited resource. Although it may not be necessary in the initial development stage to ration access, a quota system shall be implemented by the time of the first public release of the Platform.

1.4.5 Efficiency and user volumes

The job queue system shall not put any further constraints on the number of simultaneous users and on job throughput beyond those imposed by the resource limitations of the hardware backends, i.e. the neuromorphic hardware shall not be kept waiting by the user interface.

1.4.6 Reliability

The job queue server is expected to have regular (1 / month) scheduled maintenance windows. Each maintenance window will be no more than 60 minutes long.

1.5 Architectural overview

1.5.1 Job submission API

1.5.1.1 Overview

Whichever interaction method is used, the workflow for single runs will be as follows:

- the user provides a model and experiment description in the form of a Python script using the PyNN API. The script could be provided by uploading, or by giving the reference to a database entry or software repository (e.g. as a URI).
- the user provides any necessary parameter and/or data files. Again, these could be uploaded or references to databases given.
- the user selects the hardware platform and configuration to be used.
- the central server verifies that the model and experiment description are valid and suitable for the hardware. For a PyNN script, this could involve running the script with a mock/dummy backend in a sandboxed environment.
- the job is placed on a queue. The user is provided with a URL that can be checked/polled for job status.
- when available for new jobs, each individual hardware platform regularly polls the queue. When a job for that platform is found, all files are transferred to the local system and the experiment executed. This may consist of several stages (e.g. mapping followed by execution), in which case the job status can be modified accordingly after each stage.
- all data and log files generated by the experiment are transferred from the local workspace, either to the central server or to a database/distributed file system (e.g. the INCF Dataspace).
- the job status is set to "complete" (or "error", as appropriate), an e-mail is sent to the user.
- the user can retrieve the data/log files from the central server, together with any relevant metadata (e.g. provenance information). The central server could also directly notify other systems (e.g. in the case of the HBP Portal).

1.5.1.2 Endpoints

This is an initial proposal, which will be modified as necessary during development to ensure all functional and non-functional requirements are satisfied.

URI	Action	Description
/	GET	return the URIs of the queues and project list
/queue/{stage}/	POST	place a job on the given queue
	GET	return the list of jobs on the queue
/queue/{stage}/{job-id}	GET	retrieve the specified job
	DELETE	remove the specified job from the queue
/queue/{stage}/next	GET	take the job from the head of the queue
/results	GET	show a list of jobs for the current user
/results/{job-id}	PUT	used by the hardware platforms when a job is taken off the queue
	GET	retrieve the specified job
	DELETE	hide the specified job
	PATCH	used for updating job status
/projects	GET	list projects of which the current user is a member
	POST	create a new project
/projects/{project}	GET	return list of jobs for this project
	DELETE	hide a project

{stage} may be "submitted" or "validated". This may not be needed if validation is sufficiently quick. Other stages (e.g. "mapped") could be used if needed. By "retrieve a job" we mean obtain a representation of a Job resource (see below); the job is not removed, a separate deletion step is necessary.

1.5.1.3 Resource descriptions

The API will return and accept the following resources, encoded as JSON. For each resource we give its name and the names and types of its attributes. "[{*type*}]" indicates that the attribute contains a list of items of the given type.

```
resource Job
    experiment_description - text
    input_data - [DataItem]
    hardware_platform - HardwarePlatform
    user - User
    project - Project
    timestamp_submission - timestamp
    timestamp_completion - timestamp
    status - ("submitted", "validated", "mapped", "finished", "error")
```

```
    output_data - [DataItem]
    logs - [DataItem]

resource User
    username - text
    full_name - text
    e_mail - text

resource Project
    short_name - text
    full_name - text
    members - [User]

resource DataItem
    uri - text
    mime-type - text
    contents - text

resource HardwarePlatform
    name - text
    configuration - dictionary containing strings and numbers
```

1.5.1.4 Serializations and allowed document types

Resource serializations will use the JSON format with UTF-8. We plan to use vendor-specific mimetypes to provide versioning.

1.5.1.5 Physical architecture

There are no particular requirements for the location of the central server. This could be in Heidelberg, Manchester, Gif-sur-Yvette or run on a cloud service.

1.5.2 Python client for REST API

The Python client is intended to make the REST API easier to use, by providing utility functions to simplify authentication, job monitoring, batch-job submission, data handling. The client will contain two main sub-modules, one for Platform users, and one for use at the hardware sites in Manchester and Heidelberg, to simplify the task of linking the central job queue server to local resource management software such as SLURM (see below).

1.5.3 Model/experiment verification

For reasons of efficiency and responsiveness it is best to catch errors in submitted Python scripts as early as possible. We therefore plan to introduce an initial verification step,

performed on the job queue server, before a job is accepted onto the queue. This verification may involve static code analysis, and will almost certainly involve running the script with a "mock" PyNN back-end.

The requirement that Platform users be authenticated and authorized to submit jobs renders the risk of users submitting malicious code minimal. Nevertheless, to minimise inadvertent side-effects of running jobs, the verification step will be run in a sandboxed environment probably based on Linux containers (e.g. using Docker).

1.5.4 Resource management software in Heidelberg and Manchester

The central queue server is a front-end to the entire Neuromorphic Computing Platform. Each of the hardware sites, Heidelberg and Manchester, will implement a system to take jobs from the queue, execute the job, and perform error- and data-handling. Most of this work can be done by the Python client for the REST API, possibly working with local resource management software such as SLURM.

1.5.5 Tools for exporting Brain Builder model descriptions as PyNN descriptions

Simplifying brain models produced by the Brain Simulation Platform so that they can run on the Neuromorphic Hardware Platform is the job of Task 9.3.2. This is a research project, in collaboration with the Theory sub-project, and so the tools cannot be specified at this time.

1.6 Interfaces to other platforms

1.6.1 Services required from other Platforms

1) Data storage – Neuroinformatics and HPC Platforms
2) Authentication – Unifying Portal
3) Provenance tracking – Unifying Portal
4) Execution of complex mapping tasks - HPC Platform (?)

1.6.2 Services provided to other Platforms

1) Execution of network simulation/emulation experiments on neuromorphic hardware.

1.7 Key performance indicators and Function blocks

To enable monitoring the progress of the user interface to the Neuromorphic Computing Platform, the following "Functions" have been defined. A numerical measure of the overall progress may be obtained by counting the number of Functions that have been implemented.

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.1	Leader:	Andrew Davison
Function Name:	Job queue server, minimal functionality		
Description:	A developer can submit a single job using a REST API, to be executed on the development system by NEURON or NEST, local authentication, local data storage, no provenance tracking		
Planned Start Date:	Month 7	Planned Completion Date:	Month 12
Requires Functions:	none		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.2	Leader:	Andrew Davison
Function Name:	Python client for job queue REST API		
Description:	A Python package is provided to simplify use of the job queue REST API		
Planned Start Date:	Month 12	Planned Completion Date:	Month 13
Requires Functions:	9.3.1.1		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.3	Leader:	Andrew Davison
Function Name:	E-mail notifications		
Description:	The Platform will send e-mails to the user who submitted a job (unless the user has opted out of such e-mails) upon completion of the job or on encountering an unrecoverable error.		
Planned Start Date:	Month 13	Planned Completion Date:	Month 14
Requires Functions:	9.3.1.1		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.4	Leader:	Andrew Davison
Function Name:	Verification/sandboxing		
Description:	When a job is submitted to the queue server it will first be executed in a sandbox environment with a mock simulator, before being made available to the hardware systems.		
Planned Start Date:	Month 15	Planned Completion Date:	Month 18
Requires Functions:	9.3.1.1		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.5	Leader:	Andrew Davison
Function Name:	Job queue server with central authentication		
Description:	Authentication for the job queue server is provided by the HBP central authentication service		
Planned Start Date:	Month 18	Planned Completion Date:	Month 23
Requires Functions:	9.3.1.1		

Task No:	9.5.4	Partner:	UHEI (P45)
Function No:	9.5.4.1	Leader:	Eric Müller
Function Name:	Job queue server usable by Heidelberg system		
Description:	Jobs submitted to the queue server can be executed by the Heidelberg facility		
Planned Start Date:	Month 13	Planned Completion Date:	Month 18
Requires Functions:	9.3.1.2		

Task No:	9.5.4	Partner:	UMAN (P73)
Function No:	9.5.4.2	Leader:	David Lester
Function Name:	Job queue server usable by Manchester system		
Description:	Jobs submitted to the queue server can be executed by the Machester facility		
Planned Start Date:	Month 13	Planned Completion Date:	Month 18
Requires Functions:	9.3.1.2		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.6	Leader:	Andrew Davison
Function Name:	Data storage using resources provided by Neuroinformatics or HPC Platforms		
Description:	Jobs executed on the Neuromorphic Computing Platform can store output data using resources provided by the Neuroinformatics or HPC Platforms		
Planned Start Date:	Month 24	Planned Completion Date:	Month 25
Requires Functions:	9.3.1.1		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.7	Leader:	Andrew Davison
Function Name:	Provenance-tracking of Neuromorphic jobs		
Description:	Full provenance information is stored for jobs executed on the Neuromorphic Computing Platform		
Planned Start Date:	Month 18	Planned Completion Date:	Month 23
Requires Functions:	9.3.1.6		

Task No:	9.3.2	Partner:	CNRS (P7)
Function No:	9.3.2.1	Leader:	Andrew Davison
Function Name:	Export of Network level model constructed using the Brain Builder as a PyNN script		
Description:	A Task is provided for the Unifying Portal Task Registry that can export Network level models consisting of point neurons as a PyNN script, which can be executed on the Neuromorphic Platform.		
Planned Start Date:	Month 7	Planned Completion Date:	Month 24
Requires Functions:	none		

Task No:	9.3.1	Partner:	CNRS (P7)
Function No:	9.3.1.8	Leader:	Andrew Davison
Function Name:	Job submission and retrieval using Brain Simulation Platform		
Description:	Jobs can be submitted from the Brain Simulation Platform, executed on the Neuromorphic Computing Platform, and the results retrieved on the Brain Simulation Platform.		
Planned Start Date:	Month 26	Planned Completion Date:	Month 30
Requires Functions:	9.3.2.1, 9.3.1.7		

Task No:	9.5.4	Partner:	CNRS (P7)
Function No:	9.5.4.3	Leader:	Andrew Davison
Function Name:	Batch jobs		
Description:	The Platform will support submission, monitoring and execution of batch jobs, where a single network is executed repeatedly with different neuron/synapse parameters and/or inputs.		
Planned Start Date:	Month 19	Planned Completion Date:	Month 24
Requires Functions:	9.5.4.1		

Task No:	9.5.4	Partner:	UHEI (P45)
Function No:	9.5.4.4	Leader:	Eric Müller
Function Name:	Quotas		
Description:	Each user will have a usage quota, to ensure equitable use of the Platform		
Planned Start Date:	Month 24	Planned Completion Date:	Month 30
Requires Functions:	9.5.4.1, 9.5.4.2		

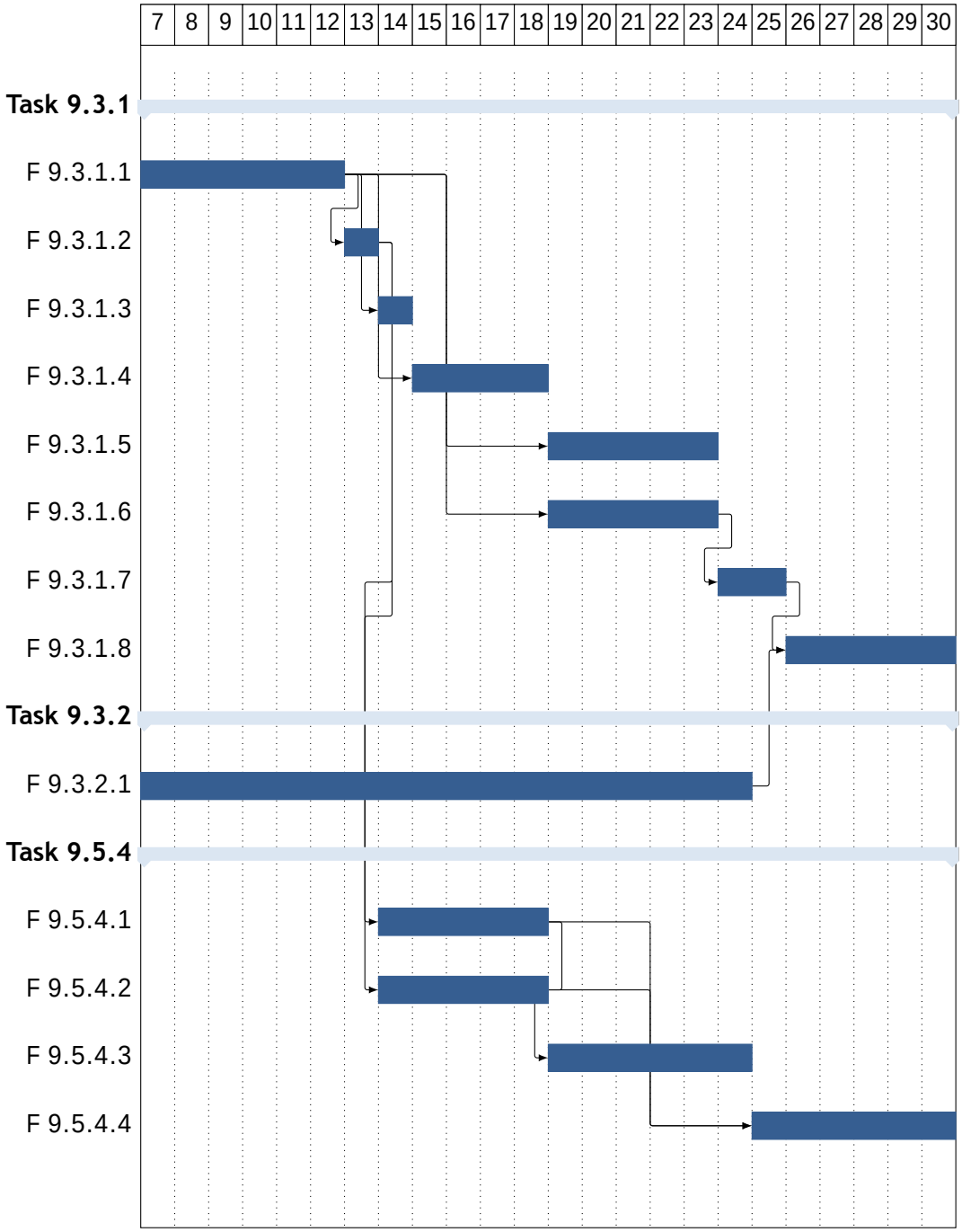


Figure 1.7.1: Scheduling of Functions to be implemented in building the user interface to the Neuromorphic Computing Platform. The numbers in the top row refer to project months. Month 7 is April 2014, Month 30 is March 2016.

Part 2

Neuromorphic Computing with Physical Emulation of Brain Models

2.1 Physical Model Platform: NM-PM

This part of the SP9 specification covers all hardware and software aspects related to the task termed "Neuromorphic Computing with Physical Emulation of Brain Models" in the HBP project. The first chapter gives an introduction to the physical system as it will be constructed as part of the SP9 platform. The second chapter introduces the individual components and how they relate to each other. The remaining chapters of this documentation cover all components in detail. The overall structure is a bottom-up approach, starting with the mechanical and electrical aspects of the hardware in chapters 2.5 to 2.9, then moving up to the communication protocols and software interfaces tying all components together (chapters 2.10 and 2.11). Chapters 2.12 and 2.13 document the different software libraries necessary to access the hardware components and to translate biological network descriptions into the configuration data for the hardware. Finally, chapter 2.2 presents a high-level view of the system as it will be seen by the scientist who plans to use the system for her or his research.

2.1.1 Neuromorphic Physical Model

The part of the SP9 platform implementing “Neuromorphic Computing with Physical Emulation of Brain Models” is based on a hardware system termed Neuromorphic Physical Model (NM-PM). It consists basically of a custom hardware system which implements the physical emulation of brain models and a conventional compute cluster to interface the custom part to the user and to execute parts of the model in synchrony to the physical models. These hybrid models are essential for all tasks involving motor feedback to the environment, since the physical model is limited to modelling neurons and synapses.

Fig. 2.1.1 shows the main components of the NM-PM system. The core of the custom hardware implementing the physical models is an electronic assembly called a Wafer Module (Wafer Module). It consists of a 20cm silicon wafer mounted on top of a large printed circuit board. The wafer is manufactured in 180nm Complementary Metal-Oxide-Semiconductor (CMOS) technology from the Taiwanese micro electronics contract manufacturer UMC. It contains 384 identical Application Specific Integrated Circuits (ASICs) named High-Input Count Analog Neuronal Network Chip (HICANN), implementing the physical models of up to 512 neurons and 114688 synapses each. Therefore, a Wafer Module has a total modelling capacity of up to 44 million synapses and 200k neurons. The first version of the SP9 platform will consist of 20 wafer modules for a total capacity of up to 4 million neurons and 0.88 billion synapses.

The most important features of the physical model implemented in the HICANN chip are the large number of inputs which can be connected to a single neuron, 14336, and the

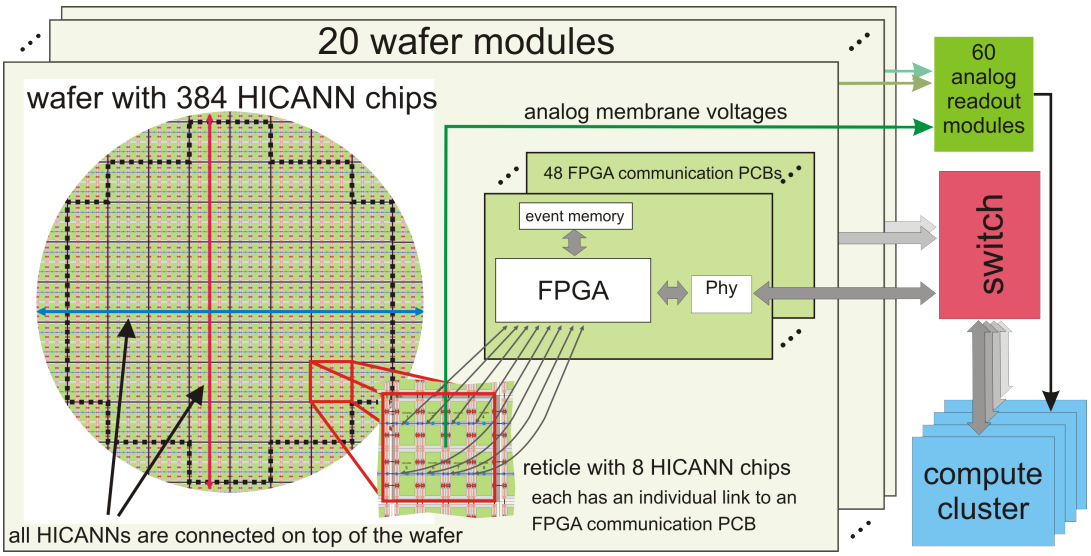


Figure 2.1.1: Simplified overview of the NM-PM1 system.

acceleration factor of the emulation compared to wall time, which is typically 10^4.

In addition to the HICANN Wafer, the Wafer Module hosts 48 FPGA Communication PCBs (FCPs), as well as power supply and interface submodules. The Wafer Module needs only a single -48V telecommunications supply. A separate single-board computer controls the operation of a Wafer Module and communicates via a standard Ethernet link with the compute cluster. Thus, the Wafer Module is completely software controlled, including power sequencing and initialization.

The Wafer Modules are distributed across five industry-standard 19" racks. Fig.2.1.2 shows a computer generated image of the planned arrangement. In addition to the Wafer Modules each rack contains 12 Analog Readout Modules (AnaRMs) to digitize the analog membrane voltages of the neurons located on the HICANN Wafer.

The communication between the Wafer Module and the compute cluster is mediated by the FPGA Communication PCBs (FCPs), which are connected by 48 Gigabit-Ethernet links to one standard Ethernet switch per Wafer Module. These Wafer Module switches provide a 10 Gigabit uplink to a central 48 port Top-of-Rack (ToR) 10 Gigabit switch.

The compute cluster consists of 20 four-core diskless workstations, one per wafer module, each equipped with a 10 Gigabit Remote Direct Memory Access (RDMA)-capable network interface. Four additional cluster nodes serve as dedicated storage nodes, connected to the central switch by 40 Gigabit Ethernet.

2.1.2 Constitutent Parts of the NM-PM1

This section contains a full list of all individual parts of the NM-PM1 hardware. It is provided for reference. A detailed specification of all components is given in the respective chapters of this document.

Figure 2.1.2: Rendered View of the NM-PM1 system. ① Wafer Module, ② Wafer Module network switch, ③ analog readout subsystem, ④ ToR 40Gbit network switch, ⑤ storage server node, ⑥ computer server node, ⑦ Wafer Module power supply, ⑧ top and bottom fan units for Wafer Module

Main components of the NM-PM:

Wafer Module 20 modules distributed across 5 industry standard 19" racks

Compute Cluster 20 1U compute server nodes and four 3U Input/Output (I/O) server nodes

Analog Readout Subsystem five rack mountable assemblies, one per wafer module rack, each containing 12 Analog Readout Modules (AnaRMs).

Wafer Power Supply Industry standard -48V supplies. Three 2kW units capable of current sharing are mounted together in one 1U case. Five of these 6kW assemblies are mounted at the bottom of the central network rack. Each supplies one rack with four Wafer Modules.

Wafer Module network switch One 48-port Gigabit Ethernet (GbE) aggregation switch per Wafer Module incorporating two 10-Gigabit Ethernet (10GbE) uplink ports per switch.

Top-of-Rack network switch 48-port 10GbE switch with four additional 40-Gigabit Ethernet (40GbE) ports. All ports use electrical interfaces based on Small Form-Factor Pluggable (SFP+) or Quad SFP (QSFP) standards, respectively.

Components of the compute cluster:

Compute/Wafer Node 20 1U compute server nodes with one single-socket high-end Desktop CPU (Intel® Core™ i7-4770), 16 GiB RAM, and one low-latency 10GbE network interface card.

Storage Node Configured as the Compute Node. Additional components are Solid-state Disks (SSDs) connected via Peripheral Component Interconnect Express (PCIe) bus and conventional Hard disk drives (HDDs).

Network Connectivity is provided by the ToR network switch and a GbE-based control network (cf. wafer module components).

Components of the wafer module:

HICANN Wafer A 20cm silicon wafer containing the neuromophic circuits, distributed across 384 HICANN ASICs and connected to each other on the wafer surface.

Wafer Module Main PCB (MainPCB) The MainPCB connects to the wafer by 384 elastomeric connectors. It contains Power Field-Effect Transistors (Power-FETs) to individually switch all supplies to the wafer. Power can be controlled on a per-reticle basis (8 HICANN chips).

FPGA Communication PCB (FCP) 48 FCP boards plug into the MainPCB and connect directly to the communication links of the wafer.

Wafer I/O PCB (WIO) Four interface boards sit on top of the FCPs, housing the 48 Gigabit-Ethernet connectors and Phy-circuits. They come in a horizontal and a vertical variant, termed Horizontal Wafer I/O PCB (WIOH) and Vertical Wafer I/O PCB (WIOV), respectively.

PowerIt Main Power Supply PCB (PowerIt) A 2kW main power supply board providing electrical insulation and down-conversion of the -48V input to an intermediate 10V supply used by the auxilliary power supplies and the Field-Programmable Gate Array (FPGA) boards. It also contains the point-of-load converters for the main wafer supply voltages (two times 1.8V, 400A each). An on-board Microcontroller Unit (MCU) provides electronic switching of all power supplies and on-board monitoring of all voltages and currents.

Auxiliary Power Supply PCB (AuxPwr) Two AuxPwrs provide miscellaneous supply voltages.

Analog Breakout PCB (AnaB) Two breakout boards to connect the analog readout channels from the wafer to the respective cabeling.

Single-Board Control Computer One Raspberry-Pi [10] is used as a control computer allowing full system control via one Ethernet link. It communicates by I2C with the power supplies and the power control and monitoring boards.

Monitoring and Control PCB for Reticles (Cure) Six small boards which plug in directly into the MainPCB. They provide monitoring of all wafer voltages and control the array of Power-FETs on the main board.

Wafer Module Mechanical Assembly The mechanical assembly provides mechanical mounting for the main pcb and the main power supply boards. It fixates and protects the wafer and generates the mechanical pressure for the elastomeric connectors.

Components of the analog readout subsystem:

Flyspi FPGA PCB (Flyspi) 12 small data aquisition Printed Circuit Boards (PCBs) containing a fast Analog-to-Digital Converter (ADC), an FPGA and 512MiB Dynamic Random Access Memory (DRAM) memory.

Analog Frontend PCB (AnaFP) Each Flyspi carries one AnaFP containing multiplexers and one pre-amplifier to connect the analog readout channels from the Wafer Module to Flyspi.

Flyspi Breakout PCB (FsBo) 12 small mechanical adapter boards for mounting the Flyspis.

Control Computer Intel Next Unit of Computing (NUC)[9] based Linux system provides the Universal Serial Bus version 2.0 (USB 2.0) resources for connecting the Analog Readout Modules (AnaRMs) to the Compute Cluster.

Analog Readout Mechanical Assembly 3U rack-mount for the 12 Flyspis, the Control Computer and four USB 2.0 hubs.

2.2 Users view of the NM-PM system

This chapter describes the user's view of the NM-PM. Each section characterizes a class of tasks that can be accomplished with the NM-PM, as well as the required tools and, where appropriate, a recommended workflow to accomplish the task. These tasks encompass the use of the hardware system as a neuroscientific modeling tool as well as the evaluation of hardware performance.

Here, neuroscientific modeling stands for the creation and investigation of mathematical models of spiking neural networks. The NM-PM allows the user to emulate such a model on a large-scale, parallel hardware device with a high acceleration (section 2.1.1), provided the model is compatible with the provided feature set. This type of usage is outlined in sections 2.2.1 to 2.2.3.

The evaluation whether the model is compatible with the provided feature set is detailed in section 2.2.4.

2.2.1 Usage of the NM-PM as a modeling back-end

The central part of the user interface of the NM-PM is the PyNN (PyNN) Application Programming Interface (API) (section 2.13.3, [16, 19]). It provides an abstraction layer for the Physical Model alongside conventional software simulators for spiking neural networks such as NEST and NEURON. This abstraction layer exposes the configuration of spiking neural networks at a level of individual, configurable neurons and synaptic connections between them. In the case of the NM-PM it hides the complexity of hardware configuration (fig. 2.2.1). This includes the mapping (section 2.13.2), which computes a hardware configuration which represents the network topology given by the user, and the calibration (section 2.13.1), which translates the user-defined neuron and synapse parameters to hardware-specific settings for the analog components (e.g. section 2.3.3.3). The mapping uses the Hardware Abstraction Layer, (section 2.11.2) as the interface to the hardware system as well as to the hardware Executable System Specification (ESS) (section 2.11.4).

The most basic use case for the NM-PM consists in the creation of a `Python` script which defines a spiking neural network using PyNN . The utilized neuron and synapse models have to be compatible with those implemented on the NM-PM, i.e. a leaky integrate-and-fire neuron and conductance based synapses with an exponential kernel (`IF_cond_exp` in PyNN notation) or an adaptive exponential leaky integrate-and-fire neurons and conductance based synapses with an exponential kernel ([15], `EIF_cond_exp_isfa_ista` in PyNN notation). The ranges for neuron, synapse and connectivity parameters are limited by the hardware

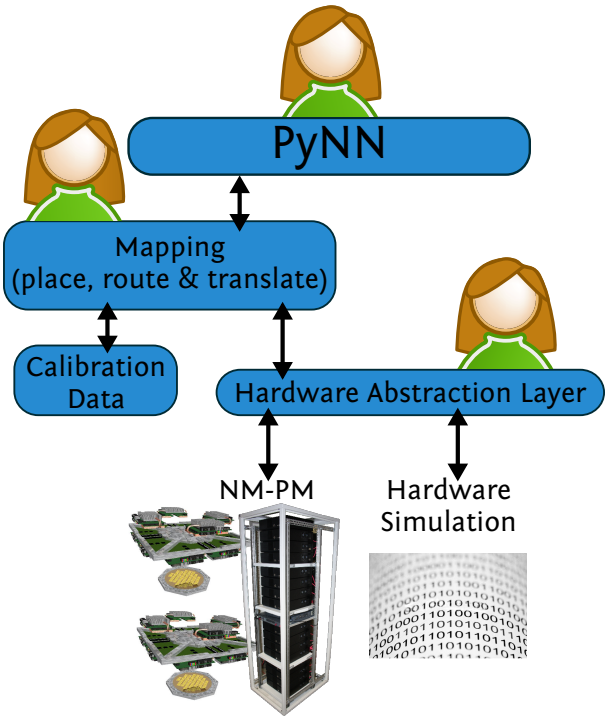


Figure 2.2.1: PyNN is the main user interface for the NM-PM. It hides the mapping and calibration steps from the end-user. Expert users can still access the hardware abstraction layers directly.

implementation.

The results of the simulation can be obtained using the PyNN API with limitations only due to bandwidth constraints (section 2.6.3.6) and, for analog voltage recording, the limitations of concurrent voltage recording given by the hardware system (chapter 2.7).

2.2.2 Low-level user access

The PyNN interface to the NM-PM provides the user with a view of the neuromorphic device which limits the configuration capabilities to a level of abstraction that is shared between neuromorphic and conventional simulators for spiking neural networks. In the case of the NM-PM, the most important features that are not accessible from the level of PyNN are the translation of topology and parameters between the biological model and its representation on the hardware device. For instance, PyNN provides no way to specify which analog circuit on the Physical Model will represent a given logical neuron. The software components that are responsible for this abstraction are the Mapping (section 2.13.2), which handles the topological translation, and the Calibration (section 2.13.1) that performs the transformation of analog parameters.

There are several scenarios in which the user wants to query information from the mapping or control its behavior:

1) The neural network topology can not be fully realized and the user wants to know the number or exact location of unrealized synapses.
2) The placement algorithm provides a suboptimal solution, and a better solution is known.
3) Some components should not be used, e.g., because the tolerance for analog deviations required by the user is lower than the deviation of a specific component.

Equivalently, the calibration output may need to be examined or controlled:

1) The calibration results for a given component need to be assessed.
2) A different calibration method needs to be used for a given use case. Example: synaptic time constants are calibrated by measuring the decay time course of the membrane potential in one voltage range, while the user requires to tune the firing rate for a given stimulus protocol.

Finally, a direct access to the neuromorphic chip is occasionally required. One example would be an evaluation of the influence of a technical parameter on an emulated network, or low-level debugging which still utilizes the mapping and calibration software to create a quick starting point. These tasks can be accomplished using the low level interfaces Hardware Abstraction Layer Backend (HALbe) (section 2.11.2) and Stateful Hardware Abstraction Layer (StH (section 2.11.3), giving the user access to the same level of control that is utilized by the mapping and calibration software.

2.2.3 Real-time interaction with the NM-PM

The NM-PM allows for an operation mode in which the accelerated emulation on the neuromorphic hardware platform interacts with a concurrent, real-time simulation that runs on a conventional computing platform (section 2.8.1). Thus, the simulation of the full model is distributed between neuromorphic and conventional devices. This operation mode differs from the one outlined in 2.2.1: there, experiment results are queried by the software after experiment completion. The need for a separate operation mode arises, because the high acceleration factor in the NM-PM is necessarily shared with the software part of the simulation. This requires an efficient implementation of this software part with as few indirection layers between computation and communication with the hardware device as possible.

fig. 2.2.2 shows the use case for a simulation that requires real-time interaction. A model of a system is defined which contains a spiking neural network and (in general) a non-spiking component, for example a neural system that interacts with a physical environment. The interaction is specified in terms of spikes. This means that, e.g., the computation of firing rates is part of the non-spiking component.

The NM-PM is used to simulate the model as follows: The user provides a description of the emulated neural network in the form of a hardware configuration, e.g., using a PyNN script, and implements the conventional part of the simulation as a software program. The latter uses the real-time API provided by HALbe (see section 2.11.2.1).

On the hardware side, external spikes are configured to be sent to the Compute Node skipping the large spike recording buffers. On the Compute Node, spikes are delivered

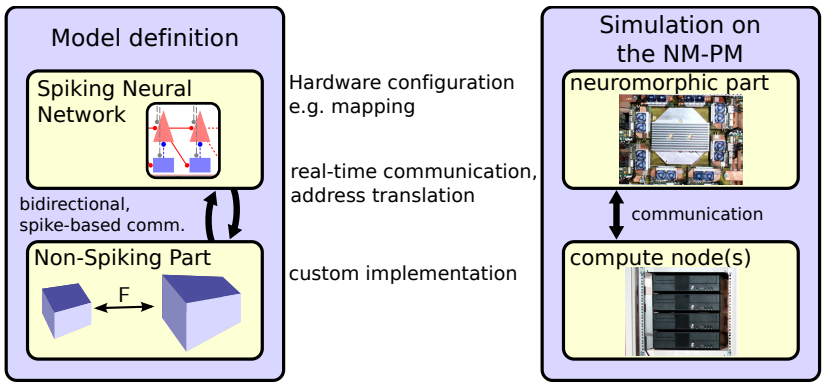


Figure 2.2.2: Hybrid simulation on the NM-PM. A system that consists of a neural network and a non-spiking part (left) is being simulated on the hybrid neuromorphic-classical device (right). The neural network is emulated on the neuromorphic part while the remaining part of the system is simulated in real-time, synchronously and with the same speed-up on a conventional compute node. The communication, which is defined on a spike level in the simulated model, is accomplished via real-time communication between the computational devices. The address translation is taken care of in the software part of the simulation.

to the software implementation, which handles a potentially required translation between hardware and local neuron addresses. Similarly, the custom executable emits spikes that are sent to the hardware device using a low-latency communication channel. Due to the strong latency requirements this operation mode requires exclusive access to all hardware components taking part in the simulation, i.e., no other experiments should run utilize the Compute Nodes, Wafer Modules or network devices partaking in the simulation.

2.2.4 Evaluation Workflow

The gains in emulation speed that arise from the use of an accelerated-time, analog neuromorphic network emulator come at the cost of limitations with respect to neuron parameters, parameter variation, connectivity and communication bandwidths. While the system has been specified to accommodate typical parameter ranges that are employed in models of cortical neural systems [48, 3.12.1], a given model can exceed at least one of those ranges. For instance, a model can require a neuron parameter outside of the supported range or specify the recording of more neurons with a high firing rate than can be accommodated by the allotted communication channels. Thus, a user usually needs to evaluate her model before running it on the NM-PM, e.g., in a large-scale sweep over a parameter range. Simple parameter limits can be checked and enforced at the time of the model definition. Several of the limitation types – e.g., bandwidth limitations of an individual component – depend on the dynamics of a given network model together with the case-specific mapping assignment. For these limitations, a validation on a system simulation level is required instead, which is accomplished with the ESS (section 2.11.4).

Hardware developers profit from the ESS as well, because it can be used as a software system validation tool. Because it uses the output of the mapping, it can detect several classes of logic and configuration errors, such as faulty routing or incorrect settings for switches, repeaters, synapse drivers, synapse addresses, mergers etc.

The possible distortions that can occur when the specified operation range of the hardware device is exceeded, can be classified as follows:

Parameter Limitation A neuron or synapse parameter is required by the model, which lies outside of the supported range on the hardware device. Example: The axonal delay does not correspond to the transmission delay on the hardware device.

Parameter Variation The variation of a parameter is larger than required by the model. Example: the membrane time constant of all neurons is required to be precisely equal, while it differs in the analog circuit due to fixed pattern noise.

Topological Limitation The topology of the model network can not (in principle, or practically) be mapped to the available hardware system. This leads to a network in which a number of synapses has not been realized. Example: An all-to-all connectivity is required for a network that uses all neurons on a single wafer (chapter 2.5).

Bandwidth Limitation The bandwidth of a component that transmits spikes to, from or within an emulated network, is exceeded. Example: Each neuron in a large network receives background stimulus with a high firing rate.

Model Mismatch The neuron or synapse model that is provided by the hardware device, does not correspond to the one required by the network model. Example: The network is defined as a network of leaky integrate-and-fire neurons with current-based synapses, while the hardware device provides conductance based synapses.

An elaborate workflow exists that allows to approach these distortions in the context of a given network model, which is shown in Figure 2.2.3.

The user starts with his network model, or, in the case of the hardware maintainer, with a benchmark library. In addition to each model, a set of performance evaluation measures is defined, which allows to discriminate between successful and unsuccessful execution of the model.

This allows to investigate the distortions listed above individually as well as simultaneously. Individually, distortions are modeled by approximating the distortion mechanism in the PyNN description directly and using a conventional software simulator. A view of the system dynamics is obtained using the ESS to model the dynamic behavior of the hardware system.

A demonstration of this approach with descriptions of possible countermeasures can be found in [17].

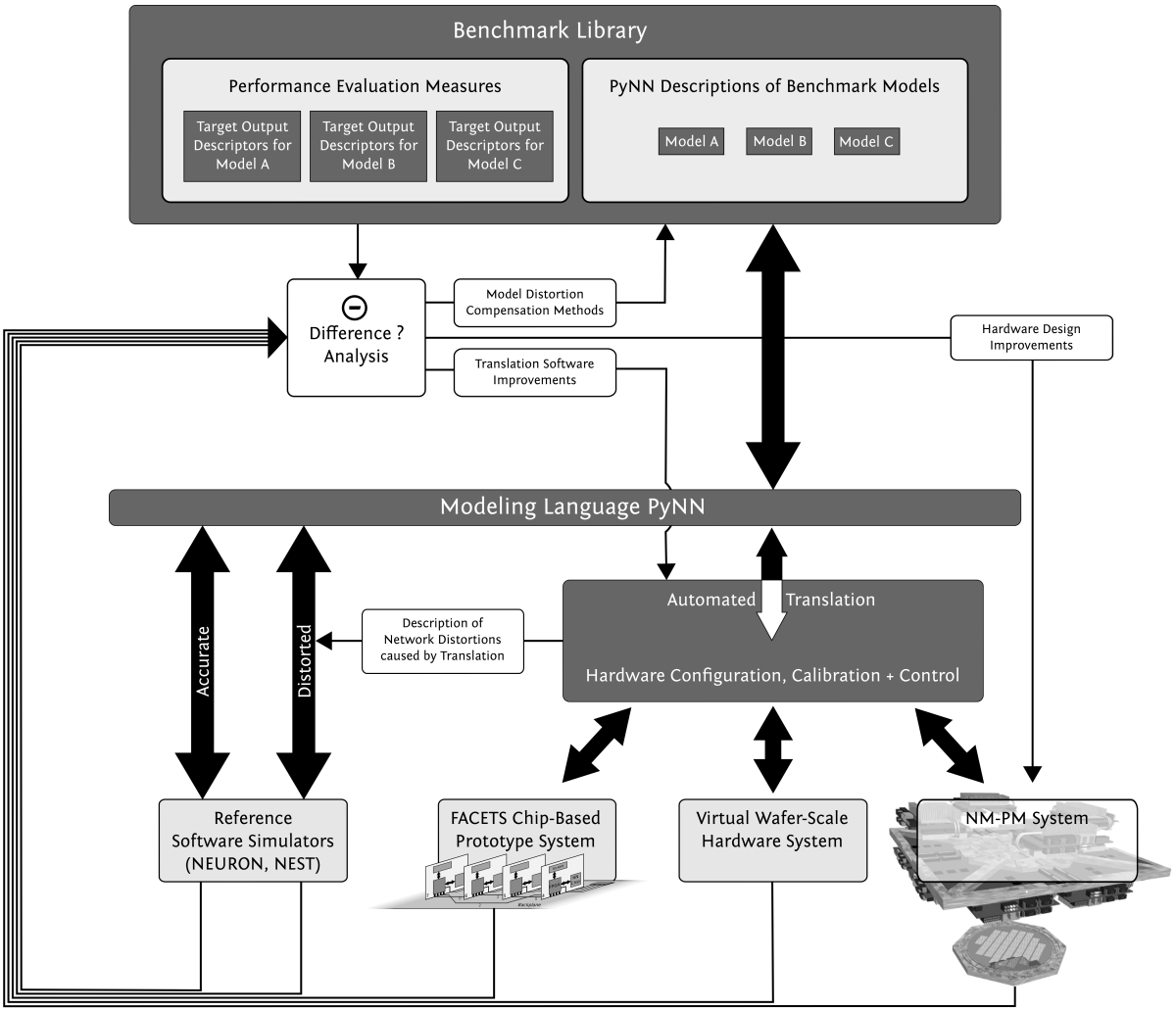


Figure 2.2.3: Hardware and Model evaluation workflow. Description in section 2.2.4. This figure has been adapted from [17].

2.3 Neuromorphic Circuits

2.3.1 Overview

The High-Input Count Analog Neuronal Network Chip (HICANN) is the primary building block for hardware emulation of brain models. It contains the mixed-signal neuron and synapse circuits as well as the necessary support circuits and the host interface logic. Eight HICANN chips are integrated on a single reticle. The size of the HICANN chip is chosen to be 5x10 mm^2. This allows to fully qualify the HICANN in silicon using Multi Project Wafer (MPW) prototyping only, thus limiting cost.

The individual reticles are connected directly on the wafer by depositing and structuring an additional metal layer on top of the whole wafer. The necessary pitch of this post-processing is about 10μm for single layer respectively 20μm for dual layer metal, allowing for a connection density of 103 wires/mm between adjacent edges of neighboring reticles (containing multiple HICANN chips). This will accommodate the maximum connection density existing at the short edges of the HICANN die, where 512 wires need to be interconnected to 256 differential bus lanes. More details of these Direct Wafer-to-Wafer Connectionss (DCONs) can be found in chapter 2.4.

In Fig. 2.3.1 the main functional blocks can be identified. The largest one is the Analog Neuronal Network Core (ANNCORE) which contains 115k synapses and up to 512 neurons. The interconnects between the HICANN chips run vertically and horizontally through the chip, with crossbar switches at their intersections. Additional switch blocks give the ANNCORE access to these signals.

Eight HICANN dies are combined to form the individual reticle of the wafer-scale system. Fig. 2.3.2 shows the connections between adjacent reticles which are created by post-processing the wafer. The reticle is larger than the area occupied by eight HICANN dies (grey border) to accommodate the contact pad windows for the post-processing. Inside the reticle the L1 bus signals of the HICANN dies are edge connected by the topmost metal layer[1]. To achieve the fault tolerance necessary for wafer scale integration each reticle has individual power supplies, JTAG and clock connections to the MainPCB of the Wafer Module. Each HICANN has an individual high-speed serial link to one of the FCP on the MainPCB for fast host communication and packet based event routing. These connections are also realized by post-processing which rearranges the pads of the eight HICANN dies into regular spaced contact rows inside the reticle.

[1] The reticle layout contains small routing structures in between the HICANN dies where edge connection is impossible.

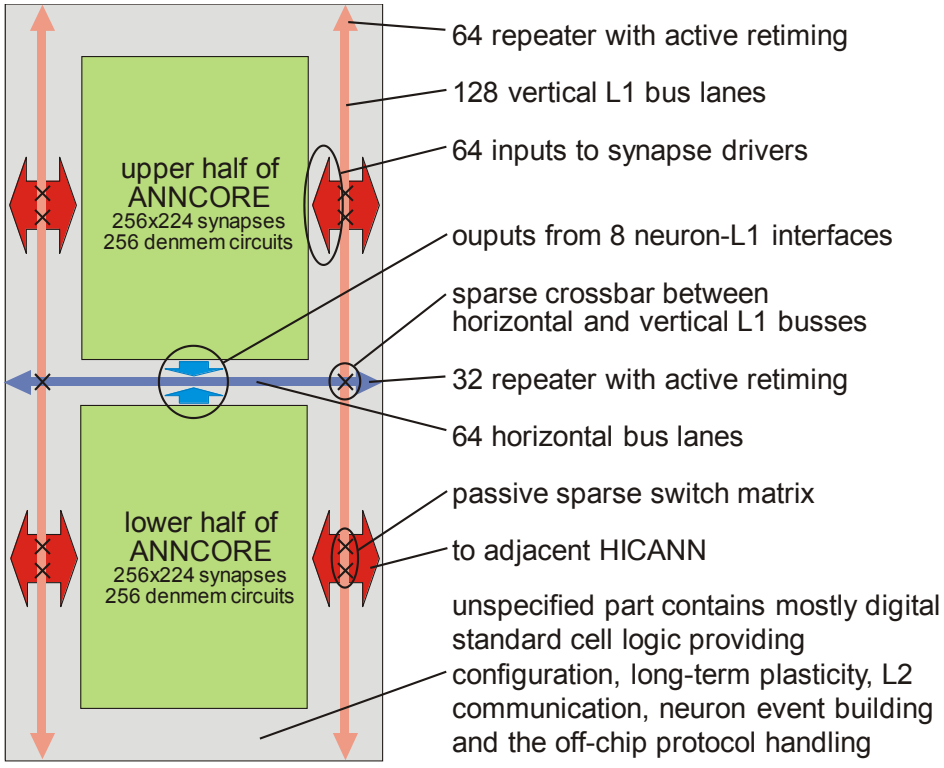


Figure 2.3.1: Block diagram of a HICANN chip.

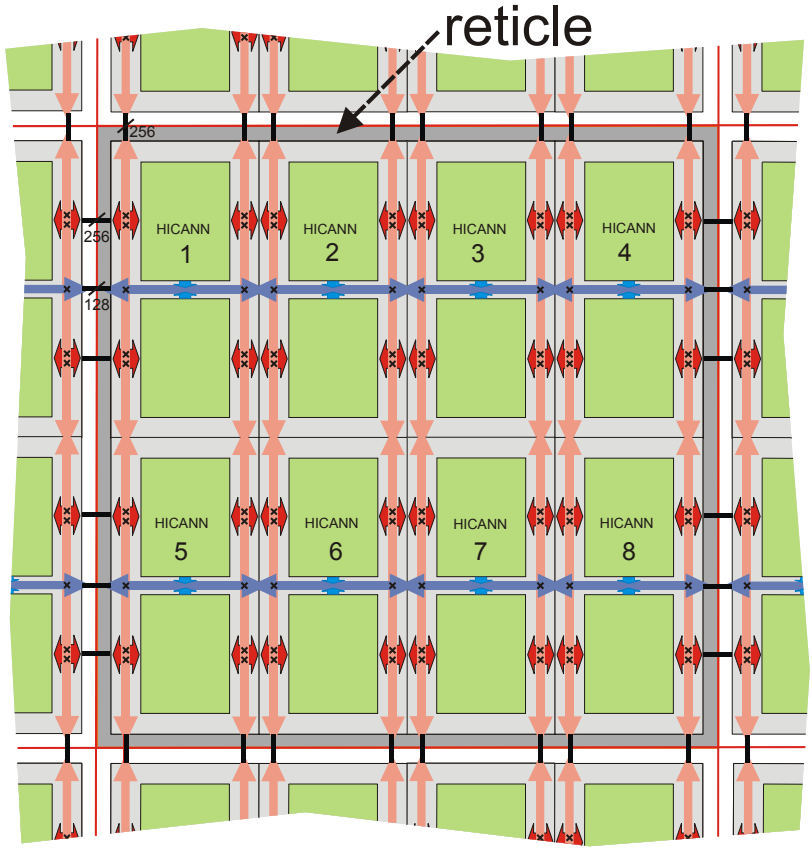


Figure 2.3.2: Wafer-scale connections of a reticle.

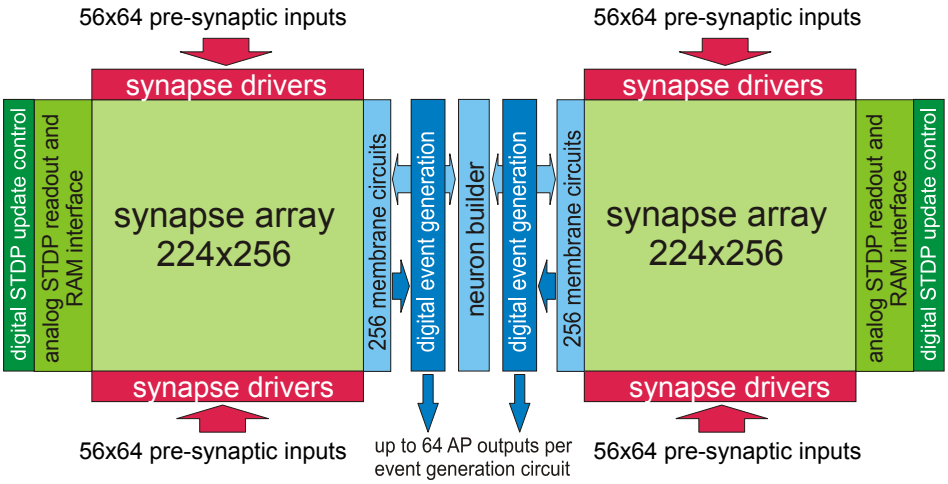


Figure 2.3.3: Block diagram of the analog network core.

Fig. 2.3.3 shows the main elements of the ANNCORE. Its geometry is optimized for a maximum input count of 14k for a single neuron. To allow different neuron sizes, the neurons are built from a set of membrane circuits, each containing full neuron functionality. Each membrane circuit receives input from 224 synapse circuits. By a set of configuration bits up to 64 membrane circuits can be connected together, resulting in a neuron with 14k synpases. The number of realizable neurons of an ANNCORE ranges therefore from 512, using 224 synapses per neuron, down to eight, using only maximum sized neurons. Any combination in-between is also possible. These high number of input signals needs an excessive bandwith: considering the case of a mean firing rate of 10Hz, the maximum acceleration factor of 10^5 and 16k inputs this equals to an average event rate of 164 Gigaevents/s, easily crossing the Teraevent/s barrier in periods of bursty neural activity. Using traditional digital coding techniques an event packet would use about 16 to 32 bit, containing target address and delivery time. To make this communication demand feasible the ANNCORE uses a combination of space and time multiplexing. Due to the high density of the DCONs between the reticles and the on-die wiring between the HICANN chips inside the reticle a large number of signals can be multiplexed spatially. The actual implementation uses 1k wires running alongside the synapse drivers (see Fig. 2.3.1)[2].

To reach the necessary numbers, each of these wires carries the events from 64 pre-synaptic neurons by utilizing a time-multiplexing serial protocol. For historic reasons, this protocol is called the Layer 1 (L1) routing, as opposed to the non-multiplexed local connections used inside previous generation neuromophic chips from Heidelberg which were called Layer 0 (L0). Subsequently, Layer 2 (L2) is the discreet-time event based inter-chip communication layer used between HICANN and the FPGA subsystem.

To further reduce the complexity at the sender as well as the receiver side the event is transmitted in continuous time, i.e. the time of an pre-synaptic event is determined by the moment of its arrival at the synapse driver. The drawback here is the potential timing error introduced in the case of heavy simultaneous firing. The average probability of such a collision happening is determined by the duration of the transmission for an event, the acceleration factor, the number of neurons sharing a wire and the joint firing probability of these neurons. The user can always adjust the first three parameters in a way to accommodate his requirements.

The L1 events are converted into pre-synaptic signals in the synapse drivers located at both edges of the synapse array (see Fig. 2.3.3). The synapse drivers connect to the pre-synaptic inputs of two rows of synapses. Each synapse can decode part of the L1 address to allow synapses connected to different membrane circuits to respond to different pre-synaptic neurons.

The fire outputs of the membrane circuits feed eight output busses running along the central column of the ANNCORE. 64 membrane circutis share one bus.

[2]The wires running vertically are shared between two adjacent HICANNs, therefore each HICANN has access to 1k wires (carrying 512 signals) while implementing only one half of it.

2.3.2 Continuous-Time Layer 1 Communication

This section gives an overview of the components related to the Layer 1 (L1) communication system within the HICANN chip. Fig. 2.3.4 shows a single serial L1 data frame. To transmit the events from up to 64 pre-synaptic neurons an event is encoded using two frame bits and 6 data bits.

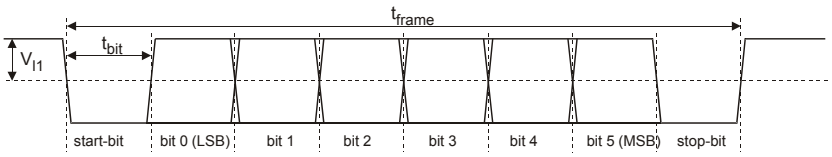


Figure 2.3.4: Timing of a serial l1 data frame. Shown is the differential signal $V_{l1\_pos} - V_{l1\_neg}$.

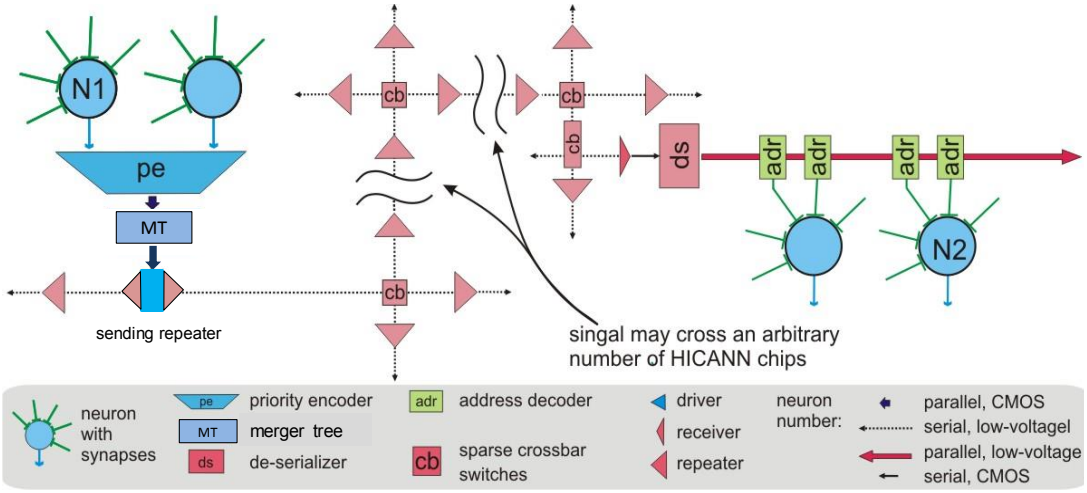


Figure 2.3.5: Schematic diagram of the L1 routing across several HICANN chips.

Fig. 2.3.5 shows how an event is routed across several HICANN boundaries via the L1 routing network. Fig. 2.3.1 depicts the spacial location of the indiviual componetns of the L1 network:

Serial L1 Sender (sending repeater) converts the events generated in the ANNCORE neuron circuits to a serial L1 frame and drives them on a horizontal L1 bus lane

Synapse Driver receives L1 frames and converts them to the pre-synaptic signals

Repeater restores the timing of the L1 signals at the HICANN chip boundary

Crossbar Switch programmable interconnect between horizontal and vertical L1 lanes or vertical lanes and synapse driver inputs

L1 bus lanes horizontal and vertical wires distribute the L1 signals

2.3.2.1 Technical Implementation of the Layer 1 Communication

Differential Signalling and Electrical Considerations

The length of a wire traversing a HICANN die is about 10 mm. This wire will see a total capacitance to its surrounding of about 2 pF[3]. If this load is driven with the full CMOS swing of 1.8 Volts each wire needs a power of $C \cdot V^2 \cdot Events/s$W (considering a simple square pulse as code for an event). For an acceleration factor of 10^5 and a mean firing rate of 10 Hz this equals to 6.5μW. If one scales this up to a whole wafer containing 200k Neurons on about 400 HICANN chips the total power is 1.5kW for the transmission of the neural event signals alone [4]. Clearly this is no feasible solution. Therefore, to limit the power consumption a serial event uses a low-voltage differential wire *l1*, consisting of *l1_pos* and *l1_neg*.

The timing parameters for the typical process corner are: t_{frame}=4ns, t_{bit}=500ps and the differential DC amplitude V_{l1}=150mV. The average number of transitions per event is 5.5, a rounded number of 6 will be used in further discussions. This reduces the total power consumption to 5 Watt (in the case of a differential voltage swing of 100mV). This is a 300-fold reduction compared to the parallel CMOS case.

The resistance of such a wire is $36m\Omega/sq \times 20ksq = 720\Omega$ and the time constant therefore $\tau = RC/2 = 0.7ns$ [5]. To reach a bit rate of 2GBit/s a certain amount of overdrive is needed. The overall geometry of the l1 buses in the HICANN chip show that the effective length is much more than 10 mm. If repeaters are placed along the edges of the chip the worst case for an unbuffered l1 line is 5 mm vertical up to the central crossbar, 5 mm horizontal and 5 additional mm vertical after the crossbar plus two times about 3 to 4mm input lines to the ANNCORE (see Fig. 2.3.1 for reference), branching off the vertical segments. To reduce the total RC-time constant of such a network to a value that can sustain 2 GBit/s the metal width must be increased from the previous example. Simulations have shown that a metal width and spacing of 1.2μm using the thick (2.2μm) metal 6 process option gives satisfactory results for all process corners and worst case routing scenarios. Only one additional provision has to be made: the total parasitic capacitance of de-selected switches in the central crossbars as well as the synapse driver switch matrices must be limited. Therefore these structures are only sparsely populated with switch transistors. See section 2.3.2.6 for further details of the switch arrangement.

Serial Layer 1 Line Driver

The serial data stream is send through a driver using strong pre-emphasis to overcome the large RC time constant. To conserve energy both differential lines are shorted to equalize their potential before the new differential voltage is applied. If the data stream is constant for more than one bit period it is connected to a differential voltage of 100 to 150mV and a common mode voltage of about 750mV. The common mode voltage can be adjusted by the

[3] The considered metal lines had the following parameters: 500 nm width, 800 nm spacing, metal 6, metal 5 orthogonal and only sparsly used, full coupling to metal 4. The dominating capacitance is the coupling capacitance within the layer which accounts for more than 90% of the total capacitance.

[4] In this calculation an event bus uses 6 address bits and 1 strobe bit, the address bits toggle with half the frequency of the strobe signal.

[5] Using a simple model which distributes the total wire capacitance equally at both end of the wire.

external L1 power supply to compensate any PMOS/NMOS imbalance introduced by process variations. This assures that the effective common mode applied by the pre-emphasis driver is the same than the common mode in the dc case.

2.3.2.2 Serial Layer 1 Sender - Sending Repeater

There are two possible sources for an L1 bus: L2→L1 converter or a neuron block. Both use the same circuit as a L1 serializer and sender, called a 'sending repeater'. These sending repeaters are normal L1 repeaters (see below) with the additional functionality of a parallel input.Eight sending repeaters are located in the right horizontal repeater block, therefore, each HICANN can drive Layer 2 (L2) or neuron events on up to eight horizontal L1 buses.

The digital controller of the HICANN uses a clock frequency of $1/t_{\text{frame}}$ generated by an internal Phase-Locked Loop (PLL) from the external reference clock. A Delay-Locked Loop (DLL) in each sending repeater generates the frame timing by dividing the clock period into eight bit periods. The build-in serializer uses the time bins generated by the DLL to produce the bit stream from the neuron number, adding start and stop bits to the data frame.

Since the neurons fire asynchronously, it is not as easy to get the timing reference for the sender DLL. The currently employed solution synchronizes the neuron output to the reference clock. This limits the time-resolution of a neuron to t_{frame}, which results in 40μs at an acceleration factor of 10^4 and $t_{\text{frame}} = 4ns$. These synchronous event are send to the digital controller which contains some additional FIFO and mixing stages, called Merger Tree (MTREE), to allow a flexible allocation of the eight sending repeaters. The L2→L1 converter is part of the MTREE structure.

2.3.2.3 Synapse Driver

Every two synapse rows share an L1 receiver circuit. Since they are alternately mounted left and right from the synapse array, there is one L1 input every four rows, totaling in 56 inputs per side and block.

To implement dynamic synapses a capacitor bank with 64 individual capacitors is implemented within the synapse driver. For each of the 64 possible pre-synaptic neurons of the L1 bus connected to the synapse driver the capacitor stores a voltage which is equivalent to the so-called *recovered partition*. See [59] for more details. The facilitation or depression of the synapse is mediated by the modulation of the default pulse width of the enable signal in accordance with the current value of the *recovered partition*.

Each L1 receiver consists of a differential amplifier restoring CMOS levels from the serial L1 signal. Since this receiver is the only circuit consuming a significant amount of static bias current without anyL1 activity it is optimized for a minimum power consumption with a positive input ($V_{\text{l1\_pos}} > V_{\text{l1\_neg}}$), which is the inactive line level of the L1 bus. Simulations show that in this case its current consumption stays below 100μA at a speed still sufficient for 2 Gbit/s. This is a crucial detail of the L1 implementation since the number of receivers on a wafer is about 220k!

The single ended CMOS L1 signal is used as an input to six dynamic data capture latches and a DLL. The DLL captures the frame timing by aligning the delayed falling edge of the start bit with the original rising edge of the stop bit, thereby dividing the frame in 16 time

bins. For each data bit there is a time bin which lies exactly in the middle of its data eye and is used to trigger the capture latch. After the setup time of bit 6 has passed the data capture latches contain the parallel data word. Any later time bin can be used to trigger further processing in the synapse driver. The DLL signals also allow to produce enable pulses for the synapses with controlled pulse widths. This is necessary for the synapse operation as explained in section 2.3.3.1.

The training phase of the receiver DLL is divided in two phases. A special input circuit masks all transition of the reference signal, which is the original data, outside of an expectation window around the rising edge of the stop bit derived from the DLL. Therefore, in the locked case, the DLL can compensate small timing variations caused by temperature drift or leakage from the control voltage storage capacitor without being disturbed by the additional transitions in the signal caused by the random data payload of the frame.

To achieve the initial lock the data frame must be free of these data transitions. After initialization of the L1 routing topology it is therefore necessary to send a certain amount of dummy events containing only neuron number zero. This is accomplished by digital background generation circuits which are part of the MTREE in the digital control logic 2.3.4.4. In addition to providing the necessary initialization signals for all the DLLs in the L1 signal paths, they can be reused during network operation to provide Poisson distributed background stimuli to the network.

2.3.2.4 Repeater

A repeater is used to compensate for the loss in signal amplitude and timing precision the L1 signal suffers on its way across a HICANN die. At each crossing from one die to another, whether its by edge connection or *direct connect*, a repeater is inserted. It consists of a combination of a receiver similar to the one used in the synapse driver and the serializer and driver circuits.

The L1 re-timing repeater is located at the end of every second vertical and horizontal L1 bus. If two HICANN chips are edge or wafer-scale connected each bus gets a repeater inserted, the odd numbered buses have their repeaters at the right respectively top HICANN, the even ones at the left respectively bottom HICANN.

Repeaters are bidirectional an can be configured to drive either the off-chip or the on-chip part of the L1 bus they are connected to. Each repeater contains eight SRAM bits to configure data direction, crosstalk compensation, power down and debug modes.

The repeater re-samples the data to its own DLL before retransmitting it. This allows the L1 signal to retain its signal quality across the whole wafer. The main source for the degradation of the edge positions is crosstalk from neighboring L1 buses. Two techniques are used to limit its deteriorating effect. First, every second L1 bus is twisted at two locations. The twist is done identically for every affected L1 bus between horizontal L1 lane number 54 and 55 in each synapse switch matrix (see section 2.3.2.6). In a first order approximation, this cancels the crosstalk since the positive and the negative L1 line of the aggressor run in parallel to the victim for the same length.

Each repeater includes two crosstalk cancellation capacitors (FEXT) which can be inserted between adjacent L1 buses. Theses crosstalk cancellation capacitors reduce crosstalk in neighboring receivers. This is essential to allow an arbitrary data direction for each L1 link.

A necessity to group L1 buses by data direction would put an additional burden on the already complex mapping algorithms.

To achieve the necessary bi-directionality each repeater consists of a receiver with an input multiplexer and a driver circuit with duplicated output stages which can be individually enabled or disabled. Disabling both switches off the total repeater circuit, removing quiescent power from all components. Enabling just one is correlated with the input multiplexer and configures the repeater for one data direction.

Depending on data direction FEXT capacitors are either switched between *pos_self* and *neg_neighbour* or *pos_neighbour* and *neg_self*. [6] They can also be disabled altogether. They provide a FEXT compensation by coupling the aggressor signal into the receiver's opposite (non-victim) line. Thereby converting most of the crosstalk from a harmful differential to benign common mode crosstalk.

The repeater's configuration SRAM is always powered. The debug mode works as follows: if enabled, the repeater's intermediate parallel data path is split and connected to two 7 bit debug buses running along the chip periphery. To check crosstalk immunity, two sets of 14 debug data lines are used, connected to the repeaters in a modulo two fashion. The digital control of the chip can receive and send data on these buses with a rate of 250 MHz, which is minimum L1 period as well as the clock frequency of the digital control. Small memories are used within the digital control block to test short L1 sequences without relying on DNC data transfer. This debug feature allows full testing of the L1 routing on the wafer.

The location of the six repeater blocks can be seen in figure 2.3.6. The four blocks at the top and bottom of the chip contain 64 repeaters each, the blocks at the left and right edge contain 32 repeaters each. The addressing in each block follows the counting direction of the connected L1 buses. Address 0 is located at the chip outer edge for the top and bottom blocks, respectively and at the bottom of the horizontal (crossbar) repeater blocks.

2.3.2.5 Neuron to Layer 1 and Layer 2 Interfaces

All L2→L1 interfaces use a two-stage design: the DNC-interface converts the L2 packets into a synchronous, parallel L1 event (spL1, 6 bit parallel data plus a valid bit synchronized to the clock). The spL1→L1 converters are implemented by using eight special repeater circuits located in the left horizontal repeater block (sending repeaters). Their addresses in the left repeater block are 0, 4, 8, etc. and the connectivity to the L1 signals is shown in figure 2.3.6.

The digital part of the interface circuit, called the merger tree, can be programmed to insert events into the spL1→L1 converters which have not been originating in the ANNCORE. *Background Generators* generate L1 events either with a fixed frequency or a Poisson-distribution, their neuron number is always 0.

The test input port of these repeaters serves asSynchronous Parallel Layer 1 (SpL1)input. The synchronously-generated valid signal is used as test clock input for the repeater; automatically supplying the necessary L1 timing reference.

To keep the DLLs of all repeaters locked the *Background Generators* are enabled, their neuron number 0 makes sure that every DLL in the L1 path can lock to the signal.

[6] Alternatively, the FEXT compensation capacitors can be connected between *pos_self* and *pos_neighbour* or *neg_neighbour* and *neg_self*.

The neuron→spL1 interface generatesSpL1data packets from the asynchronous fire events of the neuron circuits. 32 adjacent neurons on the upper and the lower half respectively share one neuron→spL1 interface circuit. It consists of a 64 bit priority encoder, selecting a single neuron and generating the according binary number (which is programmable by a six bit SRAM) together with the valid signal. Neurons with lower priority store their requests and are served after the serializer has send the higher priority one. A two stage flip-flop bank synchronizes the parallel signals for further processing as SpL1data. The maximum event rate is 125MHz. The resulting time-resolution in biological real-time is 40μs.

2.3.2.6 Crossbar and Synapse Driver Switch Matrices

Only odd-numbered horizontal L1 lanes can be connected to SPL1 repeaters in one row of HICANNs on the wafer. The layout of the crossbar switch matrices is organized in a way that even-numbered lanes on a HICANN in a different row can be used for horizontal distribution ofL1 signals in the according HICANN row if they are connected by a vertical L1 bus.

Fig. 2.3.6 shows the L1 connectivity of two edge-connected HICANNs. Vertical connectivity is established by direct stacking of two such rows. Switch transistor positions are shown as block dots. The switches on even-numbered horizontal L1 lanes can be used to horizontally distribute L1 signals that originate on a vertically adjacent HICANN. The location and indexing of nets/ports is identical to the physical implementation in HICANN. The dark grey boxes are: Crossbar switch matrix (5): left, (2): right. Synapse switch matrix (1): top right, (3): bottom right, (4): top left, (6): bottom right. Repeater block (A): top right, (B): bottom right, (C): center right, (D): center left, (E): top left, (F): bottom left. One repeater is denoted as lighter or darker rectangle (containing one driver and one pass-through) in the repeater blocks. Note: In the HICANN chip, one of the two permutation lines is located at the top, the other a the bottom. Similarly, instead of two permutations lanes at the right edge, one is located at the left and the other at the right side, respectively. The figure does not reflect this fact, since it is transparent to the user and does not affect connectivity.

The permution of a single line only cannot be implemented using the L1 repeater circuits driving every second L1 lane per chip edge (see section 2.3.2.4).

Figures 2.3.7, 2.3.8 and 2.3.9 show the switch locations within the crossbars and synapse switch matices.

Crossbar Each 128 lane vertical block contains 4 equally spaced switches per horizontal lane resulting in a step size of 32 lanes. Each vertical lane has two switches per crossbar connecting an even-numbered and the according odd-numbered horizontal lane. I.e. transistors are located at positions 0, 32, 64, 96 in horizontal L1 line 0 and 1 and at positions 1, 33, 65, 97 in horizontal L1 line 2 and 3. The configuration is read/written row-wise: Each access writes/reads 4 bit (transistor set(1) or unset (0)) with bit 0 corresponding to the lowest vertical lane number.

Synapse Driver The synapse driver switch matrices connect the vertical L1 lanes to horizontal lanes which are connected to local synapse drivers and to lanes connected to the adjacent HICANN (see Figure 2.3.6). The switch transistor layout for local and adjacent HICANN is identical for two horizontal lanes at a time. Local and adjacent connectivity

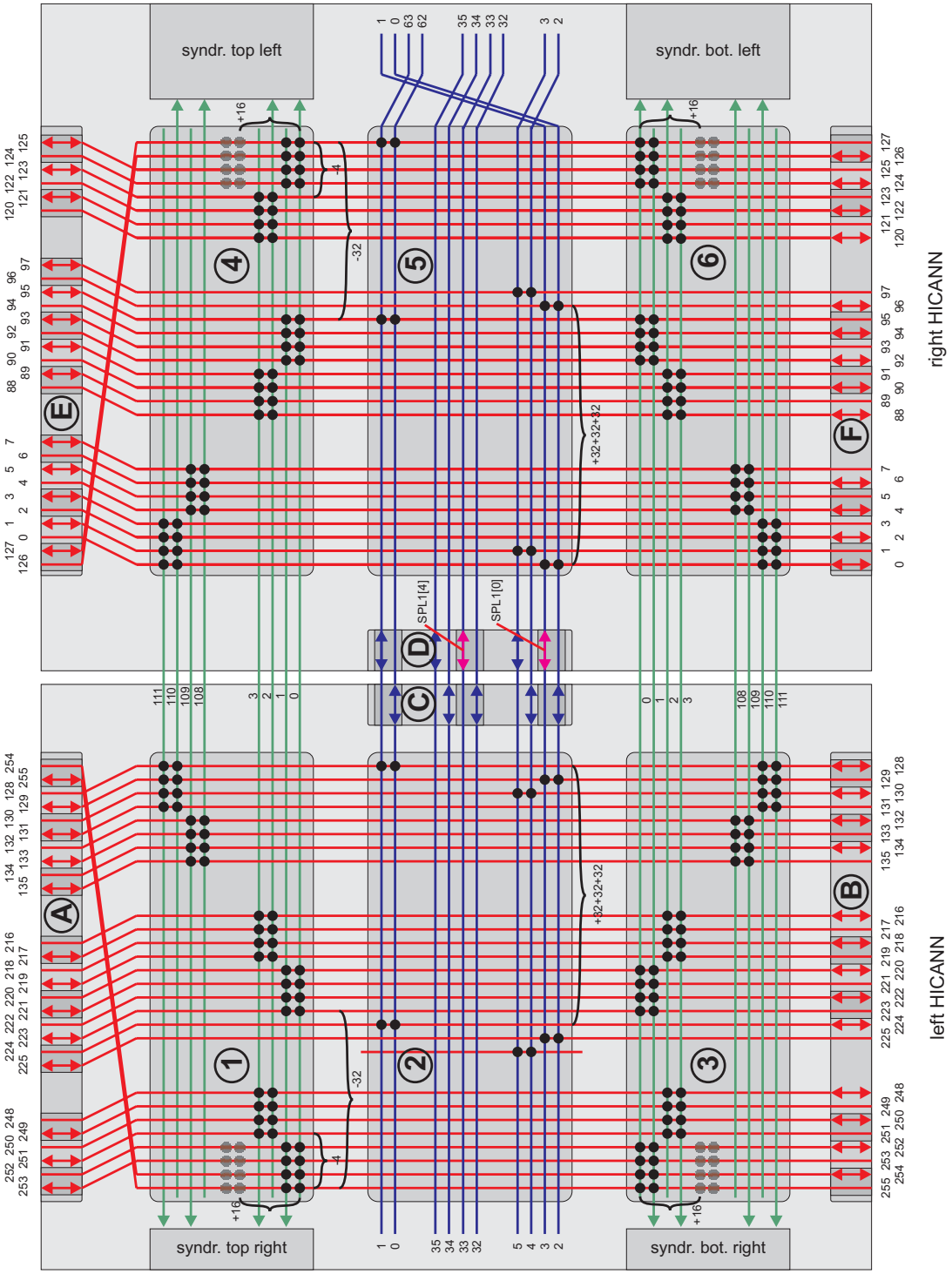


Figure 2.3.6: L1 connectivity of two edge-connected HICANNs. See text for detailed description.

is swapped between left and right side: Horizontal lane 0 connects to the local driver in the top left synapse driver switch matrix and to the adjacent HICANN in the top right switch matrix. In each block of 32 vertical L1 bus lanes, a synapse driver can connect to four consecutive lanes. With each synapse driver row, the position is incremented by four. This results in the following pattern (see Figure 2.3.6): synapse driver row 0 (in terms of fc_anncore) connects to vertical columns 0-3, 32-35, 64-67, 96-99; row 1 connects to vertical columns 4-7, 36-39, 68-71, 100-103 and so on. The configuration is again written row-wise: Each access writes/reads 16 bit (transistor set(1) or unset (0)) with bit 0 corresponding to the lowest vertical lane number. The number of row addresses (112 per synapse block and side) is twice the number of synapse drivers since two rows address local and adjacent connections as described above. These two rows can be configured independently.

Inter-HICANN-connections In figure 2.3.6 the permutation scheme for the L1 connections at the HICANN boundaries is shown. By permuting the whole vertical and horizontal buses by two lanes all lanes are equally accessible on the wafer scale despite the fixed connection schemes used in the individual HICANN chips.

2.3.2.7 L1 Pinout of the HICANN Chip

The pinout of the HICANN chip follows the L1 lane numbers illustrated in Figure 2.3.6. Note the flip of the two highest-value lanes in the vertical and right blocks due to the permutation of these lanes inside the HICANN chip. Therefore, it can be directly edge-connected in the reticle without additional routing.

Differential pinout: At the top and bottom edge, the signal lines facing the edge of the chip are the positive signals of the differential L1 pairs. At the left and right edge the lowermost signals are the positive signals of the differential L1 pairs.

2.3.3 Analog Neural Network Core (ANNCORE) circuits

2.3.3.1 Synapse drivers

The synapse drivers are the interface between the serialized event data and the synapse array (see Fig. 2.3.3). They contain the deserializer and data capture circuits described in subsubsection 2.3.2.3.

The lower four bit of the sampled neuron address are subsequently transmitted into the synapse array. The upper two are compared to stored addresses for a set of strobe lines for the synapse address decoders. The length of these strobe pulses encode for the momentary value of the so called *recovered partition R* [59] that controls the magnitude of the synaptic transconductance if it is modulated by short term plasticity mechanisms (short term depression or facilitation: STDF). The circuit used here resembles the one in [59] with the major exception that the storage capacitor for *R* needs to be replicated 64 times, since the value of *R* is independent for each pre-synaptic neuron.

The DLL provides the necessary timing information to reliably control the strobe pulse length. The length of the strobe pulse is limited to a L1 frame period since only one

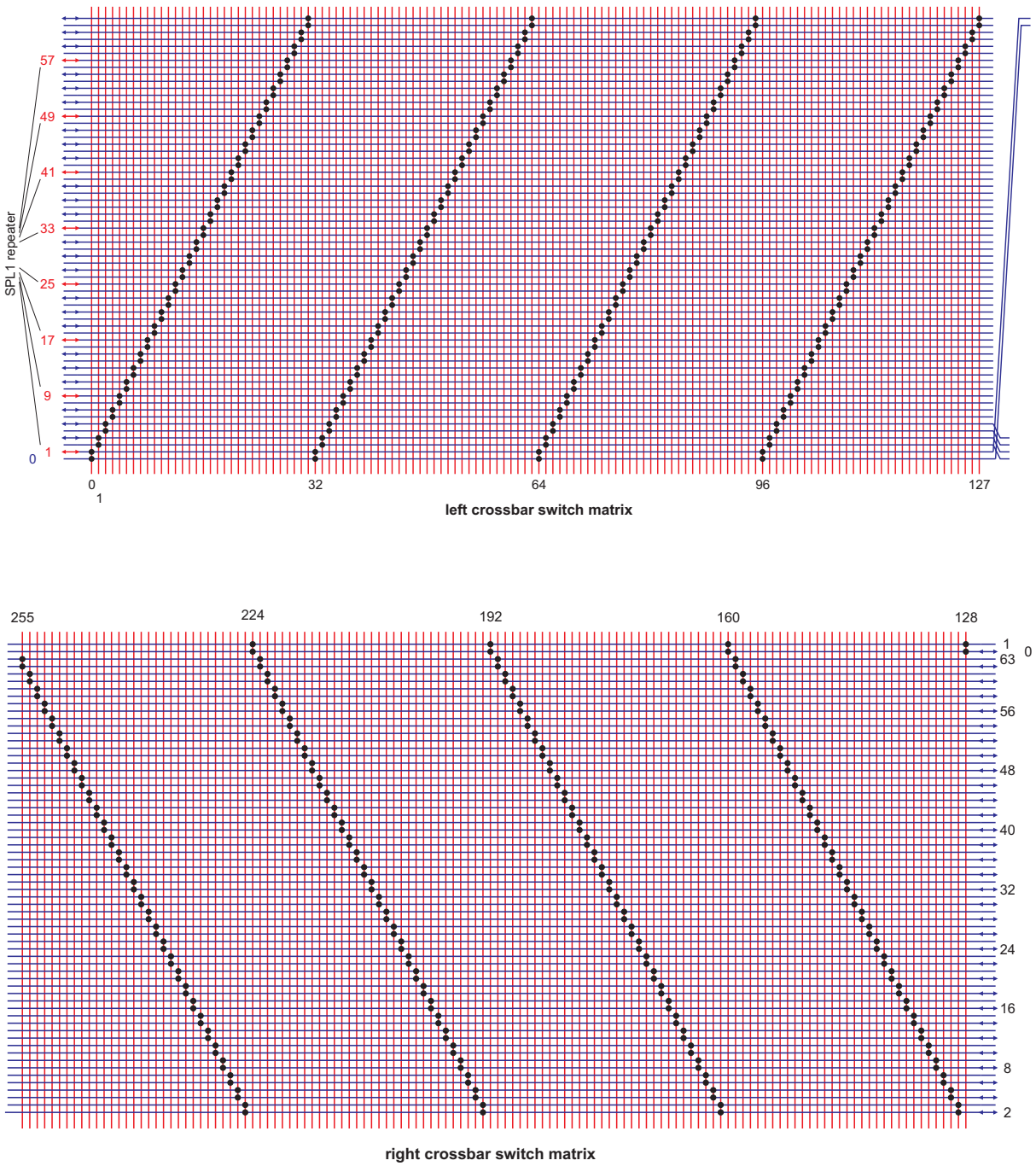


Figure 2.3.7: Left and right crossbar switch matrix in HICANN. Note the numbering of the horizontal lanes and the permution.

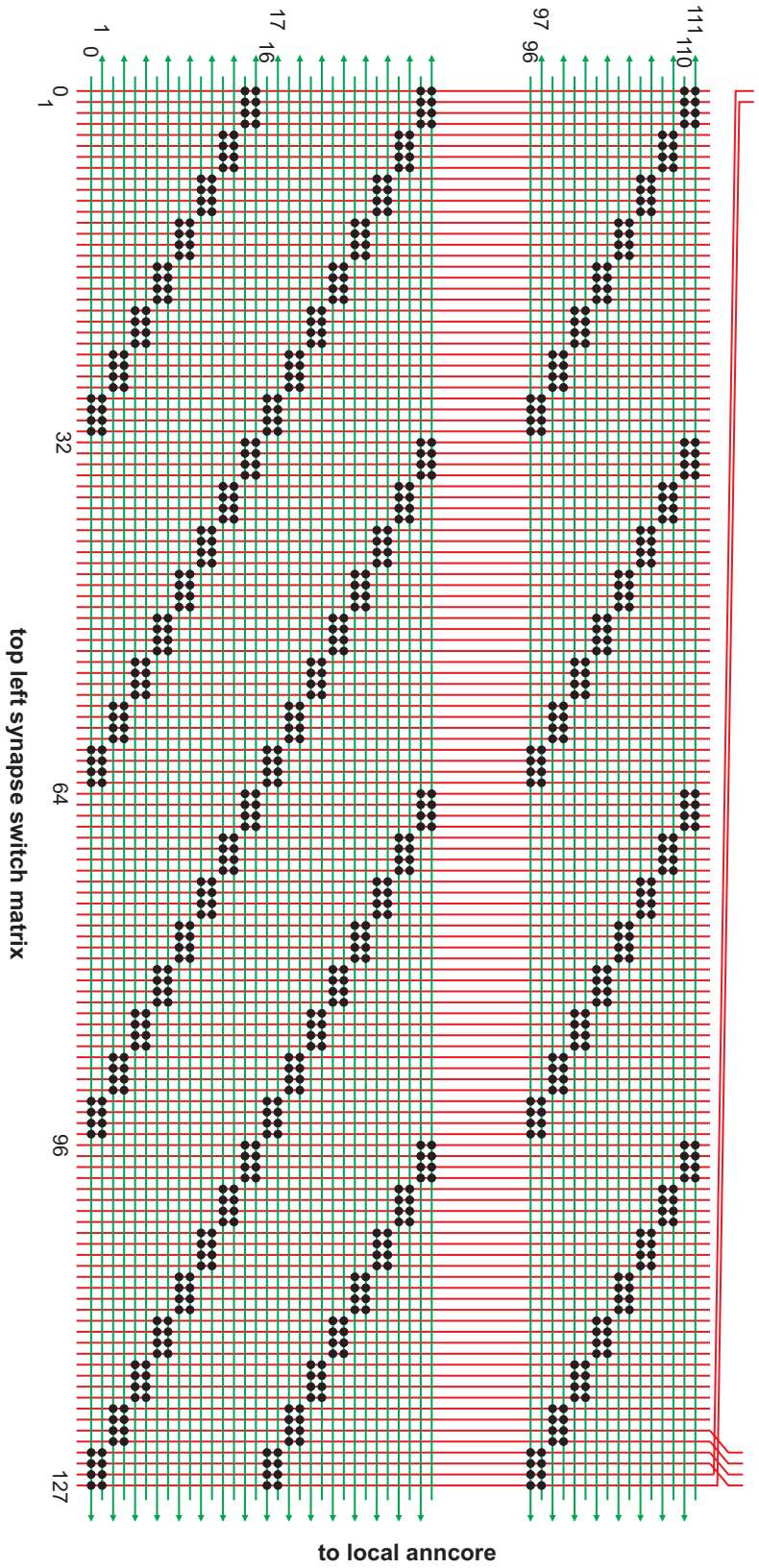


Figure 2.3.8: Top left synapse switch matrix in HICANN. The bottom left synapse switch matrix is an exact copy mirrored on the horizontal axis.

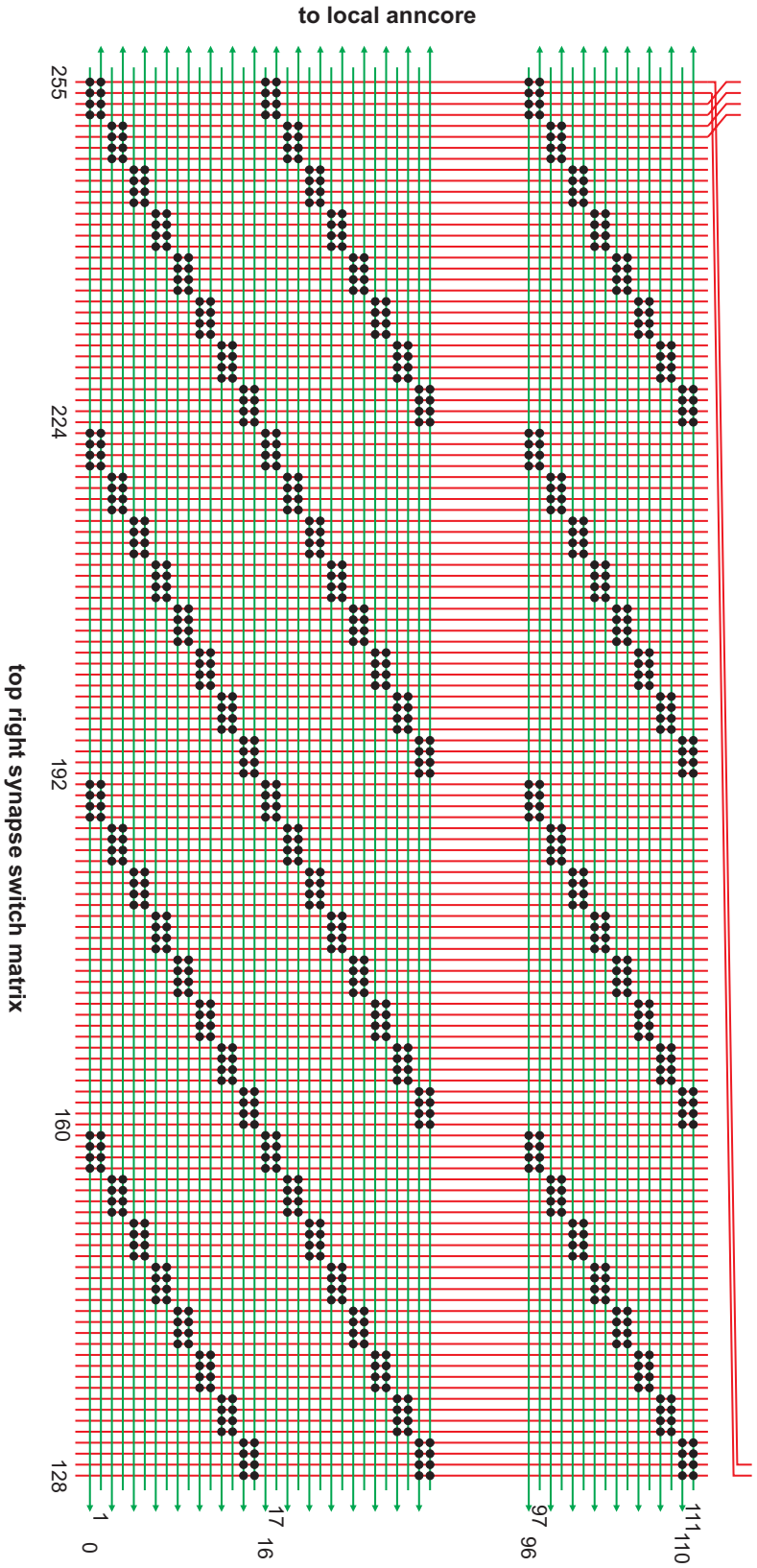


Figure 2.3.9: Top right synapse switch matrix in HICANN. The bottom right synapse switch matrix is an exact copy mirrored on the horizontal axis.

preout	synapse connection (in each four synapse cell)
0	lower left
1	upper left
2	lower right
3	upper right

Table 2.3.1: Mapping of syndriver pre-out signals

concurrent pulse can be transmitted into the synapse array.

To limit the power consumption and crosstalk of the parallel L1 data a reduced voltage swing of 1/2 Vdd (0.9V) is used inside the synapse array. The lower four address bits will be decoded in the synapses. They are therefore transmitted pseudo-differentially which also reduces crosstalk significantly.

To control the capacitive loading of the vertical L1 busses only a certain number of switches and activated connections to the Syndrivers is allowed. To share these signals between adjacent Syndriver rows, neighbouring Syndrivers have a bypass switch between their inputs, allowing to form vertical chains of syndrivers sharing one common connection to the vertical L1 busses.

2.3.3.2 Synapses

The synapse circuits are an enhanced version of the ones reported in [60]. The major change is the inclusion of a four bit address decoder replacing the single pre-synaptic signal used previously. Each synapse has a fixed connection to one of the strobe signals from the synapse driver and a programmable four bit address. Table 2.3.1 lists the mapping of the syndriver pre-out signals to the synapse array.

This allows for a much higher mapping efficiency in the case of sparsely connected random networks. The fixed maximum conductance g_{max} for the synapse Digital-to-Analog Converter (DAC) can still be set row wise by a programmable analog parameter. The output signal of a synapse in the case of an input event matching its address is as follows: a square current pulse with the amplitude $weight \times g_{max}$ and the length τ_{STDF}.

2.3.3.3 Membrane Circuits

A neuron is formed by connecting together up to 64 Dendrite Membrane Circuit (DenMem) circuits[7]. Each DenMem contains a set of ion-channel emulation circuits connected to the membrane capacitance. These ion-channel circuits represent the following membrane currents:

- Excitatory synapses
- Inhibitory synapses

[7] This section is only a short summary of the neuron implementation. For a more detailed description please reference either [49] or [48].

- Leakage
- Adaptation
- Spike generation (exponential behavior of membrane potential)

A generic ion-channel circuit was developed that is used for the first four mentioned classes of membrane currents. The exponential is build using an operational amplifier as it is no direct conductance. These circuits allow the implementation of the *adaptive exponential integrate and fire* model [15].

Neighbouring DenMem circuits can be tied together by shorting their membranes' potential. The spike detection circuit is enabled in one selected DenMem only and its output signal propagates to all interconnected DenMems of the neuron. As this pulse is transmitted back into the synapse array for *STDP*, it could be understood as a model of a back propagatin action potential, but as there is no correlation between the propagation delay in biology and the delay caused by digital transmission, this is only an artifact. It is planned to add a controlled back propagating action potential delay in the future as part of a multi-compartment neuron implementation.

The digital spike pulse is not only transmitted to the synapse array, but also to the priority encoder located in the digital event generation circuits in the center of each HICANN[8]. It is possible to enable or disable this transmission for each DenMem[9].

2.3.3.4 Additional Features of the Denmem-Block

Each HICANN includes two analog readout channels. Within each DenMem circuit an operational amplifier with output-enable is located. Every second denmem connects to one of two common output lines. This setup allows the simultaneous recording of any pair of adjacent DenMem membrane voltages. Two 50Ω output buffers drive these signals off the HICANN die.

One DenMem circuit per HICANN can be connected to a test current generator. This facilitates neuron characterization. It reuses resources from the floating gate parameter storage and allows the replay of short time-varying current patterns.

2.3.3.5 Single-Poly Floating Gate Analog Parameter Storage

The principle of the single-poly floating gate analog paramter storage is covered in the report *"Analog Floating Gate Memory in a 0.18µ Single-Poly CMOS Process"*. This subsubsection will specify the integration in the HICANN chip. The following analog parameters are required in the HICANN system:

1) neuron parameters
2) STDP parameters

[8] Eight asynchronous priority encoder with 64 inputs each determine which action potential is transmitted back into the network (see section 2.3.2.5).

[9] In future implementations a variable delay could be implemented at this point. This would allow emulating the transmission delay observed in biology at long range axonal connections. Each neuron could have as many different delays as the number of denmem circuits it consists of.

3) synapse driver parameters

4) global parameters outside the ANNCORE (repeaters, off-chip buffers, etc)

Only the neurons (i.e. the DenMem circuit) get individual parameters. Four floating gate parameter storage blocks are located beneath the DenMem circuits. Each block consist of 129 columns with 24 storage cells each. While columns 1 to 128 generate the bias currents and voltages of one half of an DenMem block, column 0 generates the parameters used for STDP, synapse drivers and global purposes called *global parameters* below. A custom rail-to-rail buffer has been developed to drive these signals.

As there are two mirrored floating gate arrays for the upper respectively the lower half of the HICANN while there is only one DenMem block symmetry is broken. Right and left floating gate arrays need a dedicated mapping to the neuron parameters. Table 2.3.3 and 2.3.4 show the mapping of floating gate parameters to neuron and global parameters.

All bias inputs use the same current range: 100nA to 2μA. The floating-gate cells can cover this range for all process corners. In the typical case the maximum current is about 2.5μA. The programming logic provides 10 bits of resolution for the nominal 2.5μA range and at least 8 bit precision[10]. All current cells are current sources. To generate the dedicated biasing currents, current mirrors are used for parameter mapping.

The voltage cells can provide the full 1.8 Volt swing by using a readout transistor connected to the floating gate as source follower. The bias current used by the source follower is set to 1μA. The offset of the source follower is compensated automatically, since the programming logic uses the same source follower to read back the cell's voltage while programming.

To allow fast parameter changes all cells providing the same current/voltage in one floating gate block, i.e. the cells located in the same line, can be programmed in parallel. Also all four floating gate blocks are independent of each other and can therefore be programmed at the same time.

[10]The maximum error is always smaller than 4nA, which is about 1/2 LSB at 8 bits.

Parameter	Type	Function and multiplicator	Range (tech)	Range 10^4
I_{gl}	neuron	g_l1:1, 1:3 or 1:33	33 nA to 666 nA	400 nS to 4 uS
E_l		E_l		0 to 1.4 V
$I_{gladapt}$		a 1:1, 1:3 or 1:33	33 nA to 666 nA	400 nS to 4 uS
I_{radapt}		τ_w 1:11, 1:187 or 1:328		1 nA to 12.3 nA
I_{fire}		b		upto 4 uA
V_{exp}		V_t		
I_{bexp}		V_t, Δ_t		
E_{syn}		reversal potential		
I_{conv}		maximum synaptic conductance		usually set to max
V_{syntc}		synaptic time constant	1.25 V to 1.45 V	
V_{syn}		voltage level of line to synapse array	set to 1 V	
I_{int}		synaptic input integrator bias	set to 2 uA	
I_{pl}		refractory periode	50 ns to 500 ns	500 us to 5 ms
V_T		Θ		0 to 1.4 Volt
V_{reset}	neuron global	reset voltage		0 to 1.4 Volt
I_{reset}		current used to pull down membrane at reset		usually set to max
V_{bout}		bias for neuron output amplifier	2 uA	
V_{bexp}		bias for buffer of V_{exp}	2 uA	
intern OP bias		bias of floating gate array amplifiers	2 uA	
V_{dllres}	layer1	dll reset voltage		
$V_{ccas,cbias}$		amplifier bias		
V_{fac}	synapse driver	facilitation bias voltage		
V_{dep}		depression bias voltage		
V_{gmax}		synaptic current		
V_{bstdf}		stdf bias		
V_{stdf}		?		
V_m	synapse	V1 in [60]		
V_{clr}		V2 in [60]		
V_{cla}		V2 in [60]		
V_{thigh}	stdp readout	threshold causal?		
V_{tlow}		threshold acausal?		
V_{br}				

Table 2.3.2: Floating gate parameter description summary

line	fg parameter	global parameter	neuron parameter
0	voltage0	V_{reset}(even)	not connected
1	current0	intern OP bias	I_{bexp}
2	voltage1	V_{dllres}	not connected
3	current1	V_{bout}	I_{convi}
4	voltage2	V_{fac}	not connected
5	current2	I_{breset}	$I_{spikeamp}$
6	voltage3	V_{dep}	E_{l}
7	current3	I_{bstim}	I_{fire}
8	voltage4	V_{thigh}	V_{syni}
9	current4	$V_{gmax}<3>$	$I_{gladapt}$
10	voltage5	V_{tlow}	V_{syntci}
11	current5	$V_{gmax}<0>$	I_{gl}
12	voltage6	V_{clra}	V_{t}
13	current6	$V_{gmax}<1>$	I_{pl}
14	voltage7	V_{stdf}	V_{syntcx}
15	current7	$V_{gmax}<2>$	I_{radapt}
16	voltage8	V_{m} shared	E_{synx}
17	current8	V_{bstdf}	I_{convx}
18	voltage9	not connected	E_{syni}
19	current9	V_{dtc}	I_{intbbx}
20	voltage10	not connected	V_{exp}
21	current10	V_{br}	I_{intbbi}
22	voltage11	not connected	V_{synx}
23	current11	V_{ccas}, V_{cbias}	I_{rexp}

Table 2.3.3: Floating gate parameter mapping for left side

line	fg parameter	global parameter	neuron parameter
0	voltage0	V_{reset}(odd)	E_{synx}
1	current0	intern OP bias	I_{convi}
2	voltage1	V_{dllres}	E_{syni}
3	current1	V_{bexp}	I_{convx}
4	voltage2	V_{fac}	V_{exp}
5	current2	I_{breset}	I_{intbbx}
6	voltage3	V_{dep}	V_{synx}
7	current3	I_{bstim}	I_{intbbi}
8	voltage4	V_{thigh}	V_{syntci}
9	current4	$V_{gmax}<3>$	I_{pl}
10	voltage5	V_{tlow}	V_{syni}
11	current5	$V_{gmax}<0>$	$I_{gladapt}$
12	voltage6	V_{clrc}	V_{syntcx}
13	current6	$V_{gmax}<1>$	I_{rexp}
14	voltage7	V_{stdf}	V_{t}
15	current7	$V_{gmax}<2>$	I_{bexp}
16	voltage8	V_{m} shared	E_{l}
17	current8	V_{bstdf}	$I_{spikeamp}$
18	voltage9	not connected	not connected
19	current9	V_{dtc}	I_{fireb}
20	voltage10	not connected	not connected
21	current10	V_{br}	I_{gl}
22	voltage11	not connected	not connected
23	current11	V_{ccas}, V_{cbias}	I_{radapt}

Table 2.3.4: Floating gate parameter mapping for right side

2.3.4 Digital Control

The top level HICANN digital control consists of the following parts:

- General system control
- Layer 2 (L2) event handling circuits
- Link protocol handling for the high-speed serial links to the FPGA Communication PCB (FCP)
- Configuration Interface (with internal bus fabric)
- Configuration Modules (connected to the bus fabric)

2.3.4.1 General system control

The general system control portion of the digital part covers reset handling, clock generation and system time counter control, as well as slow control access by means of a Joint Test Action Group (JTAG) interface.

JTAG Interface

The physical JTAG interface is connected to a standard JTAG Test Access Port (TAP)-controller that is capable of receiving instructions and shifting data in and out. All available instructions, together with a functional description, are listed in table 2.10.26. Two types of instructions can be distinguished:

- Control instructions: These are for example used to set up clock frequencies, the system time counter, or for high-speed link initialization. Parts of the design like each individual Automatic Repeat Request (ARQ) instance can also be reset via JTAG. All control registers have JTAG reset values which are described in section 2.10.5.
- Debug instructions: These include read back of (Cyclic Redundancy Check (CRC)) error or timeout counters or other status values. There is also a possibility to inject or read back configuration or pulse data payload into/from the 64 bit data buses of the Digital Network Chip Interface (DNC Interface). This way communication with the HICANN is possible, even without a correctly working high-speed connection. Since JTAG communication has no deterministic timing, pulse communication is only reasonably possible with time stamp processing disabled (see section section 2.3.4.2).

Within one 8-HICANN reticle, the JTAG ports of the chips are daisy-chained as described in section 2.4.2.1. With each HICANN having a 6 bit instruction register, this results in a 48 bit wide instruction register on each reticle.

Clock Generation and System Time Counter

HICANN has a differential clock input EXT_CLK_(P/N) that must be driven with a 50 MHz clock signal that adheres to the Low-Voltage Differential Signaling (LVDS) signaling standard. Two PLLs are present on the chip and receive input from this external clock signal. One Custom Phase-Locked Loop (FC-PLL) and an off-the-shelf PLL which is called Faraday Phase-Locked Loop (F-PLL). It is provided with the digital standard cell library that has been used for the digital part of HICANN ([25, 26]). Operation of both PLLs can be disabled and controlled by the according JTAG instructions in table 2.10.26.

One output of the FC-PLL provides a 250 MHz clock with a fixed ratio of 5:1 wrt. the input clock. This clock is used to run a 15 bit wide system time counter that is present in each HICANN. Together with a counter in the FPGA that is derived from the same global clock source, these counters are the time base for pulse delivery and recording. The FC-PLL also provides clock outputs that drive the high-speed interface logic. These can be switched between low-speed (1 Gbit s^{-1}) and high-speed mode (2 Gbit s^{-1}) using the JTAG commands in table 2.10.26.

The reminder of the HICANN is clocked by the F-PLL. While its single output can be tuned to a wide frequency range using JTAG commands from table 2.10.26, the allowed clock frequency range is 100 MHz to 250 MHz. Two clocks are derived from this output:

- An 1:1 version that drives the on-wafer pulse communication circuitry and all logic related to the digital pulse transport without time stamps (see section 2.3.2). The clock frequency can be set lower than the target value of 250 MHz in order to increase the reliability of the on-wafer communication.
- An 1:4 version (up to 62.5 MHz) that drives all the remaining core logic.

Asynchronous First-In First-Outs (FIFOs) are used to transfer data between the different clock domains. These FIFOs are located directly behind the resp. ports of the DNC Interface logic.

Reset Handling and Counter Synchronization

The HICANN has a low-active reset input RESET_N that mainly serves as a power-up reset and must be kept low (i.e. active) during power up. All internal Finite State Machines (FSMs) are reset by this signal to their idle state and the chip will leave idle state only upon external input. Data path and configuration registers are not necessarily cleared. Reset values of registers that are being cleared by this reset are given in the resp. sections. A second reset mechanism ist the test logic reset signal of the chip's JTAG controller. Reset values of relevant control registers are given in section 2.10.5.

System time counter start The RESET_N input also clears the value of the system time counter to zero. This counter is then armed and waits for the global SYS_START signal to be toggled. This signal needs to be toggled at all HICANNs that are part of a large neural network within one EXT_CLK period in order to have their counters synchronized. Since the SYS_START signal is shared with the JTAG TMS signal on one HICANN pin (see section 2.4.2.1),

it has to be assured that no JTAG activity takes place before issuing SYS_START. The correct (power-up) reset sequence is described in the following paragraph.

Reset and synchronization sequence Since the JTAG reset resets the FC-PLL and F-PLL (thus, core clock frequency) to default values and disables all high-speed communication there is a strong interdependency between JTAG reset and the design reset. Together with the system time counter start requirements described in the previous paragraph, the following power-up reset sequence can be identified:

1) Power on with reset active, or activate reset in case a reset cycle is carried out.
2) Issue JTAG reset
3) Set F-PLL clock frequency
4) Release reset
5) Toggle SYS_START at **all** HICANNs in the system

After this sequence, the chip has been correctly reset and all counters are running synchronously. All further initialization can then be done according to the actual use case.

2.3.4.2 DNC interface and Layer 2 circuits

Overview

The Layer 2 circuits on the HICANN are responsible for the communication with the FPGA Communication PCB (FCP) and also for the access to the external world. All Layer 2 circuits are grouped in a module named *DNC interface* which might be used a synonym for the Layer 2 circuits throughout this section.

The transmission provides two different packet formats to be able to transport pulse event data and configuration data. A block diagram in fig. 2.3.10 presents the setup of the Layer 2 Communication circuits.

Incoming configuration data is forwarded to the on-chip bus fabric based on the Open Core Protocol (OCP)[11] bus which targets the ARQ protocol (described in section 2.3.4.3). Incoming pulse events are handled by the "Layer 2 to Layer 1 interface" (`l2tol1_if` for short), where the events are delayed up to the target release time and finally released on the target Layer 1 bus. The opposite module `l1tol2_if` can also receive events from Layer 1. It generates packets containing the address and the time stamp and forwards this packet to the transmission protocol. Answer packet from the ARQ control block are also transmitted towards the DNC interface for external control functionality.

It is possible to control all communication data between Layer 1, HICANN configuration and FCP over the JTAG test interface. This allows for detailed verification access to observe the functionality of all components and allows fast measurement stimuli insertion without having the whole communication system available. The following subsections present the communication modules on the HICANN chip in more detail. A detailed specification can also be found in [22].

[11] *Open Core Protocol Specification 2.2* [3]

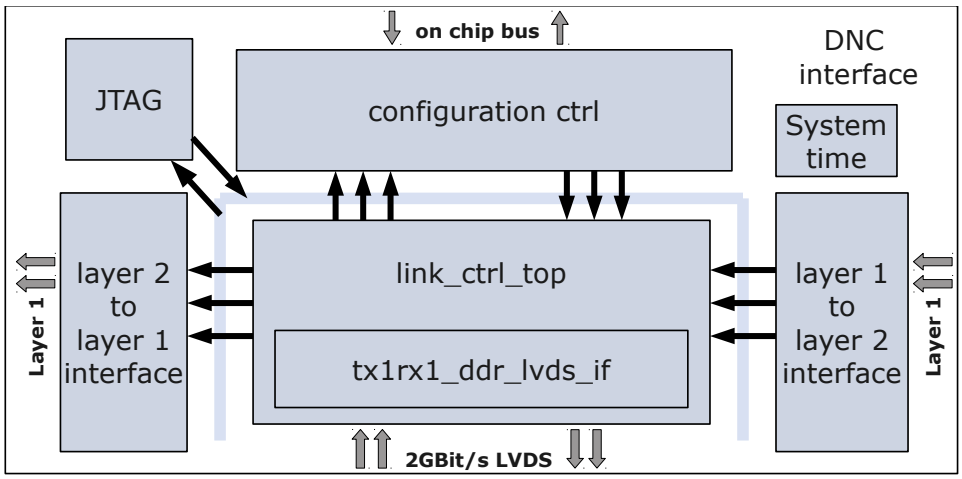


Figure 2.3.10: Block diagram of the layer 2 HICANN connection

Layer 2 packet communication - link_ctrl_top

The `link_ctrl_top` module contains all required components to establish the bidirectional 2GBit/s connection with the FCP, to hold the communication line in a ready state and to handle different types of data packets.

The communication is established using a master-slave control. The FPGA on the FCP is the master, that controls the process of initialization and the `dnc_if` of the HICANN is the slave, that waits for signals of the master. The second difference is the layout of the communication macro `tx1rx1_ddr_lvds_if`. It contains all required full custom components for the LVDS connection. It requires a special layout to be adapted to the special floorplan of the HICANN ASIC. The detailed description of the packet communication mechanisms can be found in chapter 2.6.

The Layout of the `hicann_tx1rx1_ddr_lvds_if` has to fulfil several constraints in order to fit into the physical design of the HICANN. Since Layer 1 connections of the HICANN were diagonally routed above the `hicann_tx1rx1_ddr_lvds_if` macro on top metal layer (Metal6) (see HICANN layout) and because of signal integrity reasons, the signal routing inside the macro should be limited to three metal layers. As an exception power/ground and quasi-statical nets (i.e. JTAG signals) can be routed on Metal4. The macro is placed with its pads heading to the inner core, thereby easing the connection to the post processing pads for the final design. The final layout of the macro is pictured in fig. 2.3.11.

Layer 1 access - l1tol2_if, l2tol1_if

The layer 1 to layer 2 (`l1tol2_if`) interface captures the 6 bit neuron identifier of eight available layer 1 buses. In parallel the current system time with a size of 15 bit is stored. Together with the number of the layer 1 bus, a digital event of 24 bit is created, that is forwarded to the digital wide range communication layer. To handle burst occurrence of neural events first-in-first-out (fifo) buffers are used at the output of each layer 1 connection. It is possible to transmit one event in a Layer 2 packet or two events in one packet. Depending

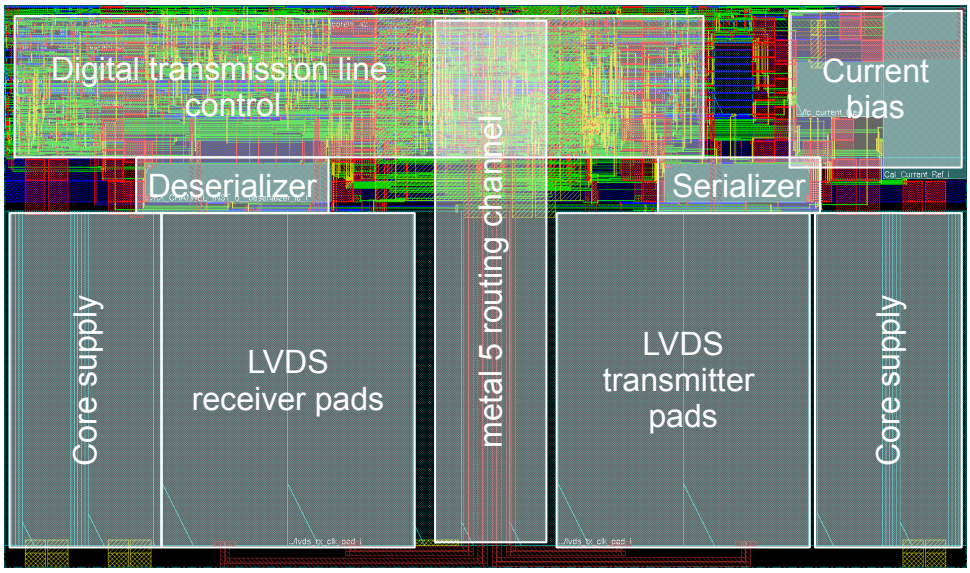


Figure 2.3.11: Layout view of the hicann_tx1rx1_ddr_lvds_if macro. The metal 5 routing channel can be seen in den middle.

on the pulse rate of the neurons, the required packet is generate automatically. A block diagram of the `l1tol2_if` interface can be found in fig. 2.3.12.

In the direction from Layer 2 packet communication towards the Layer 1 buses (`l2tol1_if`), buffer elements are required to delay the pulse events until their release time is reached. The HICANN offers two event storages in the current release, in future releases up to 16 buffers are available to be able to release neural events timely very close to each other. The lower 5 bit of the 15 bit event timestamps are compared with the current system time and released exactly at the required time. If two events needs to be released at the same time, one of them is delayed and released at the next possible time. The upper 10 bit of the time stamp were previously handled in the FPGA on the FCP, so that we have a reduced hardware complexity in the HICANN. The 5 bit of the time stamp equals 1.28 ms of biological time. The corresponding block diagram can be found in fig. 2.3.13 for the final version with 16 storage elements.

A detailed description of the Layer 1 interface can be found in [22].

All Layer1 events going that have to be transmitted to the PCS are sent through the Serial Parallel Layer 1 Interface (abbreviated `spl1_if`). This module communicates with the `l2tol1_if` and the `l1tol2_if` modules.

The `dnc_if` and the `spl1_if` have their own configuration address. Table 2.10.22 contains the possible settings.

Configuration packet handling

As described in the next subsection, the configuration of the HICANN is done using the ARQ protocol. The required data is received by the Layer 2 connection in a 64 bit configuration data vector. It is directly forwarded to the `dnc_if` and handled there as described above.

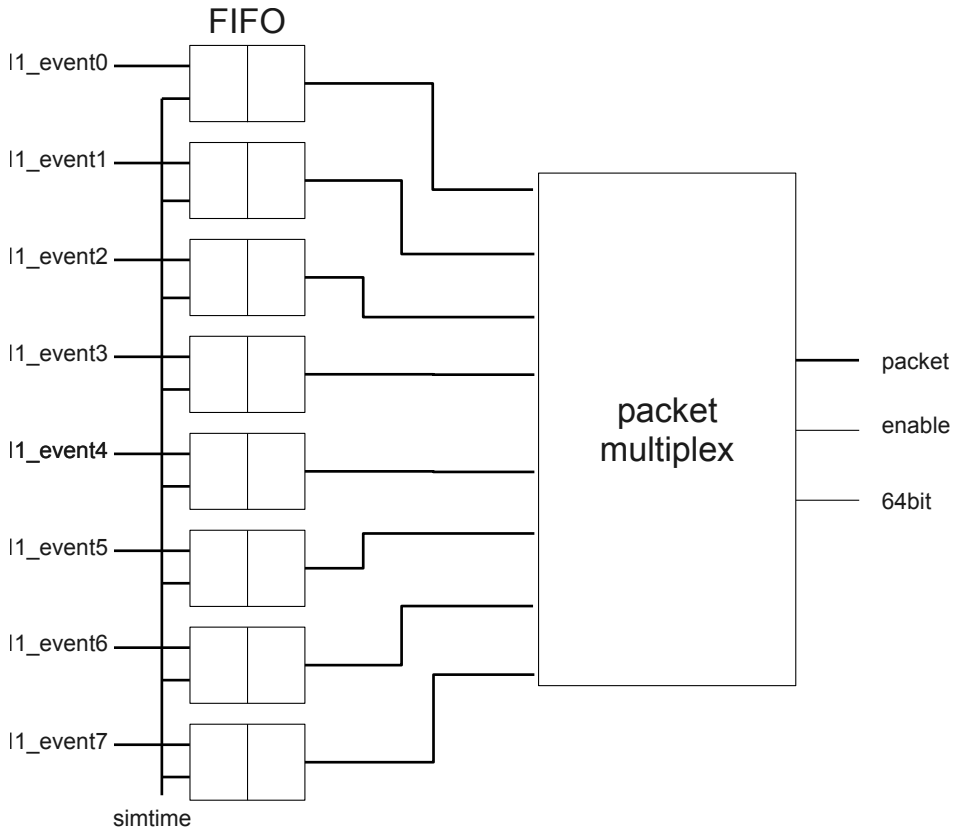


Figure 2.3.12: Block diagram of the l1tol2_if interface with fifo buffers and packet generator.

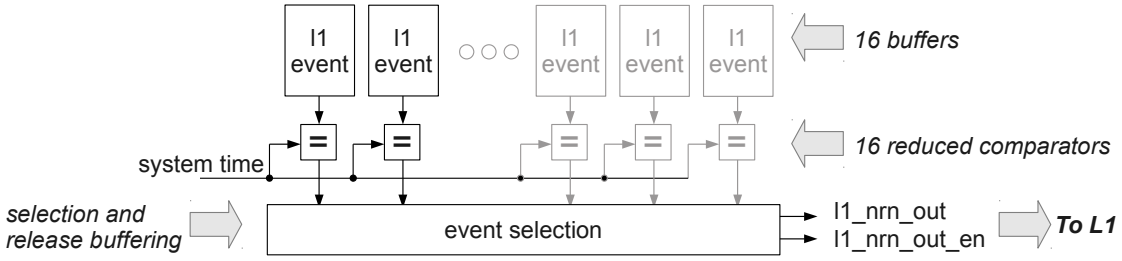


Figure 2.3.13: Block diagram of the l1tol2 interface with fifo buffers and packet generator.

The protocol also generates packets that are required at the host control. Therefore the dnc_if has a receiving bus fabric connection, whose data is transmitted as a configuration packet of the Layer 2 connection towards the FCP, where it is also forwarded to the connected host control. To handle burst transfers of configuration data buffers are used at the transmit and receive side of the on-chip bus fabric.

Bus fabric DNC interface

The bus fabric DNC interface consists of an RX- (DNC interface to bus fabric) and a TX-Port with the following configuration (see OCP Specification [3] page 64 for default values of the parameters not listed below):

```
data_width   = 64  // size of DNC configuration interface packet
addr         = 0   // no address
cmdaccept    = 1   // blocking flow control
write_enable = 1   // only write, the sender is always the master
read_enable  = 0
sdata        = 0   // no read
tags         = 0
threads      = 0
resp         = 0   // no response needed
sreset       = 0   // reset is not part of the ocp
mreset       = 0
```

The RX-port always accepts the maximum packet rate while the DNC interface as the slave of the TX-port uses SCmdAccept to block the master if the link is busy. As stated above, packets received through the RX-port might get discarded in the bus fabric due to FIFO full conditions.

2.3.4.3 Configuration Interface

The configuration interface implements the bus fabric connecting all digital modules with the host computer. The host communication is part of the data transported by the FCP. While the digital event communication is handled within the DNC interface the configuration data has to be distributed throughout the HICANN to all digital modules. This is done by a bus fabric based on the OCP. It has independent up- and downlinks, therefore two bus fabric links connect the DNC interface with the bus fabric. The FCP uses a packet-based transfer scheme with a fixed packet size of 64 bits for the configuration data. An internal CRC ensures that the data is unaltered. Due to the protocol overhead[12] the packet rate is 22 MHz at a link data rate of 2 GBit/s, resulting in a maximum data rate of 176 MByte/s per direction. If the network is working, the majority of the link traffic will be digital event data, substantially reducing the effective configuration data rate.

The host data is first sent via GBit-Ethernet to one of the FPGAs on the HICANN system board. The FPGA-based network is subsequently used to carry it to its destination HICANN.

[12] A packet including CRC data uses 80 bits plus some additional frame bits.

name	bit no.	size	description
Tag	63:62	2	tag-id
Seq Valid	61	1	sequence number valid
Seq	60:55	6	sequence number modulo 64
Ack	54:49	6	acknowledge number modulo 64
Application Layer Data	48:0	49	application layer specific
Frame	63:49	15	see frame definition
Write	48	1	defines write packet if one
Address	47:32	16	write address, address map defined by target module
Data	31:0	32	write data, content defined by target module
Frame	63:49	15	see frame definition
Write	48	1	no write packet if zero
Read	47	1	marks read packet if one
Data	46:0	47	read data, content defined by target module
Frame	63:49	15	see frame definition
Write	48	1	no write packet if zero
Read	47	1	no read packet if zero
Misc	46:0	47	reserved for future use

Table 2.3.5: configuration interface link layer (top) and application layer write, read, misc (bottom three)

Two things can happen on this way: packets may get lost or packets may be reordered. Therefor a data link layer is necessary to ensures a correct transmission. To keep the resource usage within the HICANN chips low the host handles most of the link layer complexity.

Configuration Data Link Layer Protocol

Since the individual functional units within the HICANN chip differ vastly in speed, multiple virtual connections can be established between the HICANN and the host. For every virtual connection there is a fixed connection tag assigned to it, that is placed at the beginning of the configuration interface link layer packet. Table 2.3.5 illustrates the different packet formats. The number of virtual connections implemented in HICANN is 2, one for the synapse memories and one for the rest. This allows for independent control accesses while transferring large amounts of synaptic weights. While all packets within one virtual connection are strictly in sequence, no order is imposed concerning packets from different virtual connections. The host sends all request packets with a certain tag with strictly increasing sequence numbers. After the DNC interface the packets are routed to as many independent link layer controllers. A simple fixed priority scheduler multiplexes the transmit data from the different link layer controllers into the TX-fifo of the DNC interface.

The link layer implements the ARQ[13] protocol with *selective repeat error recovery* as specified in [54][14]. The link layer uses a six bit sequence number. Acknowledge packets are merged within the normal packet stream of the opposite direction using a six bit field for the acknowledge number in each packet. If the packet contains only an acknowledge, the sequence number valid bit is false. This also implies that the application layer data is non-existent. The buffer size is 16 packets per direction and tag id, resulting in a total memory size of 512 Byte.

The advantage of this implementation is that the link never fails, it just gets slower if there are a lot of dropped packages. So no error handling is necessary in the application layer within HICANN. The pipeline depth between the input FIFO and the response FIFO of the application layer interface is known for each virtual connection. If the number of empty entries in the response FIFO is equal to the maximum number of requests in the pipeline, the input FIFO does not get popped any more. This has the advantage that a FIFO full condition of the response FIFO, which is usually caused by too much event traffic on the FCP link, translates itself eventually to a FIFO full condition on the input FIFO. This leads to a buffer full condition at the host, which can then wait with the next request unit the congestion is over or throttle down the request rate to match the effective link capacity.

2.3.4.4 Configuration Modules

- Layer 1 switch control
- Universal SRAM controller module
- Neuron builder control and DenMem configuration
- Analog output configuration
- Floating gate controllers
- Spl1 output merger
- Background event generators
- Repeater controllers and test pattern generators

Layer 1 switch control

Control of the Layer 1 switch transistors is performed by six independent control modules: crossbar left, crossbar right, synapse driver top left, top right, bottom left and bottom right (see section 2.3.2.1). For crossbar configuration, only the four lowest bits are valid, for synapse driver switch matrices 16 bits are valid. For the crossbars bit 0 is corresponding to the lowest vertical lane number, whereas for the synapse switch matrices, bit 0 is corresponding to the highest vertical lane number. The address corresponds directly to the vertical/horizontal line number as depicted in fig. 2.3.6. Configuration is stored in latches

[13] Automatic Repeat Request
[14] In the nomenclature of the referred report it uses full duplex mode with tiny packets.

that are located in the digital part. These latches are cleared to zero by the external reset, i.e. no connection is enabled after power-up reset.

SRAM control (sramctrl.sv)

The SRAM control module generates the timing for the full custom SRAM blocks located in the synapse drivers, the synapses, the neuron spl1 output circuits, the denmem circuits and the repeaters. To support configurable timing as well as an easy interface for additional module specific IO, the SRAM module uses an address field one bit wider than the addressable SRAM. The uppermost address bit selects between configuration address space (address MSB = 1) and SRAM address space (address MSB = 0). The configuration space contains the timing control registers as well as a user selectable number of static register ports. The output ports use storage registers located in the SRAM module while the input ports always reflect the state of the respective input lines of the SRAM module. The generic version of the SRAM controller is instantiated with the following parameters described in table 2.10.2a.

The timing control registers are organized as described in table 2.10.2b.

The content of IO data is defined by the module instantiating the SRAM controller.

Neuron builder control (neuronbuilder.sv)

The neuron builder control module consists of the SRAM controller for the full custom neuron output, neuronbuilder and denmem circuits located in the center of the ANNCORE macro block. The SRAM controller settings that are used in this module are listed in table 2.10.3.

It instantiates two output and one input registers, as described in table 2.10.4. The bit fields in the output registers are defined as in table 2.10.5. The controlled RAMs are used as described in table 2.10.6. The maximum address is 511. The addressing scheme for the neuron builder SRAM is also depicted in fig. 2.3.14.

See section 2.3.4.4 for a description of the meaning of the `nmem` and `nb` bits.

Denmem configuration

Additionally to the analog parameters, sourced by the floating gate array, several digital parameters are used to control a `denmem` and to generate complex neurons of several `denmem` circuits. These bits are set in the neuron builder control module (c.f. section 2.3.4.4).

The switch `bigcap` enables the larger neuron membrane capacitance for for the upper or lower block.

The slow and fast bits are used to switch the multiplication factors for current parameters. The upper three and lower three (0 to 2) bits of these fields belong to the upper and lower HICANN half, respectively.

Fast and slow bit 0 control the current mirror for the I_{radapt}parameter of a denmem. The next bits control $I_{gladapt}$ and I_{gl}, respectively.

Additionally to these global configuration bits, 8 SRAM bits can be programmed in each pair of `denmems`. Starting with address 3, these bits can be accessed with every fourth address of the neuronbuilder ram. Table 2.10.8 describes the function of the bits. The configuration of the input/output bits is described in table 2.10.8b. The neurons have separate current input and analog output lines.

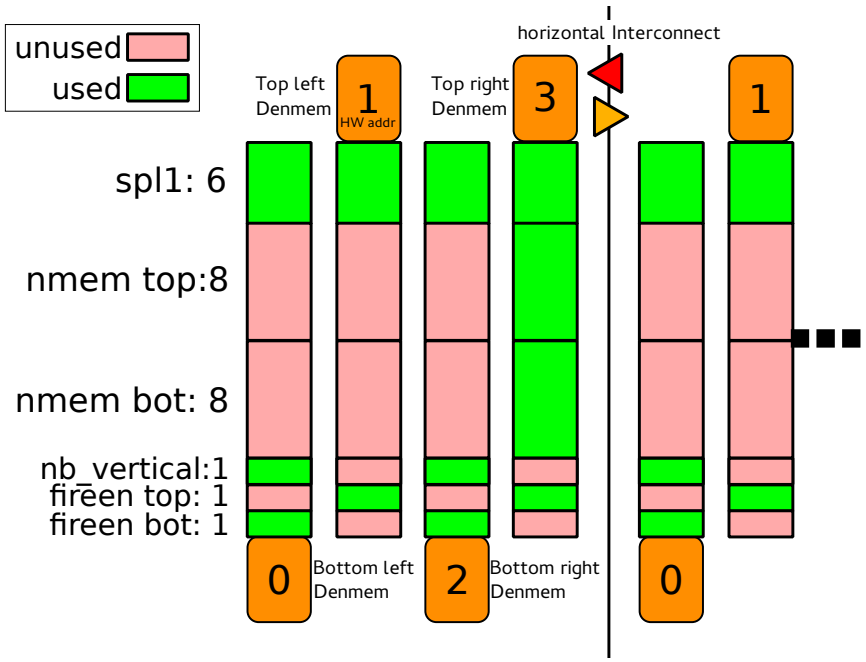


Figure 2.3.14: Overview of the denmem configuration

Even address numbers from the neuron builder RAM are used to access the neuronbuilder bits to connect bottom neurons to SPL1 and to interconnect bottom and top membrane, while odd addresses are used, when accessing bits connecting the upper neurons to SPL1.

Analog output configuration

The two analog outputs are controlled in neuron builder control (section 2.3.4.4). At reset, both amplifier outputs are set to high impedance and the calibration current from the DNC interface is connected to the output pad of the upper analog output.

The input of each output amplifier can be chosen by a ten-fold analog multiplexer, as described in table 2.10.9.

Floating gate control(unit::fgateContol)

Each floating gate array is controlled by an individual instance of fgateControl, being responsible for parameter programming, analog parameter readout of floating gate cells and current stimulation of neurons.

An fgateControl consists of three parts, an interface to the on-chip bus fabric, a programming state machine and a dual port latch memory with two banks to enable memory update while the floating gates are programmed.

Numbering of fgateControl instances is as follows: top left=0, top right=1, bottom left=2 and bottom right=3.

Programming state machine The fgateControl programming state machine can be controlled by six different commands (table 2.10.10). The command `read` sets the floating gate in read out mode and connects a cell to the analog output. This analog value can be read until another command is executed. After reset, the state machine is ready for calibration - it is set to readout mode, but no parameter is connected to the output.

`writeUp` and `writeDown` do the actual cell programming for a floating gate line. At first, the values of the individual cells are compared to the values in the memory and only the cells with values below respectively above the desired value will be programmed. In the second step, the cells are programmed, followed by another comparison step. The last two steps are repeated until all cells reached their values or the maximum number of cycles has been reached. While the controller is active, a busy flag is set. If not all cells could be written to the proper value, an error flag is set and the column address of the first cell which did not reach its value can be read out (see table 2.10.15). To get the next wrong programed cell, the instruction `getNextFalse` is used. Once no further wrong programmed cell is available, the error flag will go down.

The instructions `stimulateNeurons` and `stimulateNeuronsContinuous` use the programming memory and the DAC in the floating gate array to stimulate a neuron by a changing current. The first one replays the RAM values only once, while the second one starts at the beginning when the last ram address is reached.

Several parameters of the programming state machine can be set to optimize programming (table 2.10.11). The parameter `maxcycle` is the maximum number of cycles allowed in a programming process. `currentwritetime` and `voltagewritetime` are the respective lengths of a writing pulse. Depending on the aimed value, these parameters are very critical for writing precision. When the values are controlled during a write process, the controller waits a certain number of intern clock cycles after a cell has been accessed as the readout line needs to be loaded. A small value will shrink precision, a large value will cause long writing times.

As the effect of a writing pulse for voltage cells gets smaller the higher the actual voltage on a floating gate is, longer writing pulses are a good choice for higher values. To allow longer writing pulses for higher voltages, the write times can be doubled during a write process. The parameter `acceleratorstep` gives the number of programming cycles after which the write time is doubled. This doubling of the actual write time happens every `acceleratorstep` cycles. The counter register that holds the write time is therefore 9 bits wide although it can initially only be set to a 6 bit wide value (that from table 2.10.11). The doubling only stops if the 8th bit of the write time register (actualWriteTime[7]) is set, i.e. the actually applied write time cannot become larger than 254 cycles.

As the floating gate programming is in the order of magnitude of milliseconds, the slow HICANN clock is multiplied by `fg_pulselength`+1 to be able to use smaller counters. Setting `fg_pulselengh` can also be used to setup the speed of the neuron stimulation. If set too maximum, the maximum period of the neuron stimulus is 32 ms at normal HICANN clock speed.

Given a clock period of 40 ns (i.e. 100/4 MHz) and maximum values for all parameters including maxcycle we get a write time per block of 34.98 seconds. For all parameters set to maximum value and acceleratorstep set to 1 we get 35.28 seconds. These results have been

calculated with the python code in the following listing. With the currently used default settings in HALbe, the programming time should be 21.96 seconds. The used parameters (in the order as used in the following listing) are (15,9,255,63,9).

```
def progtime_per_row(writetime,pulselength,maxcycles,readtime,accstep):
        total_clock_cycles = 0 # cycles
        last_double = 0
        for cnt in range(maxcycles+1):
                if cnt == (last_double+accstep) and writetime < 128:
                        last_double = cnt
                        writetime=writetime*2

                total_clock_cycles += writetime*(pulselength+1)
                total_total_cycles += (readtime+4.6)*(pulselength+1)*129

        return total_clock_cycles
```

Reading out the error flags after programming can take as much as 615 ms (as measured on Wafer 2) per row (i.e. 14.76 seconds in total). This number was not included in the previous calculations. The number was measured when programming with minimal programming parameters and with every column in every row showing an error flag.

Programming memory The latch based programming memory has two banks of 65 20 bit words and two ports for interfacing - one for the control interface which can be directly accessed via the on-chip bus and one for the programming state machine. The port of the programming interface is read only and has only ten bits data width so every word is split at this port. The memory has been designed this way to allow a better bus utilization.

Control interface The control interface is the top level unit of `fgateControl` and gives access to the state machine and ram. Access to Random Access Memory (RAM) is directly gained if bus address bit 8 is 0.

The state machine can be controlled if bus address bit 8 is set. The `biasRegister` (table 2.10.12) can be accessed if address bit 3 is set. Here `groundvm` shorts V_m to ground and `calib` connects the current measurement resistor in the array to the calibration current source.

If bit 1 is active, the so called `operationRegister` (table 2.10.13) can be writteni or read.

If address bit 0 is active, the `addressInstructionRegister` (table 2.10.14) is written and a request signal is send, switching the programming state machine into the instruction decode state.

The register `slaveAnswerData` (table 2.10.15) can be accessed if 2 is set. This register is the only digital feedback from the programming state machine accessible. `error` is set if a value was not reached during programming. The column address of this value can be found in `slaveAnswer` than. When `busy` is set, the programming state machine is active and not ready for other instructions. When using fgateControl `busy` has to be polled to see when the controller is finished.

Calibration If the calib option(see table 2.10.11) is set, the calibration current from the DNC interface is connected to the current measurement resistor in floating gate array and the voltage drop-out can be measured at the analog output of the floating gate array.

SpL1 output merger (neuronctrl.sv)

The SpL1 output merger connects the eight SpL1 output buses from the ANNCORE with the DNC interface and the SpL1 driving repeaters. Its general structure is shown in fig. 2.3.15. Each merger module in the tree is depicted by a multiplexer symbol and has two input and one output registers. An event appearing at an input is transmitted to the output register. If the output register is full, the event is stored in the according input register. Sending of a non-empty input register always precedes an input event on the other channel. A chain of merger modules acts like a fifo since the output register is only transmitted to the next merger module in the chain if the input of the merger it connects to can accept an event. If a merger at the top of the tree can not accept an input event the event is dropped. Each merger module supports the configuration bits that are listed in table 2.10.16.

The mergers have two modes of operation. If their `enable` bit is set to 1, they merge both input streams into successive cycles of the output stream. If their `enable` bit is set to 0, they statically output only one input stream as selected by their `select` bit. If the `slow` bit is set to 1, the merger outputs events at most every second clock cycle. After an event has been sent in this mode the merger waits for one clock cycle before transmitting a possible next event. This has been implemented to guarantee that the sending repeaters only emit one event every second cycle.

By programming the mergers to be either static or mergers, two, four, six or eight neuron SpL1 channels can be merged into one SpL1 output data stream. The possible combinations and the utilized outputs can be deduced from fig. 2.3.15. The eight outputs from the different tree levels pass through a last row of mergers that merge the input from the DNC interface into the event stream coming from the neurons or the background event generators. For testing purposes, eight additional tri-state buffer groups at the outputs of this last row of mergers allow for the re-direction of the event stream to a neighbouring output register. Table 2.10.17 lists all configuration addresses for the SpL1 output merger module. The data-width is 16 bits.

The top of the tree is formed by the background event mergers. They combine the neuron SpL1 output with the output from eight background event generators. The neuron number of a background event generator can be individually configured if the background event generator is used in non-random mode.

The data from the neuron SpL1 outputs can be sampled either at the rising (phase=0) or falling (phase=1) edge of the 250MHz sysclk.

All registers are initialized to zero. After reset, the dnc mergers (lowest row) are configured as static muxes (enablednc=0). Their input can be selected by the `dncen` bits, which act as the select input for the mergers (select dnc=1 switches to dnc input). Making use of the merger feature of the dnc mergers is similar to configuring the merger tree: set enable=1 for the respective dnc merger. If the SpL1 repeater output is going to be used, the slow-bit must be set to one as well. The slow bits can be set to zero everywhere else in the merger hierarchy.

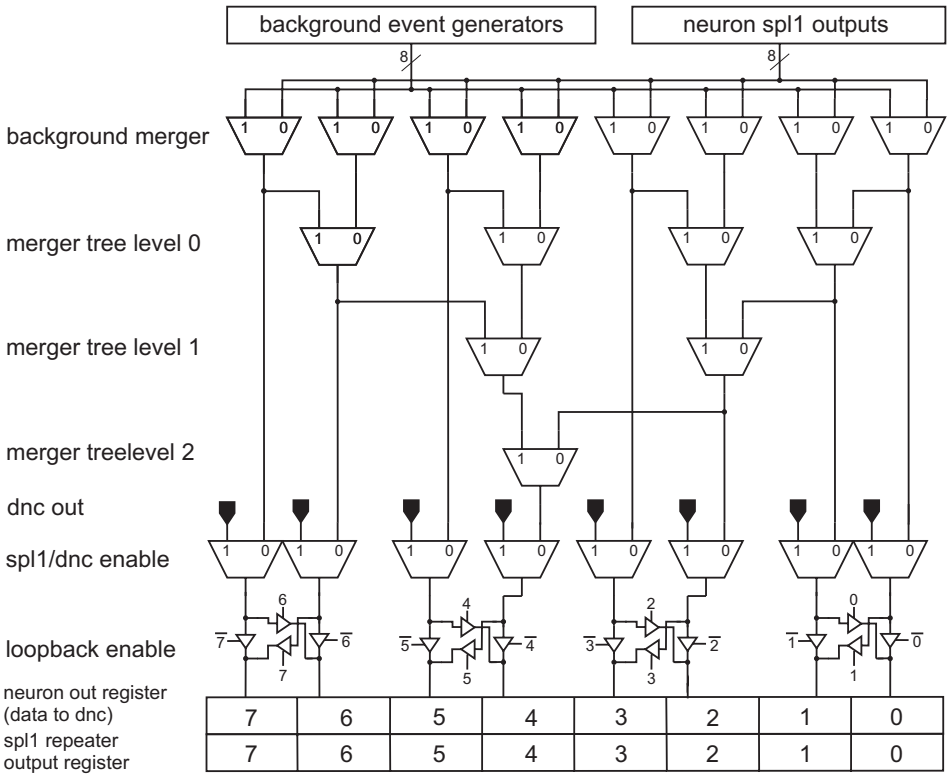


Figure 2.3.15: Block diagram of the spl1-merger tree.

To use the individual neuron number feature of the background generators, the neuron number has to be written into the appropriate register before enabling the background generator. Additionally, the background event generator has to be set to non-random periodic mode.

Background Event Generators

There are 8 background event generators, which can produce regular or pseudo-random spiketrains. Each background generator has the configuration parameters defined in table 2.10.18. The configuration is written to the registers in the `neuronctrl` module that also holds the configuration for the merger tree. The location of the background event generator configuration in this module is listed in table 2.10.17.

In random mode the spikes are generated with a 16-bit linear feedback shift register(LFSR) with tap configuration: 16, 14, 13, 11, which implements a maximum length polynomial for 16 bit. [15] At each clock cycle the bits are shifted towards the MSB, the LSB is set as following:

$$LSFR[0] = LSFR[15] \oplus LSFR[13] \oplus LSFR[12] \oplus LSFR[10]$$

with $\oplus$ meaning an XOR, and the right hand side of the equation holding the values of the previous cycle. The value in the LFSR register is compared to the PERIOD value. If LFSR is bigger than PERIOD and in the cycle before LFSR was not bigger than PERIOD, then a spike is generated. Hence, the minimal inter-spike-interval is 2 clock cycles. The register cycles through the maximum number of 65535 states, which means that the spike sequence is repeated every 3.28 s assuming 5ns sysclk and a speedup factor of 10^4.

In non-random mode, spikes are generated every period + 1 cycles.

Repeater control

Each instance of the repeater control module is in charge of one of the six repeater blocks. It has the following functions:

- Control the static configuration signals of a repeater block
- SRAM controller for the SRAM built into the repeaters
- two light-weight test pattern generators (TDO) and two receivers (TDI) to access the parallel debug ports of a repeater block

The SRAM controller settings are as follows:[16]

IO address map of the reapeter block's SRAM controller can be found in table 2.10.21.

Table 2.10.20 shows the function of the repeater SRAM bits.

Recen and touten are set to zero with the HICANN reset. In the sending repeater, also dir is cleared. If touten and tinen are activated simultaneously, no data is visible at the tdo-data

[15] The LFSR in the HICANN is exactly the same as Fibonacci LFSR example on wikipedia: `http://en.wikipedia.org/wiki/Linear_feedback_shift_register#Fibonacci_LFSRs`

[16] Due to an error in the HICANN V1 the address bits are permuted: bit 0 → bit 6, bit 1 → bit 5 etc. There is also a wrong comment in the repeaterctrl.sv source code: `rep_adrw` is 7 not 6! All addresses in this paragraph assume a correct bit ordering, the software will swap them transparently.

outputs, but the DLL-output can be still monitored with the tclko-pin. For the TDI pattern receiver this looks like receiving patterns with neuron number zero (see below).

TDO operation After setting the corresponding tinen bits in the receivers, the TDO pattern generator allows sending of up to three SpL1 events to the receiver parallel test data in port. If the receiver is configured for serial ouput, it transmits these events on the corresponding L1 line. The pattern generator is started by setting start to one. After waiting the number of sysclk cycles specified in the entry 0 time field, it generates a SpL1 event with the neuron number of entry 0 at the receiver test port. Afterwards it proceeds with entry 1 and 2. If start is still one after entry 2 has been transmitted, the pattern generator repeats the sequence. The repeater locks onto sysclk if tinen is active.

TDI operation The test data receiver gets its input from the repeater with activated touten. Only two repeaters per block can have touten enabled: one with odd address and one with even address! The moment its start bit is set, an internal counter starts counting sysclk cycles starting by one. Each time an SpL1 event is present at the test data out port of the repeater block, it is stored in a test_data_in entry together with the current counter value in gray code format. After entry 3 has been written no further data is stored and the full bit in the config register is set to one. If the 10 bit counter wraps around to zero counting is stopped to mark the overflow condition. Setting start to zero resets the full flag.

2.3.4.5 Digital synapse control

Digital Synapse Control (DSC) is the digital controller for the analog synapse array. Two instances of this block are implemented on one HICANN: one for each synapse array block. DSC is used for three tasks:

1) Read and write synapse weights and decoder addresses.
2) Read and write synapse driver configuration memory.
3) Perform Spike Timing Dependent Plasticity (STDP) during network operation.

For tasks 1 and 3, the basic mode of operation is for the user to write to configuration, control, and data registers only. A state machine then performs the actual access to the synapse array in response to certain command codes given in one of the registers. Task 2 is implemented by passing the request on the internal bus to an instance of the sramCtrl module (Section 2.3.4.4).

The DSC was designed as part of [29]. A more detailed description of the implementation can be found there.

Pipelined connectivity

The module is connected on Tag 1 of the internal bus using a pipelined interconnect. A request - either read or write - can be made in every clock cycle of the bus. It will then take 6 cycles to reach one of the DSC blocks, where it is processed within one cycle. Then, the response is returned again with 6 cycle delay.

STDP

For the realization of STDP, DSC has to evaluate the analog correlation measurement performed in the synapse circuit itself (Section 2.3.3.2). The synapse circuit maintains two analog accumulation voltages a_+, a_- reflecting the timing and number of recent spike pairs at the synapse. Here, a_+ reflects the accumulation of seemingly causal pairs, where pre-synaptic firing precedes post-synaptic action potentials (pre-before-post). Inversely, a_- represents seemingly anti-causal pairs (post-before-pre).

For the automatic weight update, these voltages are converted by an evaluation circuit to a single correlation bit. This is done two times with different configurations p_0 and p_1 of the evaluation circuit to produce two bits C^0 and C^1 as input to the weight computation.

Evaluation The following description is given in [29]:

> For the evaluation of the local accumulation capacitors, the evaluate block is used. The syn_encrb signal triggers readout of the capacitors to a temporary storage location in the evaluation unit. The sca/scab pair of control signals for one and scc/sccb for the other capacitor control whether both, one, or none of them are stored. The 4 bit pattern control the evaluation operation. With csen a digital bit is generated from the analog evaluation result and presented on the corrin bus, which is multiplexed as for the synapse bitlines. A low-active reset bus corresetb is split to the selected columns to reset both accumulation capacitors.
>
> The evaluation block in Figure 2.3.16 implements a generic evaluation function $E^H([\ldots])$. It is configured by an evaluation pattern p with bits $p = (e_{aa}, e_{ac}, e_{ca}, e_{cc})$ that control the evaluation operation. Two analog parameter voltages $V_{\text{th}}, V_{\text{tl}}$ are provided from the global parameter storage. The accumulation trace $a(t)$ $[\ldots]$ is represented using the dual capacitor value with a positive "causal" capacitor with value $a_+(t)$ and a negative "acausal" one with value $a_-(t)$. The combined trace is then given by $a(t) = a_+(t) - a_-(t)$. The temporal storage locations V_c, V_a in the evaluation block are set depending on whether scc or sca is asserted:
>
> $$V_c \leftarrow a_+(t) \quad \text{if scc} = 1 \wedge \text{sccb} = 0 \tag{2.3.1}$$
> $$V_a \leftarrow a_-(t) \quad \text{if sca} = 1 \wedge \text{scab} = 0 \tag{2.3.2}$$
>
> If neither scc nor sca is set, V_c and V_a remain unchanged. The evaluation function is described by
>
> $$E^H(V_{tl}, V_{th}, p, V_c, V_a) = \begin{cases} 1 & \text{if } \frac{V_{tl}+e_{ac}V_c+e_{ca}V_a}{1+e_{ac}+e_{ca}} > \frac{V_{th}+e_{cc}V_c+e_{aa}V_a}{1+e_{cc}+e_{aa}} \\ 0 & \text{else.} \end{cases} \tag{2.3.3}$$

Weight computation With the two result bits

$$C^0 = E^H(V_{\text{tl}}, V_{\text{th}}, p_0, V_c, V_a) \tag{2.3.4}$$
$$C^1 = E^H(V_{\text{tl}}, V_{\text{th}}, p_1, V_c, V_a), \tag{2.3.5}$$

and the current weight of the synapse w, a new weight w' is computed using a look-up table $L_w^{C^0C^1}$. This look-up table provides the 4 bit value of w' for every possible w and combinations of C^0 and C^1, where at least one is set:

$$w' = L_w^{C^0C^1} \tag{2.3.6}$$

Iteration The automatic weight update controller will perform the following algorithm, when started:

```
 1: repeat
 2:     r ← CREG.adr
 3:     s ← 0
 4:     for r ≤ CREG.lastadr do
 5:         for s < 8 do
 6:             SYNOUT ← ReadWeights(r, s)
 7:             SYNCORR ← ReadCorrelation(r, s)
 8:             SYNIN ← LookupWeights(SYNIN, SYNCORR)
 9:             WriteWeights(r, s, SYNIN)
10:             ResetCorrelation(r, s, SYNCORR)
11:             s ← s + 1
12:         end for
13:         r ← r + 1
14:     end for
15: until CREG.continuous = 0
```

The functions ReadWeights(), ReadCorrelation(), WriteWeights(), and ResetCorrelation() are symbolic representations for the operations described in Section 2.10.4.10. LookupWeights() describes the weight computation mechanism described in Section 2.3.4.5. The field `CREG.continuous` (Register 2.10.4) controls if the iteration is repeated or not. See Section 2.10.4.10 for all register definitions.

Implementation

Figure 2.3.16 schematically shows all signals of the synapse array analog block that are controlled by DSC.

Figure 2.3.17 shows a block diagram of the internal organization of DSC.

Analog interface timings *This section is taken from [29]:*

Reading weight or decoder address Figure 2.3.18 shows a timing diagram for a read access to a synapse weight. The timing for reading decoder bits is identical, but instead of syn_ensynb the syn_endecb signal is used. Every constraint that applies to syn_ensynb mentioned below also applies to syn_endecb. Synapses are addressed in one 128 bit *column set*. The column set is selected by presenting an address on the syn_a and syn_ab address bus, side selection on syn_en, and an enable pattern on en. A typical enable pattern would be en = 0*x*80808080

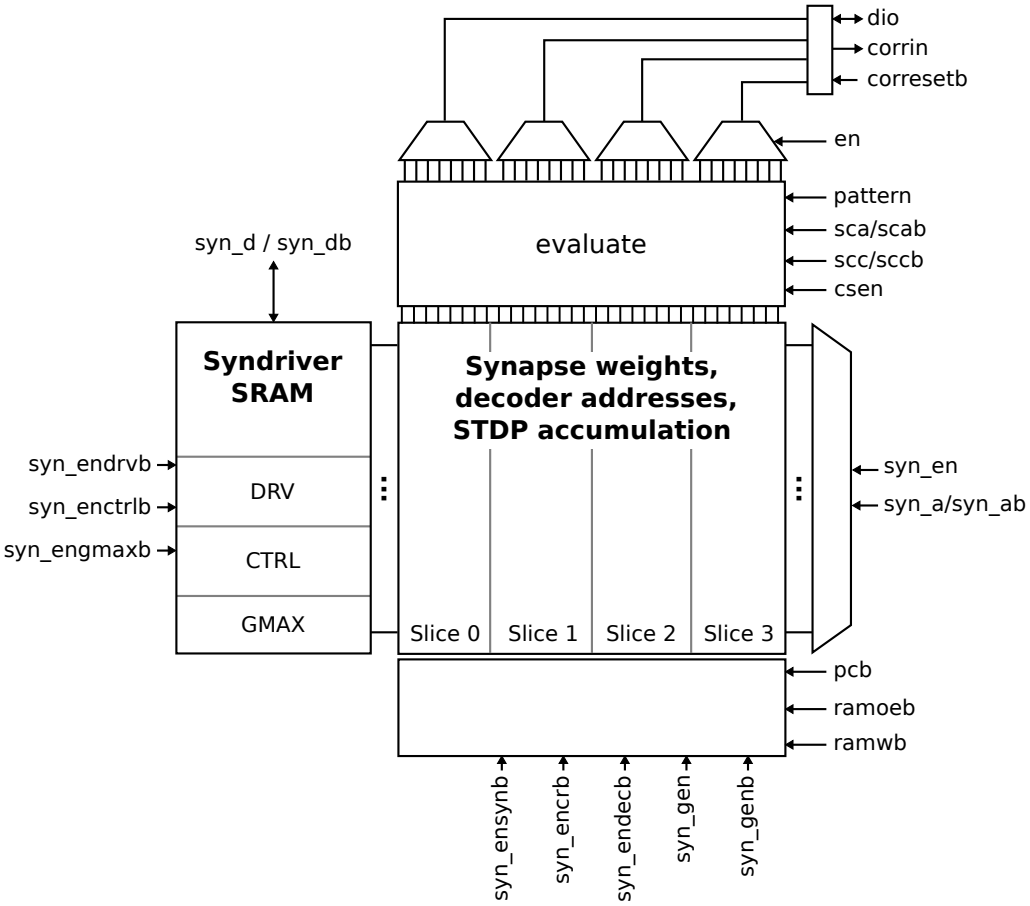


Figure 2.3.16: Overview of the synapse array organization and the signals controlled by the DSC. (Figure taken from [29] with permission.)

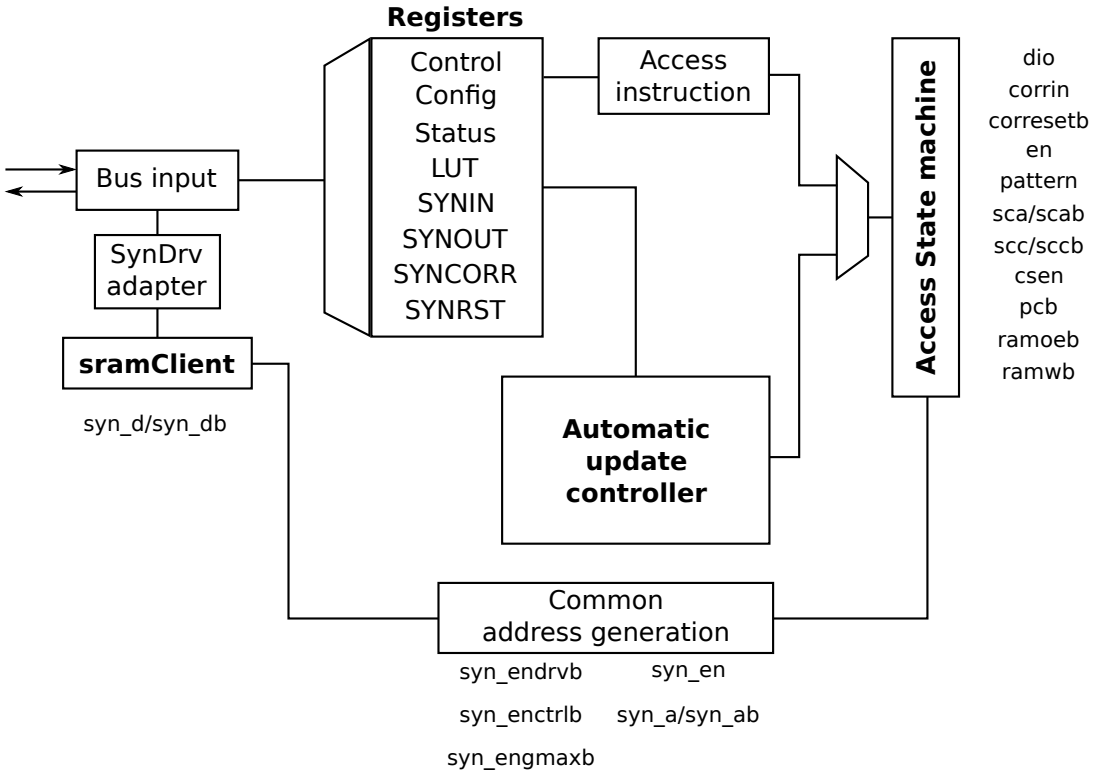


Figure 2.3.17: Overview of the implementation of the DSC module showing the main components and registers. (Figure taken from [29] with permission.)

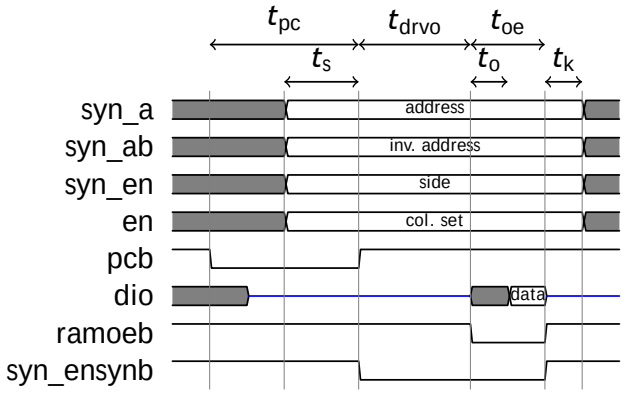


Figure 2.3.18: Timing diagram of a synapse array read operation. Signals not shown are at their inactive level. (Figure taken from [29] with permission.)

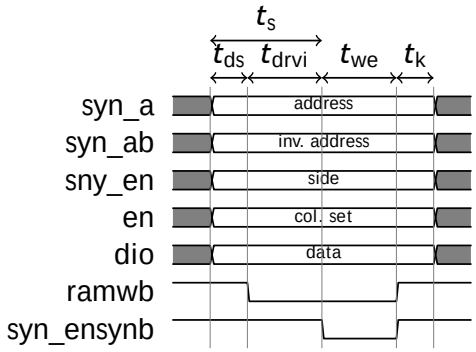


Figure 2.3.19: Timing diagram of a synapse array write operation. Signals not shown are at their inactive level. (Figure taken from [29] with permission.)

to select the first 32 bit from every slice. Each byte of en configures one multiplexer and only one bit per byte may be set.

When syn_ensynb is active, the wordline for synapse weights of the currently selected row is activated. Therefore, the address decoder must have settled before syn_ensynb is active and the address lines must stay stable while it is active. This safety margin between syn_a/syn_ab and syn_ensynb is described by the settling time t_s and the keep time t_k. The Static Random Access Memory (SRAM) bitlines are pre-charged for a time t_{pc} by activating pcb. Afterwards, the wordline is activated for a time t_{drvo} to drive the stored bit value onto the bitlines. Activating ramoeb for time t_{oe} enables the dio output driver to present the result to user logic. The output is stable after time t_o.

Writing weight or decoder address Figure 2.3.19 shows timing of a write access to a synapse weight. The same timing is used for decoder address writing by exchanging syn_ensynb with syn_endecb. Again, the address decoder settling and keep times t_s and t_k have to be satisfied. Additionally, after presenting the write data on dio and selecting the demultiplexer configuration with en, the user must wait for a settling time t_{ds} until the demultiplexer has settled. Then, the bitlines are driven for a time t_{drvi} by activating ramwb. When they have reached a stable state the wordline is activated for the selected row by activating syn_ensynb for duration t_{we}.

Accumulator readout and evaluation The timing diagram for a readout and evaluation sequence of the local accumulators in the synapse is shown in Figure 2.3.20. Settling and keep times t_s, t_k have to be satisfied between syn_a/syn_ab and syn_encrb. While syn_encrb is active, the accumulation circuit drives the readout lines for time t_{cro}. The temporal storage capacitors V_c and V_a are set by activating scc and sca, respectively[17] for time t_{sc}. Asserting the 4 bit evaluation pattern p triggers the evaluation operation as described in Section 2.3.4.5. The pattern may not overlap with activation of sca and scc, which is accounted for by the waiting time t_{scw}. The pattern is kept for time t_e. When the analog

[17] Not shown in the figure: sccb and scab are the inverted versions of scc and sca

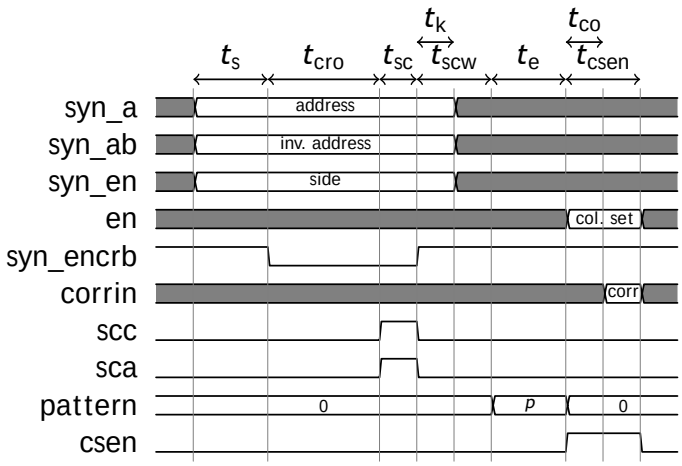


Figure 2.3.20: Timing diagram of readout and evaluation of the local accumulation capacitors. Signals not shown are at their inactive level. (Figure taken from [29] with permission.)

comparison is finished, activation of csen for a time t_{csen} generates a digital bit from the result and drives it to the corrin port as selected by the multiplexer configuration en. The output is stable after time t_{co}.

Accumulator reset The accumulator reset clears the local capacitors to zero. The corresetb bus controls which of the synapses are reset in a column set. The reset at the accumulation circuit is enabled by the weight word line. Therefore, syn_ensynb must be activated for a reset. This implicates, that accumulation can only be reset during a synapse weight read or write sequence. Figure 2.3.21 shows timing for reset during a write access. This is the most likely case for the STDP application: After reading weights and evaluating accumulation, new weights are written and the accumulators reset. Two additional timing parameters are relevant: the settling time of the corresetb multiplexer t_{cs} and the time needed for reset by the cell t_{cri}. In a combined write/reset cycle, syn_ensynb needs to be pulled down for the larger of the times t_{we} and t_{cri}.

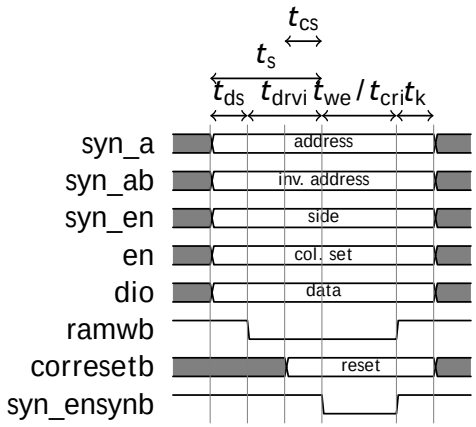


Figure 2.3.21: Timing diagram of reset to zero of the local accumulation capacitors. Signals not shown are at their inactive level. (Figure taken from [29] with permission.)

2.4 Wafer-Scale Integration

HICANN is designed in a way that the horizontal and vertical L1 busses can be driven to adjacent HICANNs by the L1 repeater cells that are located at the chip edges (see section 2.3.2). As a consequence, L1 connectivity over several chips can be obtained by placing chips next to each other on the silicon wafer. This edge-connectivity cannot be extended over the whole wafer because the lithography steps that are carried out during manufacturing introduce a maximum area of approx. 2 cm × 2 cm that can be exposed at a time. This area is called a reticle and is stepped repeatedly over the whole wafer with a constant pitch. Eight edge-connected HICANNs in two rows of four chips are placed within one reticle. Reticles cannot be edge-connected because the manufacturer reserves gaps between reticles for placing process monitoring structures. In order to bridge these gaps, i.e. to establish full-wafer connectivity, a post-processing method has been developed in cooperation with Fraunhofer IZM [39]. Post-processing also forms the mating pad structures for connection to the MainPCB via elastomeric stripe connectors (see section 2.5.3.4). It will be described in section 2.4.1. The design of one reticle is described in section 2.4.2. An introduction to the wafer-scale integration idea is given in section 2.3.1 and figure 2.3.2.

2.4.1 Post-Processing Procedure

The post-processing procedure has been developed within the FACETS and the BrainScaleS projects. Some documentation on the development can be found in the specification document of the 1st version wafer-scale hardware [35]. Only the actually used procedure will be described in the following. It is a slight modification of method B described in [35].

Figure 2.4.1 shows a cross-section of the post-processing. Three conducting routing layers are available: First, the fine-pitch routing that is used for inter-reticle connections only. Second, the intermediate routing which is being used to re-distribute signals. Both fine-pitch and intermediate routing are fabricated on top of layer Benzo Cyclo Butene (BCB) (2) in different process steps due to their different feature sizes (see below). The third routing layer is the top layer which is exclusively used to form the large contact pads that are connecting to the stripe connectors, which in turn connect to the MainPCB. Cu-studs form the contacts between the top metal layer of the silicon wafer and both fine-pitch and intermediate routing. The via contacts from intermediate to top layer routing need to be drawn but do not require a separate processing step (see below). The stripe connector pads have deepenings above these vias to the subjacent intermediate routing. These vias should therefore only be placed at the edges of the stripe connector pads in order to provide a flat

support for the Elastomeric Stripe Connectors (ElCos) within the relevant pad area (please refer to section 2.4.2 for details).

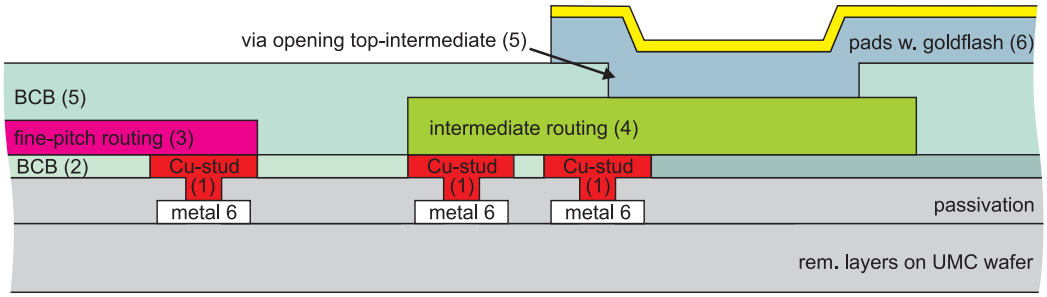


Figure 2.4.1: Schematic cross-section of post-processing layers. Note that the openings for vias between intermediate routing and contact pads are generated during step (5). The via material itself is deposited together with the large pads in step (6). As a consequence, the large pads have deepenings above the vias.

The following process steps are carried out during post-processing; steps with production layers are marked with numbers that correspond to the layer numbers in figure 2.4.1.

- sputtering of plating base TiW/Cu
- lithography with photoresist for electro-plating mask of Cu-studs

(1) electro-plating of Cu-studs
- strip photoresist
- wet chemical etching of plating base

(2) deposition of a planarizing polymer layer (BCB)
- lithography with photoresist for electro-plating mask of via openings
- dry etching of via openings
- strip photoresist
- sputtering of plating base TiW/Cu
- lithography with 5 µm photoresist for electro-plating mask of fine pitch inter-reticle connections

(3) electro-plating of fine pitch wires (thickness 2 . . . 5 µm)
- strip photoresist
- lithography with 10 µm photoresist for electro-plating mask of intermediate on-reticle routing

(4) electro-plating of intermediate routing wires (thickness 3 . . . 4 µm)
- strip photoresist
- wet chemical etching of plating base

(5) deposition and structuring of a planarizing polymer layer (BCB)
- sputtering of plating base TiW/Cu
- lithography with photoresist for electro-plating mask of large contact pads

(6) electro-plating of Ni with goldflash of top layer (stripe connector pads, thickness 4 . . . 6 µm)
- strip photoresist
- wet chemical etching of plating base

The planarizing BCB layers equalize the comparably bumpy surface of the wafer and provide a plain surface for the subsequently manufactured conducting structures. An illustrated photograph of a post-processed wafer, as it will be used for NM-PM1 is shown in figure 2.4.2.

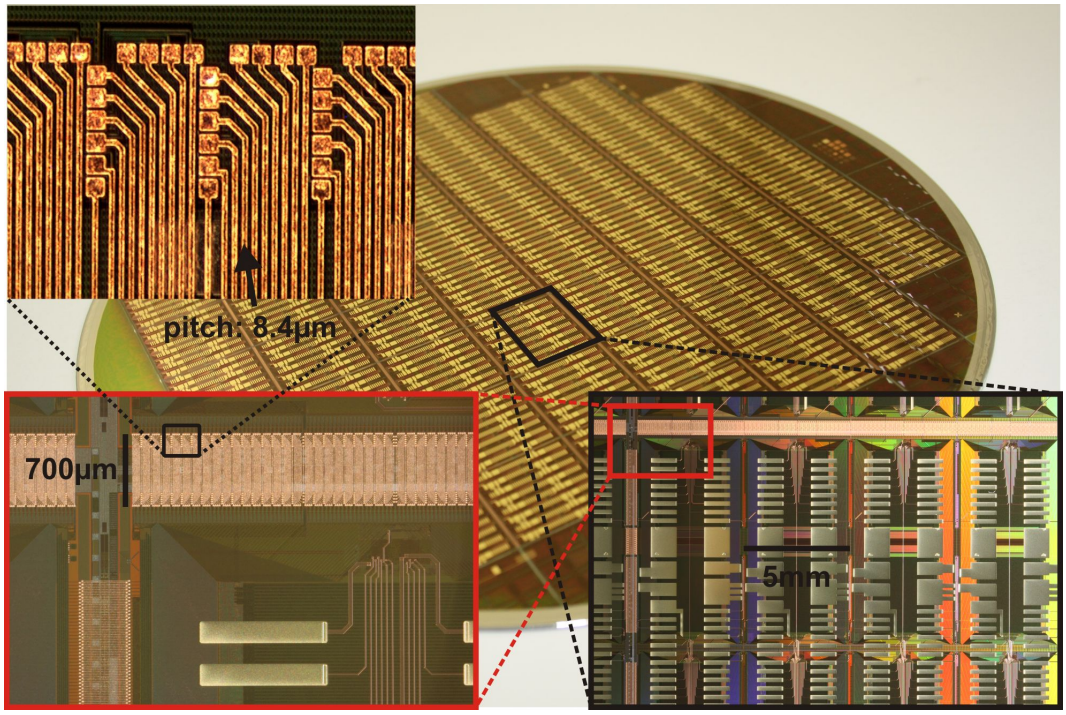


Figure 2.4.2: Photograph of the post-processing structures on an NM-PM1 wafer at various zoom levels.

2.4.1.1 Post-Processing Design Rules

All required post-processing design rules are collected in table 2.4.1 and illustrated in figure 2.4.3. Only one size of passivation opening is allowed in order to guarantee a homogeneous quality of the wafer contacts on the whole surface. These passivation openings must be overlapped by the Cu-studs by 5 µm on each side, while United Microelectronics Corporation (UMC) standard design rules apply for top metal. Arrays of these contacts must be formed to allow for large in/egress currents on power signals. Minimum fine-pitch spacing is allowed between Cu-studs in arrays.

Fine pitch routing is only used for inter-reticle connectivity and cannot be contacted by any other routing. Intermediate routing is used for connecting Cu-studs to top-layer contact pads, with corresponding overlap constraints on Cu-studs and the vias to the contact pads, respectively. Cu-studs, intermediate routing and top layer pads can be stacked as can be seen in figure 2.4.1. This can for example be used to place large contact arrays directly underneath top layer pads. In case minimum spacing is not required, lines should always be equally spaced as far apart as possible.

When put next to each other, Cu-studs have a larger pitch than the fine-pitch routing. Since the minumum pitch of the fine-pitch routing is required over the full width of the top and bottom edge of a reticle, Cu-studs are arranged in an L-shaped manner that is illustrated in figure 2.4.3a). These structures are used in all situations that require minimum pitch fine-pitch routing.

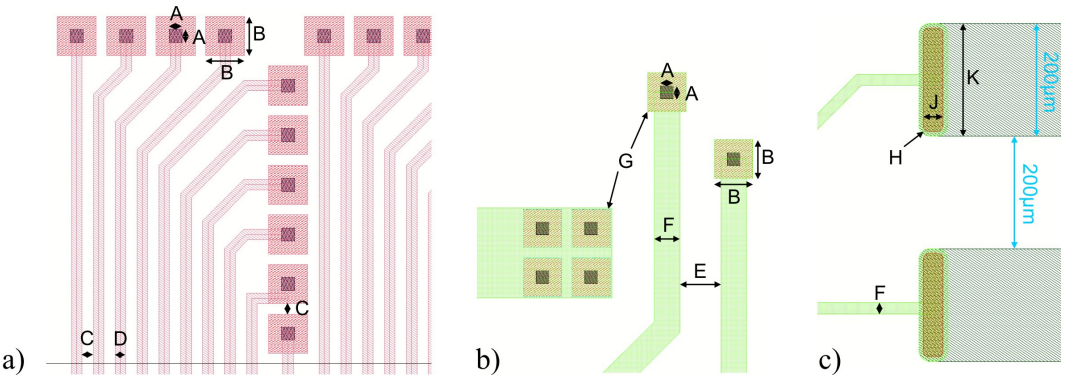


Figure 2.4.3: Illustration of PP design rules. a) fine-pitch routing with coverage of Cu-studs. b) intermediate routing with coverage of Cu-studs. c) intermediate routing with vias to top layer contact pads.

Rule	type	value [µm]	description
A	equals	5 × 5	passivation opening
B	equals	15 × 15	Cu-studs, centered above pass. opening
C	min	4	spacing between fine-pitch lines
D	min-max	4 . . . 10	width of fine-pitch lines
E	min	10	spacing between intermediate routing lines
F	min-max	10 . . . 35	width of intermediate routing lines
G	min	0	overlap of Cu-studs by intermediate routing
H	equals	8	overlap of top via on intermediate routing and top layer
J	min	35	minimum width of top via structure in all dimensions
K	min	51	minimum width of top pad structure in all dimensions

Table 2.4.1: post-processing design rules

2.4.1.2 Integration of PP Layers into ASIC Design Flow

All post-processing geometries are drawn in the Cadence DFII design environment, which is also used for designing the HICANN chip. This way, the layout can be drawn as an exact overlay of the UMC structures. Unused design kit layers are used to draw the post-processing geometries. They are listed in table 2.4.2.

A patched extraction rule file is available for Layout Versus Schematic (LVS) checking[1]. Together with a Verilog description of the reticle's connectivity (see section 2.4.2), LVS checking can be performed on the full reticle, including post-processing structures. Design Rule Checks (DRCs) cannot be performed automatically, since the according rule files have not been patched, yet. However, all required checking can be done at IZM and potential DRC

[1]The rule file is located in the HICANN full-custom repository:
`ncf-hicann-fc/units/reticle/signoff/calibre/rules/icpro_lvs_m7.rules`

stream layer	stream datatype	DFII layer name	purpose in post-processing (PP)
56	0	ME6	metal6 routing (original layer)
63	63	IPWM	intermediate PP layer 10...35 µm, incl. landing pads for 89.36
66	0	PAD	pass. opening (overlapped by "90.4 PP via")
85	0	GTEXT	PP Text Label
89	36	FLPMARKP	PP Via from top to intermediate layer
90	4	RFMMCMK	PP Via to metal6 (Cu-studs)
91	5	SBK	wafer outline
91	6	SMK	area on wafer that is usable for PP
106	0	M6_TEXT	metal6 Text Label
121	3	prBoundary	reticle outline
122	30	PIXELMK	PP top layer (stripe connector pads)
123	30	DECODER	fine pitch PP routing (at reticle boundary)

Table 2.4.2: Assignment of UMC design kit layers that have been used to draw post-processing structures. Fraunhofer IZM uses stream layer number only to identify the layers.

errors would require iterating with IZM.

2.4.2 Reticle Design

A schematic layout drawing of a reticle including the on-wafer L1 network is illustrated in figure 2.3.2. The reticle design in particular covers the connections between adjacent HICANNs within the reticle, as well as their connections to HICANNs on adjacent reticles and their connection to the stripe connector pads. A schematic description is not available, but rather a Verilog structural netlist that instantiates 8 HICANNs and defines all required intra-reticle connections and the reticle's pinout.

Inter-HICANN and Inter-Reticle Connections Adjacent HICANNs have mutual connections on the un-post-processed wafer via the L1 buses, only. Logical connectivity is identical for each two adjacent HICANNs and is described in section 2.3.2.7 and figure 2.3.6, respectively. The L1 repeaters are arranged in a way that corresponing pins are facing each other with the exception that positive and negative signal of a differential pair are swapped due to symmetries in the repeater layout. This swap needs to be reverted for each inter-HICANN connection; it is not present in the netlist and needs to be done in the layout between each two HICANNs. P/N swaps are also drawn at the bottom and left edge of the reticle layout for correct inter-reticle connectivity. The space required for these swaps, together with the L-shaped pad layout for the fine-pitch inter-reticle connections give the final reticle dimension of 20 145 µm × 20 048.2 µm, which is also illustrated in figure 2.4.4.

Connections to Stripe Connector Pads The used elastomeric stripe connectors are described in section 2.5.3.4. They offer alternating, vertically conducting and isolating slices with a

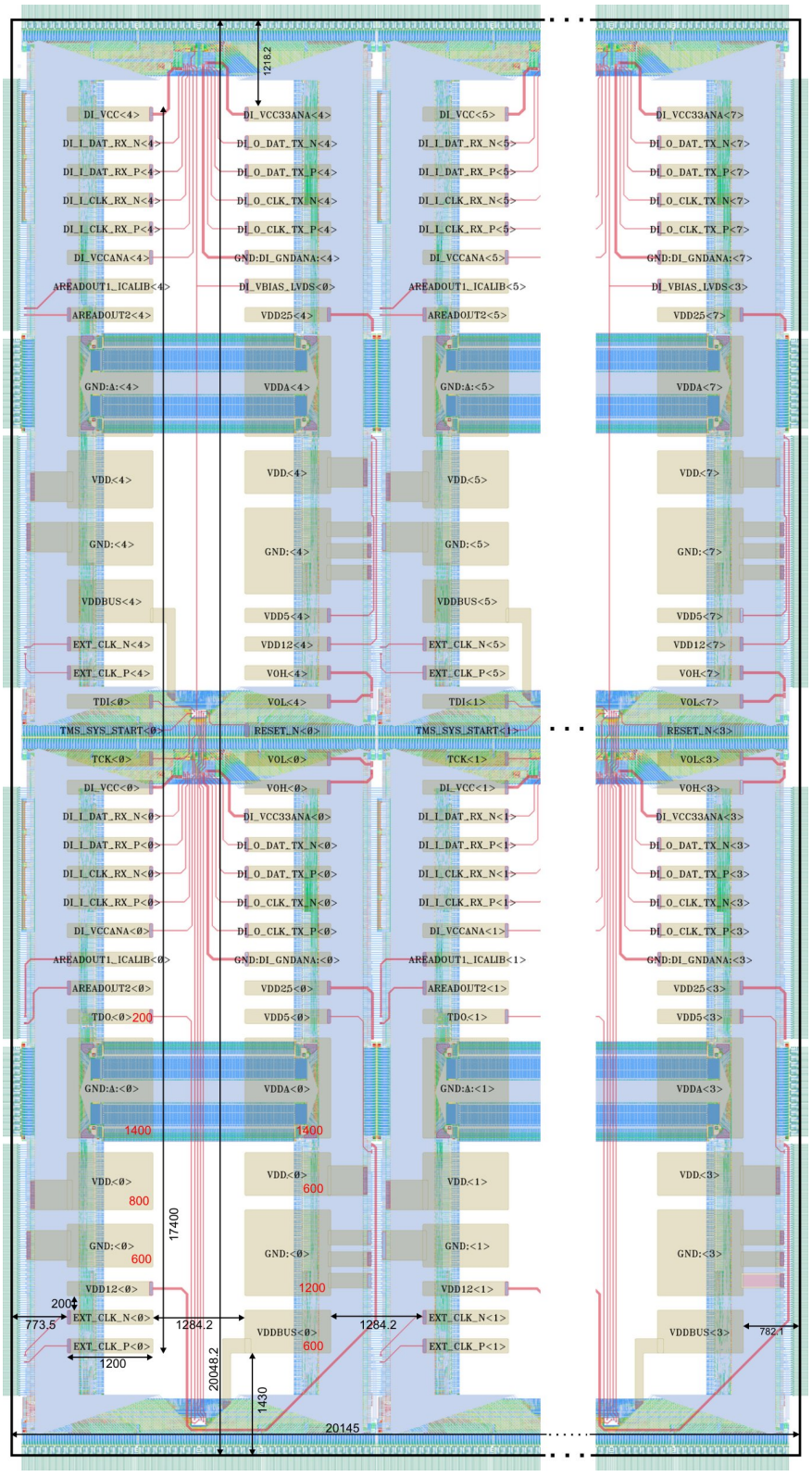


Figure 2.4.4: Layout of the stripe connector pads covering one reticle. The bold black line marks the reticle design border. Yellow: stripe connector pads. Pink: intermediate routing. The numbers in red identify the height of the resp. pad. Indices in signal names only have technical relevance and can be ignored. Actual indexing of HICANN chips is described in section 2.4.2.1.

width of 50 µm each. For reliable connectivity and avoidance of shorts, the minimum pad width and spacing was therefore decided to be 200 µm, allowing for at least two conducting slices per pad and two spacer-slices for isolation between pads. Pad length has been set to 1200 µm to accomodate the width of a connector of roughly 1000 µm with some headroom. This pad size, together with mechanical constraints for holding the stripe connectors (see section 2.5.3.3), led to the pad layout that is illustrated in figure 2.4.4, which has two columns of pads (i.e. two stripe connectors) per two vertically adjacent HICANNs. Power pads carrying large currents need to be wider due to the limited current rating of single conducting slices. For that reason, the size of all power pads has been set as an initial constraint for the stripe connector pad arrangement. All pads are shown in figure 2.4.4, including their dimensions. For information on power consumption on the different supply voltages, please refer to table 2.9.1.

Stripe Connector Pinout Pad positions on the post-processing have been chosen such that access to the corresponding top metal passivation openings could be achieved with as short as possible intermediate routing distance. The resulting stripe connector pinout of one reticle is shown in figure 2.4.5.

Analog power and ground pads for top and bottom HICANN are placed near the vertical center of each chip and simultaneously supply top and bottom chip half. Not every signal port of the HICANN chips can be connected to a dedicated stripe connector pad due to the number of usable pads being limited by the size of the power pads and the usable height inside the reticle. For that reason, some ports have to be multi-purpose and some signals have to be shared between two HICANNs:

- Multi-purpose pins
  - AREADOUT1_ICALIB: Analog readout line 1, and if analog readout is disabled inside the HICANN chip, current input for calibration of internal resistors. Present on each HICANN.
  - TMS_SYS_START: Digital input. Starts the system time counter on its positive edge when toggled after power-up or reset. Otherwise, serves as Test Mode Select (TMS) pin of the chip's JTAG interface.
- Shared pins
  - RESET_N: Low-active reset pin of both HICANNs.
  - TMS_SYS_START: Is also shared, besides being multi-purpose.
  - TCK: Clock signal of the chip's JTAG interface.
  - DI_VBIAS_LVDS: Common mode voltage of the LVDS transmitter circuits on both upper and lower HICANN.

Additionaly, the JTAG TDO pin of the top HICANN is connected to the JTAG TDI pin of the bottom HICANN via intermediate routing, effectively chaining the two vertically adjacent HICANNs together. As a consequence, only the TDI pin of the top HICANN and the TDO pin of the bottom HICANN are present on the stripe connector pads.

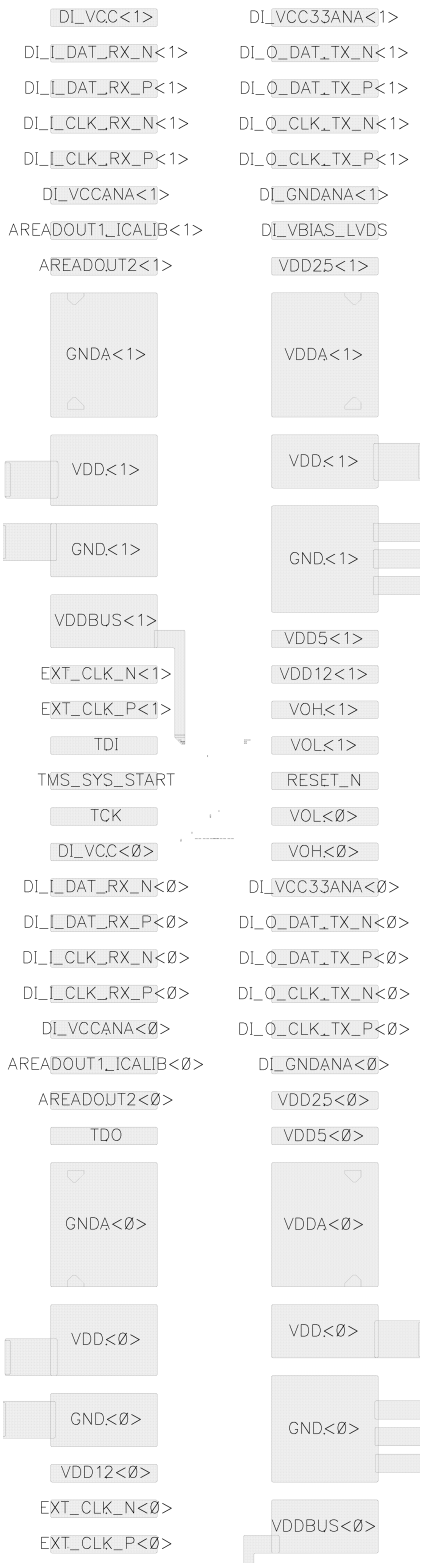


Figure 2.4.5: Pinout of one double-HICANN column.

2.4.2.1 Pinout and HICANN Indexing

The pinout of one reticle is given in table 2.4.3. Only 4 TDI and TDO pins are present, respectively due to the TDO-TDI connection within one column of two HICANNs. TDI and TDO pins of adjacent columns within one reticle need to be interconnected on the MainPCB to form the reticle's JTAG chain.

All power signals are global for one reticle and shall be shorted on the MainPCB. The signals EXT_CLK_(P/N), RESET_N, TMS_SYS_START, TCK and DI_VBIAS_LVDS are global for one reticle as well and shall also be shorted on the MainPCB.

Type	Name	Description
digital input	DI_I_DAT_RX_(P/N)<0:7>	diff. data input of Digital Network Chip (DNC) interface
	DI_I_CLK_RX_(P/N)<0:7>	diff. clock input of DNC interface
	EXT_CLK_(P/N)<0:7>	diff. system clock input
	RESET_N<0:3>	low-active design reset, shared betw. 2 vertically adj. HICANNs
	TMS_SYS_START<0:3>	shared and multi-purpose, see section 2.4.2
	TCK<0:3>	JTAG clock, shared betw. 2 vertically adj. HICANNs
	TDI<0:3>	JTAG test data input of HICANNs 1,3,5,7 (JTAG indexing)
digital output	DI_O_DAT_TX_(P/N)<0:7>	diff. data output of DNC interface
	DI_O_CLK_TX_(P/N)<0:7>	diff. clock output of DNC interface
	TDO<0:3>	JTAG test data output of HICANNs 0,2,4,6 (JTAG indexing)
analog	AREADOUT1_ICALIB<0:7>	multi-purpose, see section 2.4.2
	AREADOUT2<0:7>	analog readout line 2
	DI_VBIAS_LVDS<0:3>	LVDS common mode voltage, shared betw. 2 vertically adj. HICANNs
power	DI_VCC33ANA<0:7>	LVDS power supply of DNC interface
	DI_VCC<0:7>	digital supply of DNC interface
	DI_VCCANA<0:7>	analog supply of DNC interface
	VDD<0:7>	digital supply
	VDDA<0:7>	analog supply
	VDDBUS<0:7>	synapse line driver supply
	VDD25<0:7>	floating-gate readout supply
	VDD5<0:7>	floating-gate readout supply
	VDD25<0:7>	floating-gate programming supply
power	DI_GNDANA<0:7>	analog ground of DNC interface
	GND<0:7>	digital ground
	GNDA<0:7>	analog ground

Table 2.4.3: Pinout description of an 8-HICANN reticle.

All DI_* differential input pin pairs are 3.3 V tolerant and conform to the LVDS standard. The differential EXT_CLK input is 1.8 V tolerant and accepts standard LVDS input with a common mode of 1.25 V. All other digital pins are 1.8 V tolerant and have a single ended signal level of 1.8 V (both inputs and outputs).

The system-wise relevant indexing of HICANN chips 0 to 7 inside the reticle follows the channel indexing in the FPGA firmware on the corresponding FCP. This is identical to the indexing on the formerly used DNC, which has been used in an earlier revision of the specified system. This is inverse to the JTAG indexing that automatically results from the JTAG chain connectivity. Both channel indexing and JTAG indexing are illustrated in figure 2.4.6.

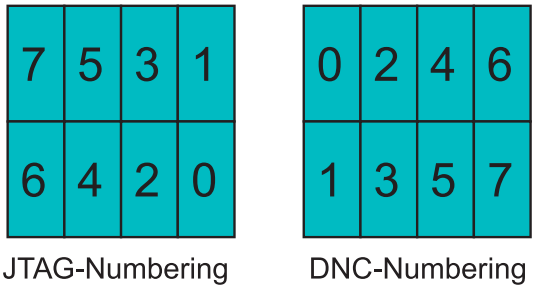


Figure 2.4.6: Indexing of HICANN chips inside one reticle for the given scenarios.

2.4.3 UMC Wafer Map and Post-Processing Masks

The position of reticles on the wafer can be determined by means of the wafer map in figure 2.4.7. The total size of one reticle, including process monitoring structures at its edges, is 20 395 µm × 20 468.2 µm and has been set by UMC, with the die size given in section 2.4.2 as a basis. How to read the map: The center of the reticle with coordinate (5,5) has an x-offset of 0 µm with respect to the wafer center and a y-offset of 10 234 µm.

Eight reticles that are touching the effect diameter of the UMC wafer map cannot be completely post-processed since the effect diameter of the post-processing procedure is smaller (in particular: X2/Y2, X3/Y1 and the point symmetric ones). The masks still cover all 56 reticles that have been marked good by UMC in order to have the post-processing structures distributed over the wafer as homogeneous as possible. However, these eight reticles are not being contacted by the MainPCB which results in a total of 48 reticles that are accessible by the FCPs. The complete drawing of the post-processing layout including all mask data is shown in figure 2.4.8.

In addition to the post-processing structure that contact the active reticles, the masks contain mechanical and electrical alignment structures that are required for a correct positioning of wafer and MainPCB with respect to each other. The position of the strucures relative to the Wafer center are as follows:

- Mechanical alignment: Geometrical center of structure:

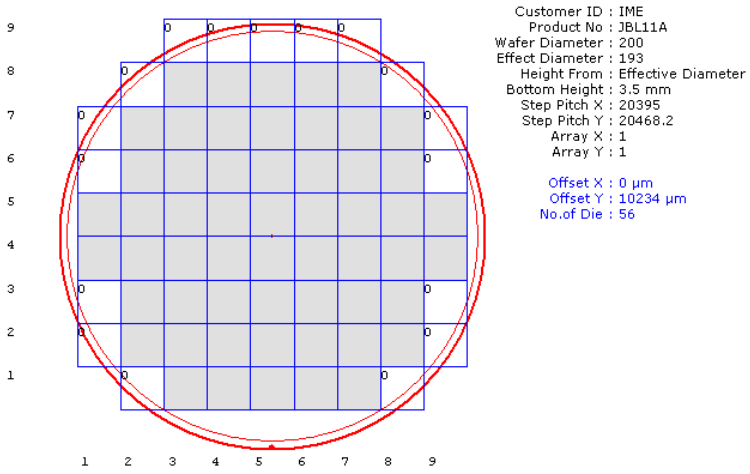


Figure 2.4.7: Wafer map of the NM-PM1 wafer.

  - top right: $(x, y) = (65\,231\,\mu m, 65\,231\,\mu m)$
  - bottom left: $(x, y) = (-65\,231\,\mu m, -65\,231\,\mu m)$

- Electrical alignment: The origin of the drawn structure is at its bottom left corner. Coordinates of the origins:
  - top left, left: $(x, y) = (-67\,589\,\mu m, 62\,266.3\,\mu m)$
  - top left, top: $(x, y) = (-72\,984\,\mu m, 54\,298.1\,\mu m)$
  - bottom right, right: $(x, y) = (72\,975.4\,\mu m, -54\,506.5\,\mu m)$
  - bottom right, bottom: $(x, y) = (67\,589\,\mu m, -62\,474.7\,\mu m)$

Please refer to chapter 2.5 for a description of the alignment and the usage of these structures.

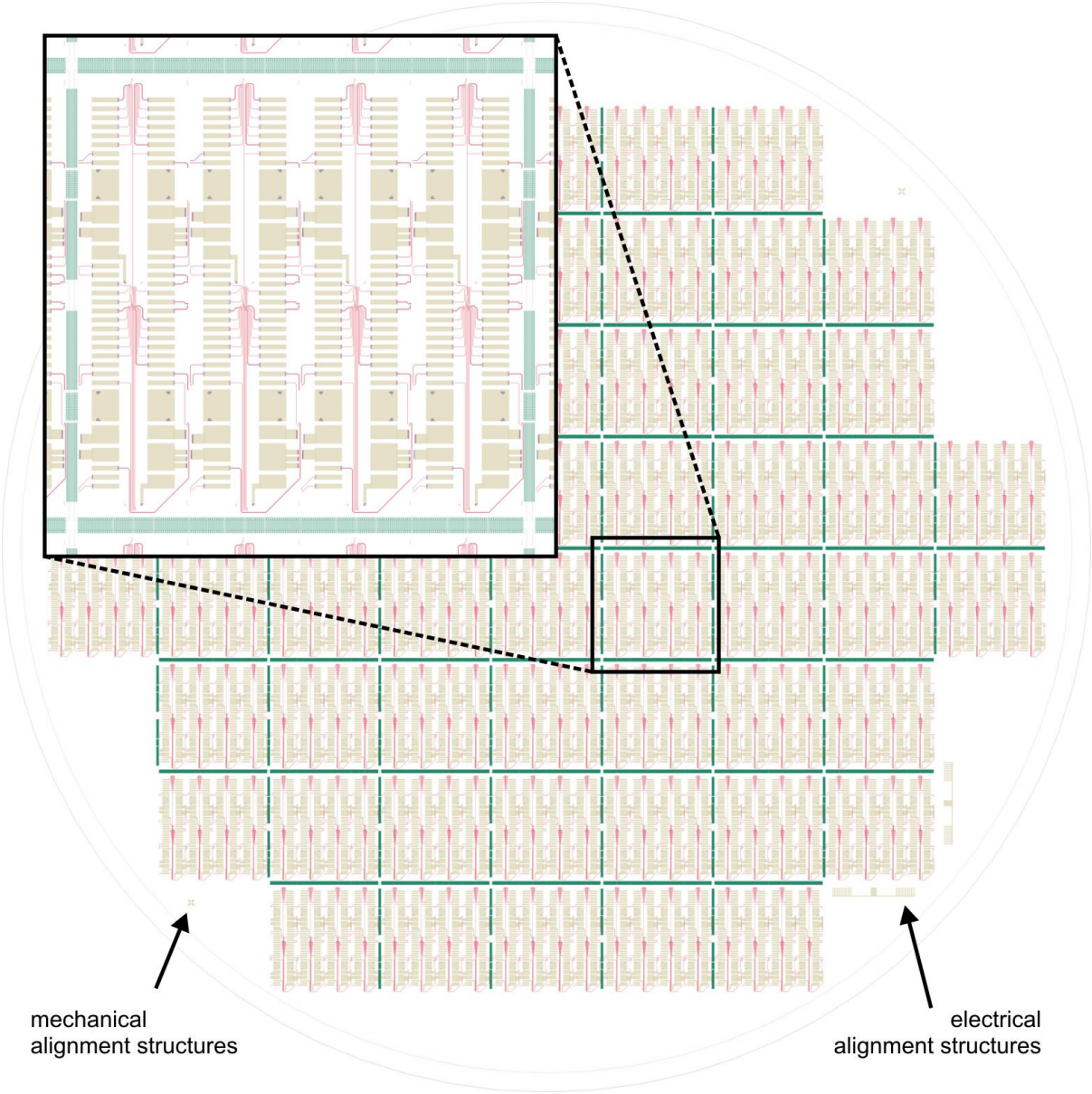


Figure 2.4.8: Full mask layout data of the post-processing structures applied to the NM-PM1 wafer. The zoomed in area shows one reticle with its surrounding fine-pitch connections to the adjacent reticles.

2.5 Wafer Module

2.5.1 Overview

The main functional unit of the Neuromorphic Physical Model version 1 is the Wafer Module. This unit consist of the HICANN Wafer and all its supporting components. These components are electric devices as well as mechanical components. Altogether one Wafer Module will consists of 68 PCBs and more than 16 mechanical components. These components such as power supply boards and communication and monitoring devices have to be fitted into a small form factor to allow scalability. The Wafer Module has only one central power supply point which supplies it with -48 V. All the internally used voltages are generated by the Wafer Module itself. Further connections to the outside of the Wafer Module are the communication channels and the analog readout signals.

2.5.2 Wafer Module Composition

This section describes the composition of the Wafer Module. The construction of such a complex system with a large number of inter depending components leads to the necessity of using a 3D CAD program. This helps solving conflicts due to component placing and helps visualizing the interplay between components. The 3D CAD program Solid Works [69] is used for that purpose. The mechanical components can be therefore directly produced by the mechanical workshop.

The main parts with their needed quantity for one Wafer Module are:

- Wafer Bracket for mechanical fixation and cooling of the wafer (WBr) (quantity: 1)
- Post processed Wafer (quantity: 1)
- Sealing rings to keep wafer under nitrogen atmosphere (quantity: 2)
- Positioning Mask for the Elastomeric Stripe Connectors for positioning of the Elastomeric Stripe Connectors (PMk) (quantity: 1)
- Elastomeric Stripe Connector (quantity: 384)

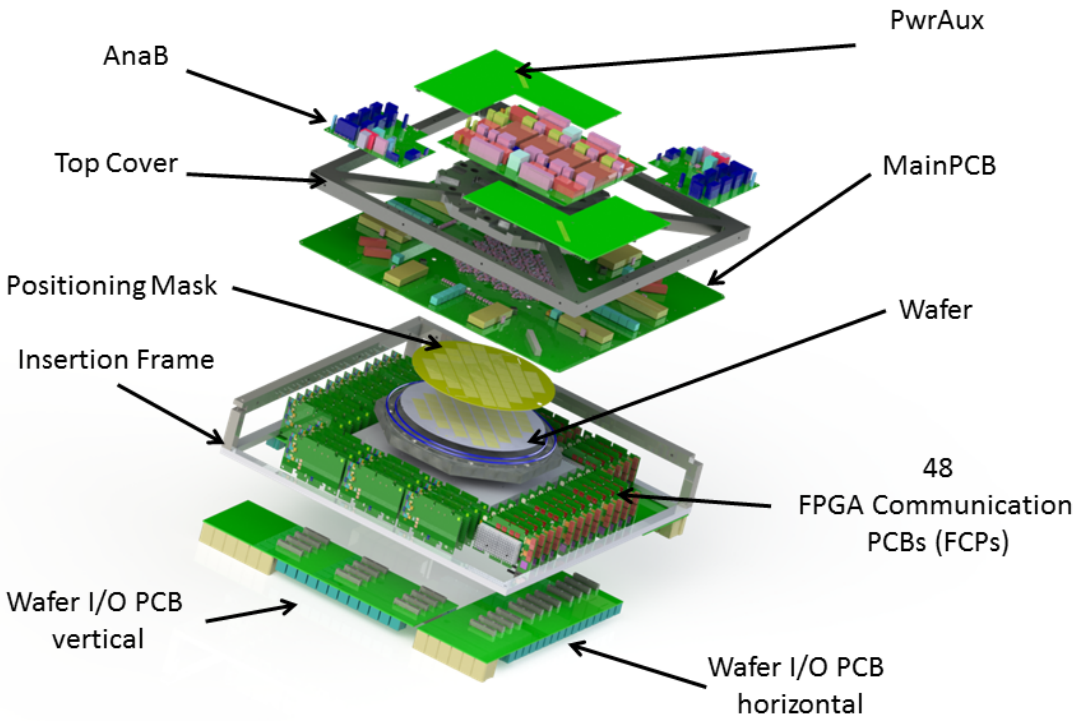


Figure 2.5.1: Exploded view of the design drawing of the Wafer Module

- Printed circuit board with a height of 430 mm and a width of 430 mm (MainPCB) (quantity: 1)
- FPGA based communication PCBs (FCP) (quantity: 48)
- Physical-layer and communication connector boards with horizontal orientation (WIOH) (quantity: 2)
- Physical-layer and communication connector boards with vertical orientation (WIOV) (quantity: 2)
- Breakout boards for the analog readout signals of the Wafer (AnaB) (quantity: 2)
- Main system control unit (MaCU) (quantity: 1)
- Main Power supply board delivering the main 1.8 V wafer supply and the 10 V intermediate power supply (PowerIt) (quantity: 1)
- Auxiliary Power Supply PCB for the remaining supply voltages (AuxPwr) (quantity: 2)
- Top Cover for the mechanical stability of the system (ToCo) (quantity: 1)

- Insertion frame for mounting of additional PCBs (InFra)
  (quantity: 1)

2.5.3 Mechanical Specification of Components

This section provides a description of all used components in the Wafer Module and the connectors between them. The mechanical sizes and other important production details such as screw holes can be found in the appendix A

2.5.3.1 Wafer Bracket (WBr)

The Wafer Bracket (WBr) is an equilateral octagon made out of a planarized aluminum plate. The space between two opposite parallel edges of this octagon is 250 mm. The thickness of the plate is 15 mm. On the left side of fig. 2.5.2 the top view of the WBr is shown. The large round cavity in the middle of the octagon is for the placement of the Wafer. The two coaxial flutes surrounding this cavity are for the sealing rings that guarantee air tightness. In between these flutes different holes in a circular pattern are visible. The holes with the largest diameter are for distance plates to adjust the distance between wafer and MainPCB. This is used to adjust the maximal compression length for the Elastomeric Stripe Connectors and to balance possible differences in Wafer thickness.

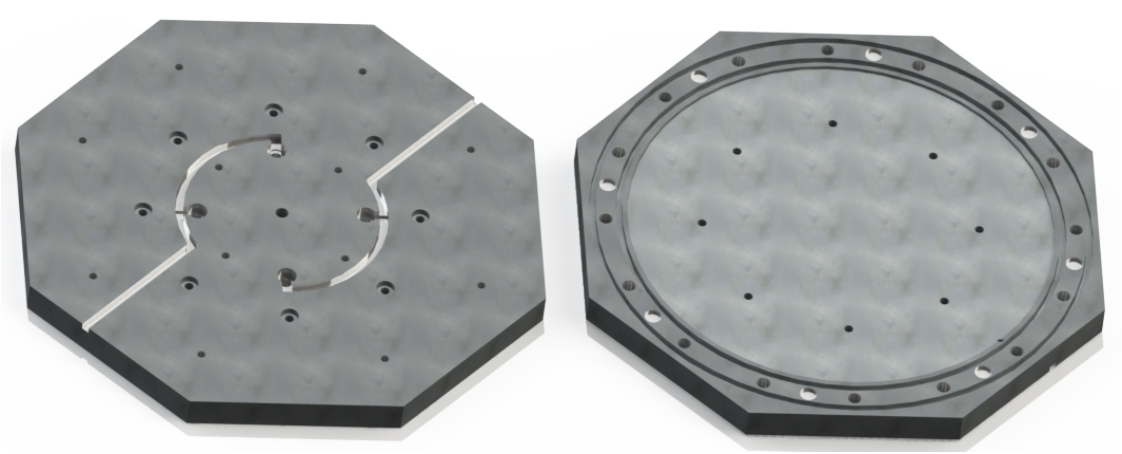


Figure 2.5.2: Top and bottom view of the Wafer Bracket.

The smaller holes without the thread of screw are for springs which are used during the alignment process of the system. These springs make sure that the MainPCB is kept in a determined distance above the Wafer surface. In this position the Elastomeric Stripe Connectors are not touching the wafer and the adjustment procedure can be performed. In the other stud holes are inlaid *Heli-Coils*[1] which are used to screw the Top Cover against the Wafer Bracket.

[1]Heli-Coil inserts are precision formed screw thread coils of stainless steel wire having a diamond shaped cross-section thread inserts

A concentric round cavity with a diameter of 200.2 mm and a depth of 1.325 mm is milled around the center of the plate. In this cavity the Wafer will be placed. The diameter of the milling is greater than the Wafer diameter to balance the differences due to thermal expansion. The cavity is about 0.8 mm deeper than the thickness of the Wafer with the post processing above (about 0.75 mm). The free space will be covered by the ElCos.
To avoid contact problems between the wafer and the Main-PCB produced by oxidation the clearance is filled with nitrogen. For the necessary air tightness a sealing ring has to be placed between the WBr and the MainPCB. To avoid mechanical strain during the assembly process two sealing rings instead of one are used. They are placed on both sides of the screw holes. These sealing rings are placed into two round gaps with inner diameters of 205 mm and 230 mm. The gaps will have a thickness of 3 mm and a depth of 2.5 mm.

In the left side of figure 2.5.2 the bottom view of the WBr is shown. The hole in the center is used to place the Wafer-bracket into the adjustment facility. The drill holes with the counter sunk are for the insertion of a mechanical tool which is used to push out a placed wafer. In order to keep the system airtight these holes will be closed by screws with a silicon inlay.
The two channels ending in two holes each are used to place temperature sensors and their belonging cabling. These sensors are placed close to the backside of the Wafer and so temperature profiles can be recorded.

2.5.3.2 Sealing Rings

The Wafer should be kept under nitrogen atmosphere after the system is assembled. Therefore sealing rings are placed between the WBr and the MainPCB. They have diameters of 205 mm and 230 mm and a diameter of the cross section of 3 mm. These sealing rings will make sure that no air can enter from the sides.
To guarantee that no air can enter from the top, the MainPCB has to be hermetically sealed. The material of the MainPCB is FR4[2] which is not airtight by default. The used vias should not be drilled through. Therefore only blind or micro vias are used inside the covered area.

2.5.3.3 Positioning Mask for the Elastomeric Stripe Connectors (PMk)

The Positioning Mask for the Elastomeric Stripe Connectors (PMk) is a 0.51 mm thick mask made of a non-conducting composite material. Good results were achieved using 0.5 mm thick FR4 material which is used in the PCB production process. It is used for the positioning and fixation of the ElCos.

The mask is connected with 4 countersunk head screws which sunk-in the mask and which are screwed through the MainPCB into the Top Cover (ToCo).
Figure 2.5.3 shows the distribution of the cut-outs which match the position of the ElCos. The shown mask has 388 cut-outs, one for each ElCo. The shape of a cut-out is not a perfect

[2] FR-4 is a composite material composed of woven fiberglass cloth with an epoxy resin binder that is flame resistant (self-extinguishing).

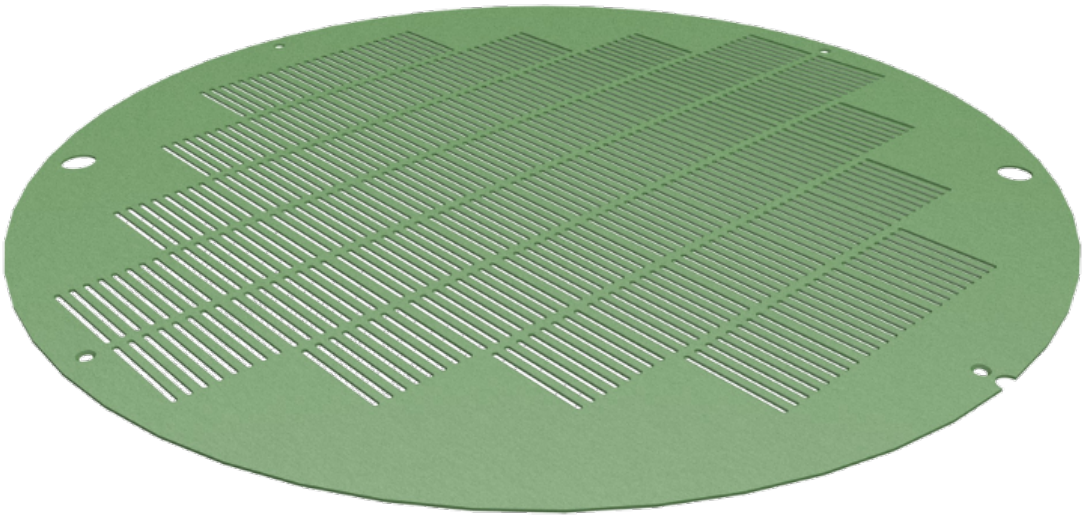


Figure 2.5.3: Angular view of the Positioning Mask for the Elastomeric Stripe Connectors out of SolidWorks.

rectangle. Figure 2.5.4 shows a detailed view of one cut-out. The circular arcs on the short sides of the cut-out have a diameter of 1.1 mm. This is a result of the usage of a 1.1 mm milling cutter. To give additional space to the ElCos during compression the long sides of the cut-out are not milled straight but are following an arc with a radius of 400 mm. Because of these arcs the smallest gap in the longer direction is 1.1 mm and the widest gap is 1.2 mm.

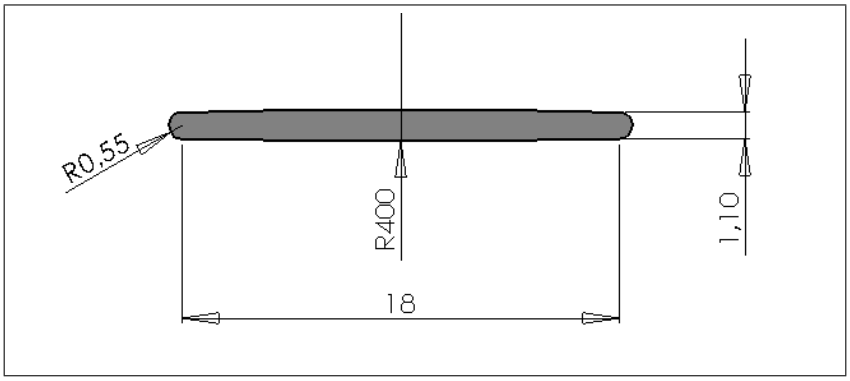


Figure 2.5.4: Dimensioned drawing of one cut-out of the Positioning Mask for the Elastomeric Stripe Connectors.

For the optical alignment and for the filling with nitrogen the 6.4 mm holes are needed. The smaller holes are used for screwing the mask through the MainPCB to the ToCo.

2.5.3.4 Elastomeric Stripe Connectors (ElCo)

The wafer will be electrically connected to the MainPCB by using flexible Elastomeric Stripe Connectors from the *Zebra Silver* series from Fujipoly [37].

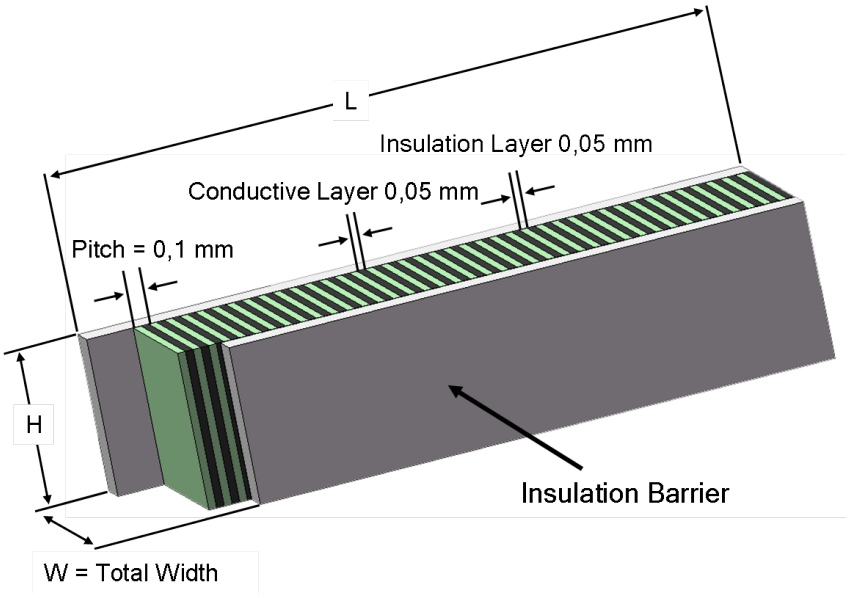


Figure 2.5.5: Close-up view of an Elastomeric Stripe Connector

The length of the connectors is 18 mm, their height is 1 mm and their width is 1 mm. The ElCos are made of two alternating layers of non-conducting silicon and silicon that is made conductive by interlaced silver balls. The contact pitch is 0.1 mm. Figure 2.5.5 shows a close-up view of such a connector.

Due to the fine pitch a connector of 1 cm length has 100 contact planes. Combining several contacts can help balancing the tolerances in the length direction (see b of fig. 2.5.6). The contact areas made by the post processing (see section 2.4.1) on the Wafer have a width of 0.2 mm and there will be a space of 0.2 mm between two adjacent ones. This method guarantees the usage of at least two contact surfaces. With the used pitch of 400 µm the total number of 180 contacts per stripe is reduced to one fourth and therefore to 45 contacts per cm stripe. Because of the uncertainty in positioning the connector only 43 contacts will be used. The power connections are spread over several contact areas and the space between the contact areas is in this case filled. The tolerance in the crosswise direction of the ElCo can be equated by using wider contact pads on the wafer and the board (see a) of fig. 2.5.6). This limits only the number of usable connectors. But even with a space of 1 mm between two adjacent connectors 10 of these can be placed on each reticle.

To compensate the tolerances in the height a more complicated consideration has to be done. The resistance of the connectors is almost constant in a range of 10% up to 40% compression. If a connector of 1 mm height is used the range will be from 0.9 mm down to 0.6 mm. This can cover a variation in the z-direction of 0.3 mm as shown in c) of fig. 2.5.6.

The volume resistivity of the connectors is $R_V = 10^{-5}$ Ωm. The formula for the calculation

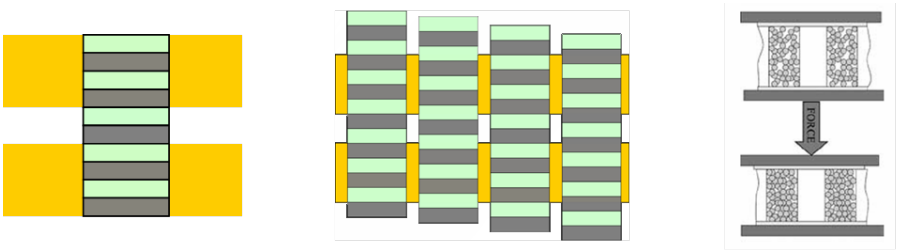


Figure 2.5.6: Tolerance balancing of the Elastomeric Stripe Connector in all three dimensions

of the resulting resistance is $R = \frac{R_V \times H}{A}$ (A is the area of the cross-section).
The resulting resistance for each contact is:
$R = \frac{10^{-5}\ \Omega\ m \cdot 10^{-3}}{0{,}05 \cdot 10^{-3}\ m \cdot 10^{-3}\ m} = 200\ m\Omega$.
The necessary pressure for a compression of 10% is for one ElCo is about 0.4 N. For a compression of 40% it rises to 1.5 N.
To reach a medium compression of about 25% 0.8 N is needed for each connector.

Number of contacts and necessary force in dependence of the number of Elastomeric Stripe Connectors

The number of connectors determines the force needed to assemble the whole system. The number of usable contacts for each HICANN is depending on the number of connectors. As already mentioned one fourth of the contacts of a connector can be used for signals. For power or ground connections the contacts on the Wafer can be stretched over several contacts without gaps.
Table 2.5.1 shows the number of usable signals per HICANN depending on the number of used connectors and the percentage of pins for power or ground. It also shows the resulting cumulative force needed.

2.5.3.5 Wafer Module Main PCB (MainPCB)

The main component for the integration of the Wafer into an electrical and mechanical framework is a printed circuit board called MainPCB. This MainPCB has to distribute the supply voltages to the HICANN chips on the wafer. It has also to distribute the power supply to the FPGA Communication PCBs (see section 2.9.1). Besides that it has to fan out the differential signals to the connectors of the FCP (see section 2.6.2.1).
The size of the wafer and the complexity of the requirements leads to a board size of 430 x 430 mm^2 with a thickness of 2.1 mm. Figure 2.5.7 shows the top side of the MainPCB.
Two different areas on the surface of the PCB can be spot. The first one is the inner part of the PCB which shows a regular pattern of little squares. These squares are the equivalent to the reticles on the Wafer. To avoid destructions on the wafer the voltage supply of each reticle is monitored and can be switched on and off individually (see chapter 2.9). The pads of the necessary components for this task such as Power Field-Effect Transistors and blocking

$E_{reticle}$	$Part_{PWR}$ [%]	$HICANN_{PWR}$	$HICANN_{sig}$	E_{total}	F_{total} [N]
4	40	17	14	180	192
4	50	21	11	180	192
4	60	26	9	180	192
6	40	25	20	270	288
6	50	31	16	270	288
6	60	39	13	270	288
8	40	35	27	360	384
8	50	43	22	360	384
8	60	53	18	360	384

Table 2.5.1: $E_{reticle}$: ElCos per reticle, $Part_{PWR}$: percentage of power and ground pins, $HICANN_{PWR}$: power pins per HICANN, $HICANN_{sig}$: signal pins per HICANN, E_{total}: total number of connectors, F_{total}: cumulative force for all connectors

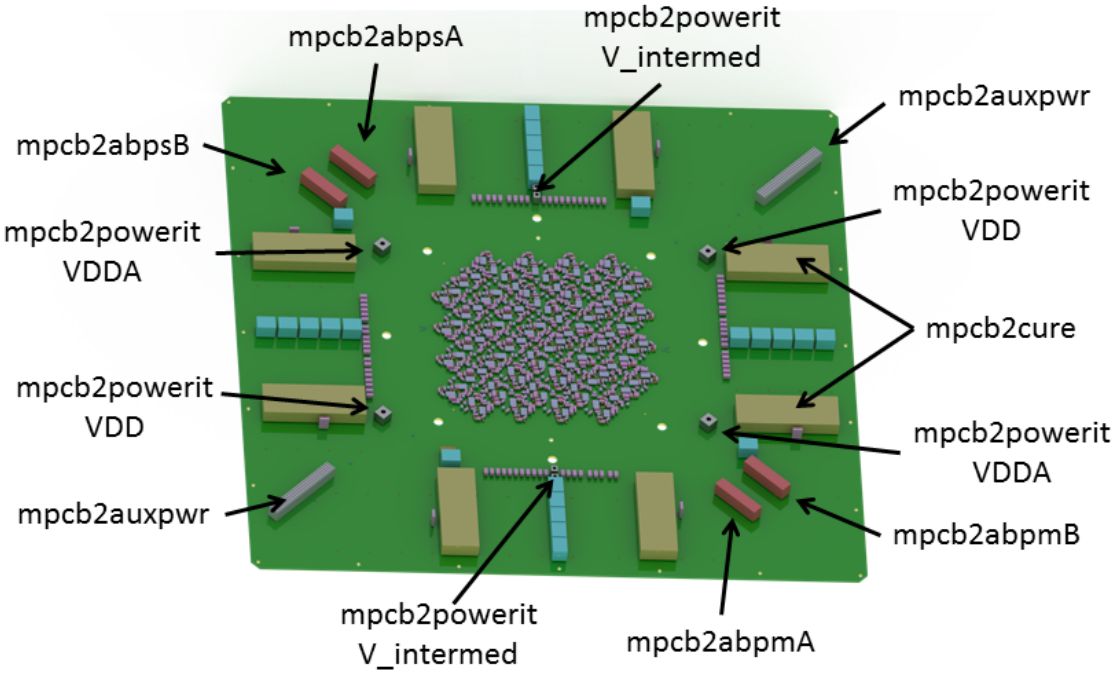


Figure 2.5.7: Angular view of the top side of the MainPCB with the connectors marked

capacities are forming the visible pattern. Each HICANN chip has 12 different supply voltages which should be monitored and switched on and off separately. But the total amount of 384 HICANNs on the Wafer together with the high number of voltages affords a trade-off. Not each HICANN, but each reticle (8 HICANNs) can be switched on or off individually. This monitoring task is done by 8 daughter boards (Cures). These boards are connected to the MainPCB using 8 sodimm-connectors placed on the four edges. The pin-out of these Sodimm

connector between MainPCB and Cure PCBs (MainPCB to Cure PCB connectors) connectors is shown in figure 2.5.17

At the right-bottom and the top-left corner of the MainPCB four connectors are visible which are rotated by 315 degrees. On top of these connectors the AnaBs will be placed. These PCBs will be split into one master and one slave module. The master module will be placed at the right-bottom corner of the MainPCB.

At each corner of the inner reticle area the connectors for the main power supply of the Wafer are placed. In the upper-left and in the lower-right corner 1.8 V analog power supply voltage for the Wafer (1.8 V) is fed in. In the upper-right and in the lower-left corner 1.8 V digital power supply voltage for the Wafer (1.8 V) is fed in.

The other four smaller power connectors are for the intermediate voltage for the Wafer Module (7-13.5 V). This power supply is directed through the MainPCB to the FCPs.

Two more connectors with a rotation of 45 degrees are visible in the corners. These 120 pin PC104plus power connector for the auxilary power supply of the MainPCB and the Wafers are used to supply the Wafer with the additional voltages. The pin-out of these connectors is shown in fig. 2.5.8

The AnaB PCB which is located on the right-bottom corner of the MainPCB (see fig. 2.5.7 is called master.

The electrical connection of the MainPCB and the Wafer will be formed over 384 ElCos (see section 2.5.3.4) placed on the bottom side of the PCB. The connection is organized in pairs of Elastomeric Stripe Connectors. Therefore two HICANNs stacked by their short side are connected via one connector pair called connector pair between MainPCB and Wafer (MainPCB to Wafer connector pair). The configuration of this connector pair can be found in section 2.4.2. Figure 2.5.11 shows the layout of the bottom side of the MainPCB. In the middle a pattern with the pads for those connectors is visible. The wider contacts are for the supply voltages and the smaller ones are for data and control signals. The figure shows also that the reticles are turned through 45 degrees. This is done in order to gain more routing area. Each reticle has 32 impedance controlled differential pairs to be routed to the side of the board. The impedance for each pair should be 100 ohms. With a good signal-to-noise ratio this results in a space necessity of about 750 µm per pair. The number of staggered reticles and the limited number of layers on the PCB prohibits the usage of more than one layer for this routing. Without the turning of the wafer the routing space has a maximum of 20 mm. 32 pairs with 750 µm each are dissipating at least 24 mm of space. With the 45 degree turn of the wafer the routing area can be increased by a factor of 1.4 up to 28 mm.

The production method of the MainPCB and the system requirements let to constraints for the layout of the MainPCB.

- The tolerances in the thickness of the board over the whole area are usually about ±10 %. This would lead to a tolerance of ±0.21 *mm*. The tolerances are mainly determined by the proportion of metalized and non-metalized areas. Therefore a highly symmetric layout was chosen to minimize the thickness variations. Another influence is the thickness of the Carbon fiber reinforced plastic CFRP (PrePreg) layers. With a careful layout the tolerances should be about ±0.1 mm.
- Gas-tightness of the board area on top of the wafer can be achieved by using only blind- and buried-vias in this region.

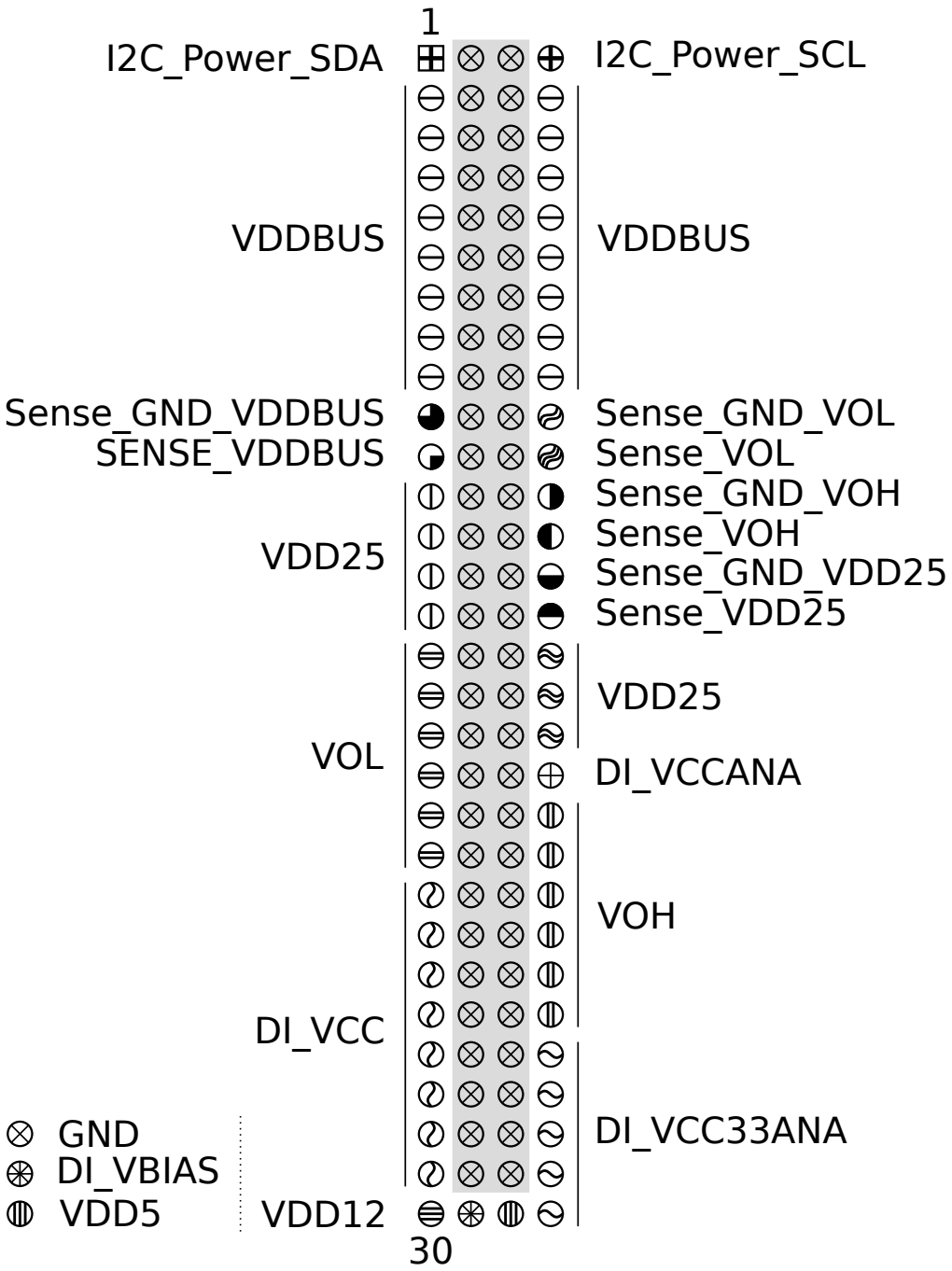
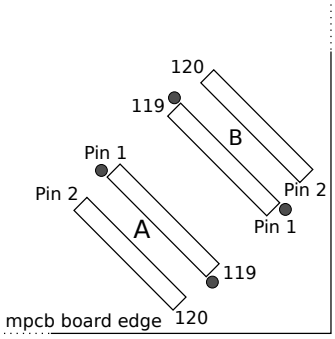


Figure 2.5.8: Pinout of the AuxPwr connector

mpcb2anab master connector



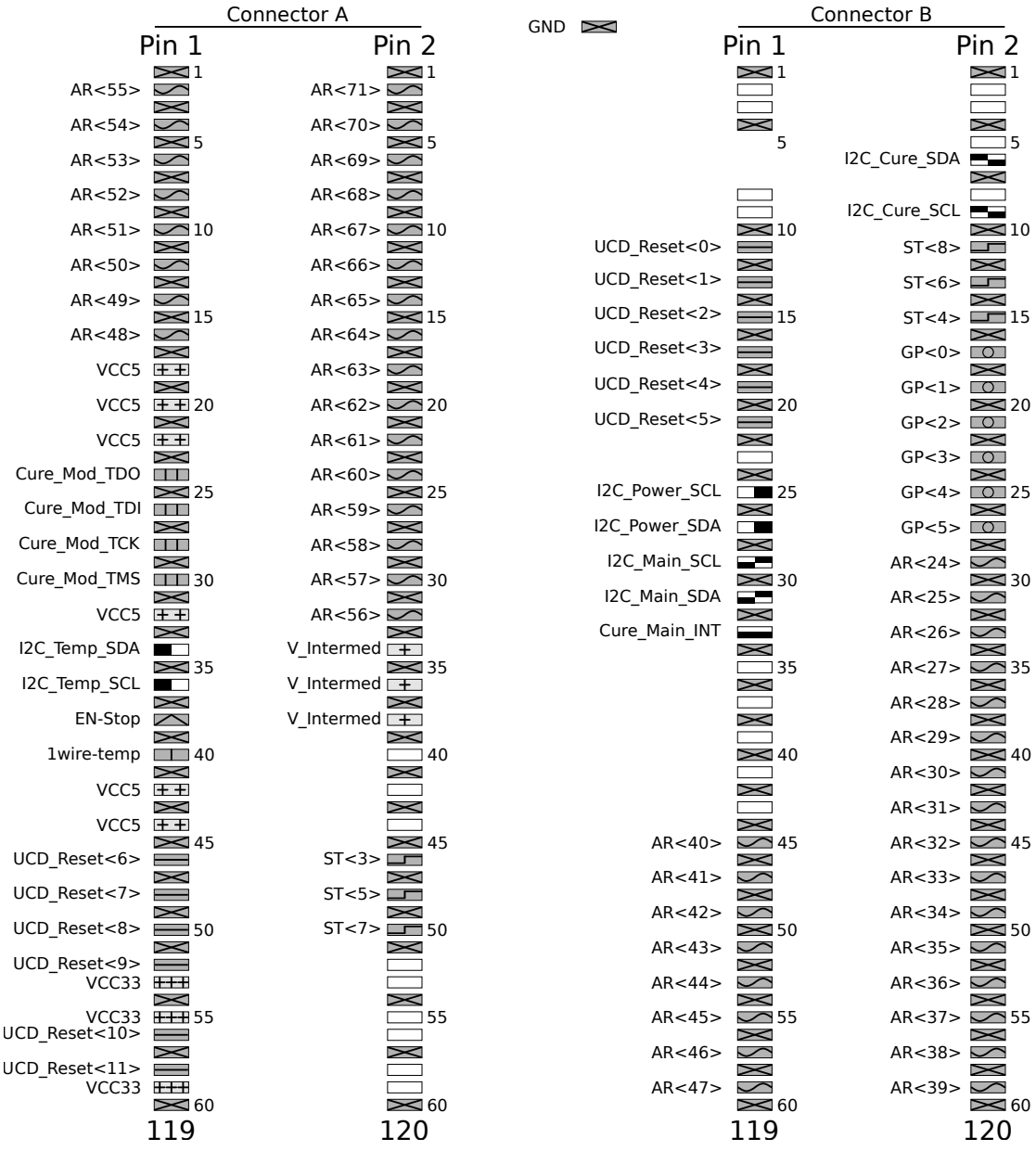
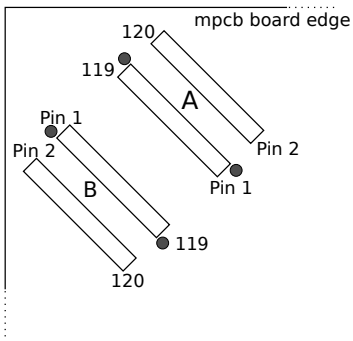


Figure 2.5.9: Pin-out of the MainPCB to AnaB-Master connector A and MainPCB to AnaB-Master connector B

mpcb2anab slave connector



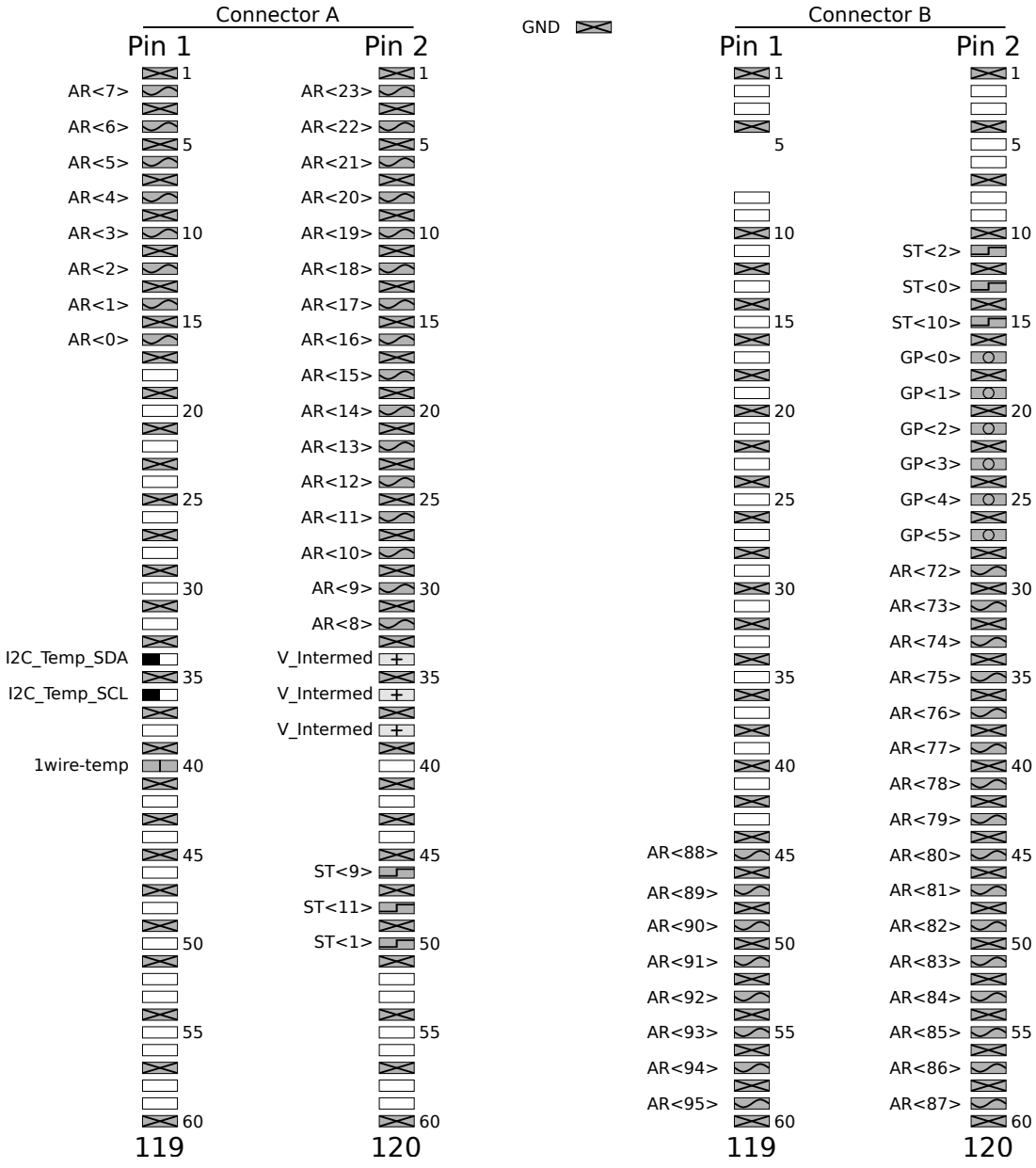


Figure 2.5.10: Pinout of the MainPCB to AnaB-Slave connector A and MainPCB to AnaB-slave connector B

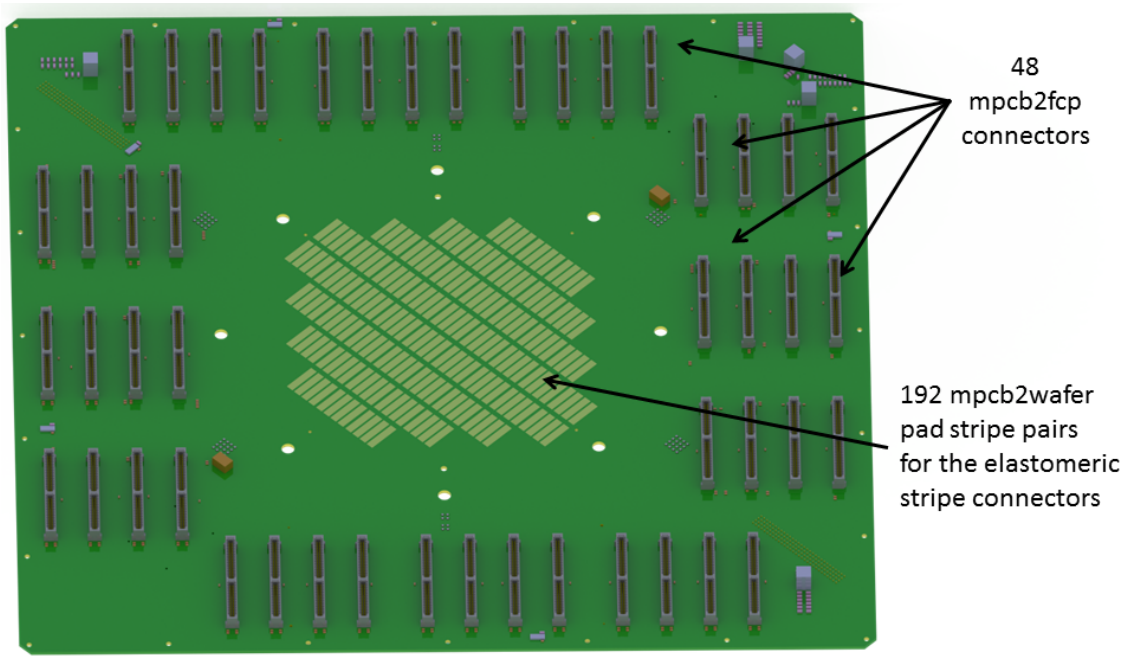


Figure 2.5.11: Angular view of the bottom side of the MainPCB

Layer structure and via stack

The high signal-density and the large amount of supply power needed leads to the necessity of a multi-layer fine pitch board. From the inner part of the MainPCB more than 1500 differential pairs and 400 monitoring signals have to be routed to the connectors at the edges. The differential pairs have to be impedance controlled with an impedance of 100 ohms. To achieve these requirements and to keep an acceptable signal-to-noise ratio a lot of routing space is needed. Therefore all the differential pairs of a reticle have to be routed in one routing layer. The reticles behind (closer to the center) are also needing the full reticle space for routing which efforts the usage of laser drilled micro-vias. These micro-vias are connecting a layer only with its directly underlying layer and therefore they are not visible in the layer beneath that underlying layer.

The usage of this kind of micro-vias appoints to a certain layer thickness and a special manufacturing method. The board has 14 layers with two cores in the middle which results in a total thickness of 2.1 mm. Figure 2.5.12 shows the layer structure and the usage of the different layers. The starting point of the production is a four layer board with the layers grouped symmetrically around two cores. Through this base only mechanical drilled vias can be used. They are visible on all the 4 layers. Now five identical production steps are following. In each step two foils will be glued on both sides of the PCB. Then the copper structure of this layers will be developed and the foils will be fixed on the board using an operation with high pressure and high temperature. Now the micro-vias on the both outside layers can be drilled with a laser using a diameter of 0.15 mm. The problem with this

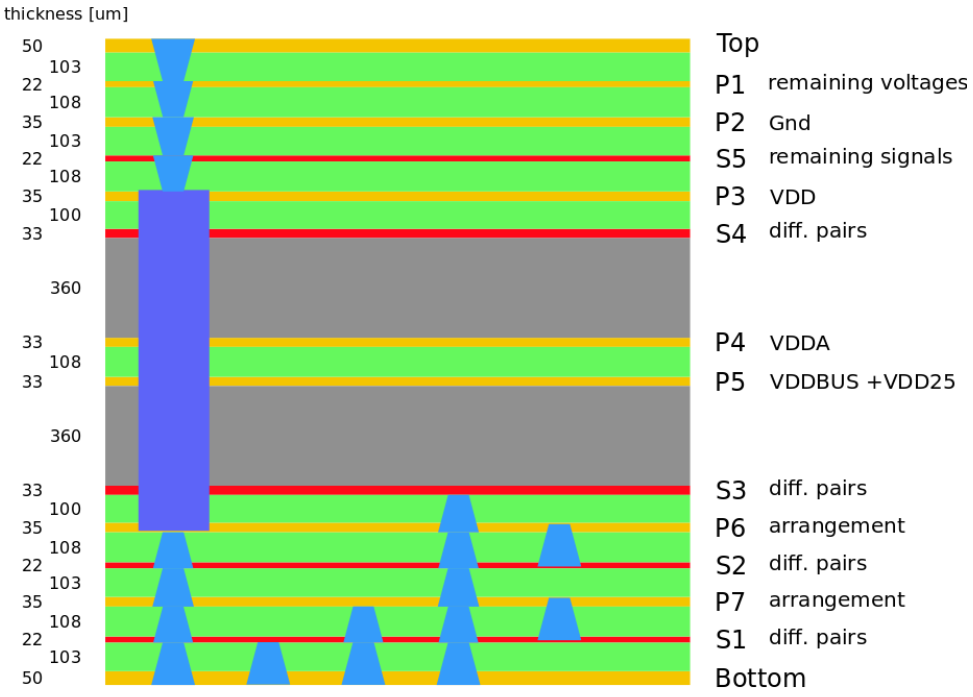


Figure 2.5.12: Layer structure and via stack of the MainPCB.

method is that the material gets more and more brittle with each production step. In normal production only 4 of these steps are allowed. A cautious handling of these processes with an accurate monitoring after each step allows the extension to 5 pressing operations. This leads to the maximum of 14 layers for the MainPCB.

2.5.3.6 Main power supply board (PowerIt)

To minimize the necessary cabling into and inside the Wafer Module there is only one power supply entry point into the Wafer Module. This point is located at the PowerIt Main Power Supply PCB. The board width is 250 mm and the height is 240 mm. Figure 2.5.13 shows a computer-generated picture of the top side of the board.

The input voltage of the PowerIt is main input voltage of the Wafer Module (-48 V). The PowerIt will generate 4 voltage supplies out of this input voltage.

- The first voltage supply is the 1.8 V analog power supply voltage for the Wafer (1.8 V).
- The second voltage supply is the 1.8 V digital power supply voltage for the Wafer (1.8 V).
- The third voltage supply is the intermediate voltage for the Wafer Module (7-13.5 V).
- The fourth voltage supply is the 5 V standby supply voltage for the Wafer Module which will supply the Main System Control Unit. This voltage will only be switched of in case

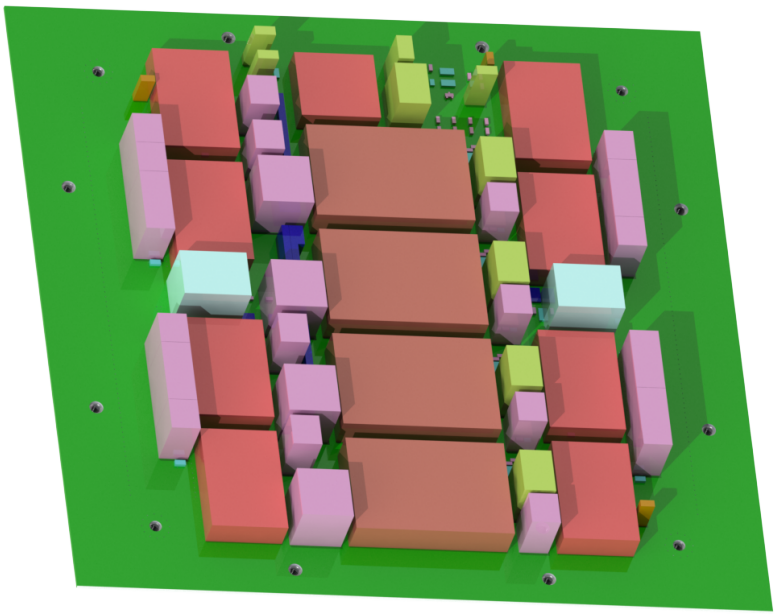


Figure 2.5.13: Top view of the PowerIt Main Power Supply PCB. Computer-generated picture made with SolidWorks

of a power failure (such as input over-voltage) of the PowerIt itself.

The intermediate voltage will be used as a supply for different components of the Wafer Module. A detailed description of the power distribution can be found in section 2.9.1 and is illustrated in figure 2.9.1.

- Power supply of the 48 FPGA Communication PCBs.
- Power supply of the 4 Wafer I/O PCBs
- Power supply of the 2 Auxiliary Power Supply PCBs.
- Power supply of the 2 Analog Breakout PCBs.

2.5.3.7 Auxiliary Power Supply PCB (AuxPwr)

The remaining voltages of the Wafer are produced by two Auxiliary Power Supply PCB. The width of the Auxiliary Power Supply PCB is 208 mm and the height is 208 mm. To fit into the Wafer Module the board will have a rectangular cut-out with a width of 116 mm and a height of 116 mm. The input voltage of these AuxPwrs is the intermediate voltage generated by the PowerIt board. The boards are placed in the upper right and in the bottom left corner of the MainPCB and each board supports one half of the wafer. Figure 2.5.7 shows top view of MainPCB with the two AuxPwrs visible. In fig. 2.5.14 a bottom view of the AuxPwr is shown. The 120 pin PC104plus power connector for the auxilary power supply of the MainPCB and

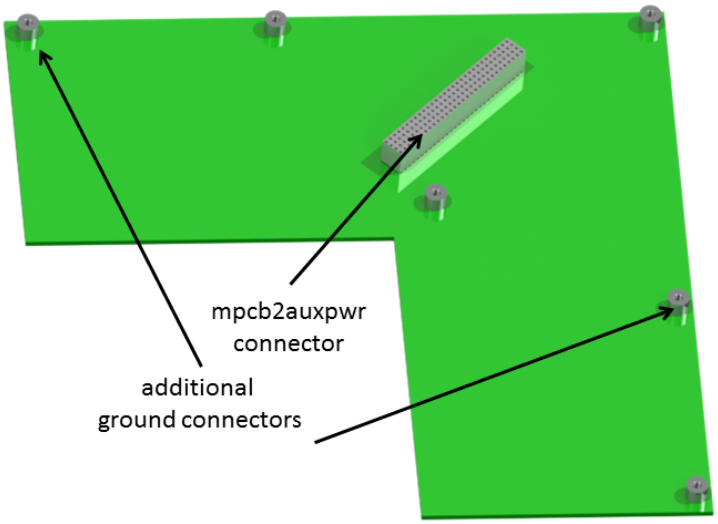


Figure 2.5.14: Angular view of the back side of the Auxiliary Power Supply PCB. Computer-generated picture made with SolidWorks.

the Wafer and some additional ground connectors are visible. The fact that the rest of the PCB remains empty is due to the fact that the AuxPwr is still under development.

The 10 voltages generated by the AuxPwr are:

1) VDDBUS: synapse line driver supply (VDDBUS)
2) VDD25: floating-gate programming supply (VDD25)
3) DI_VCCana: analog supply of DNC interface (DI_VCCana)
4) V_OL: lower voltage level for the Layer 1 signaling (V_OL)
5) V_OH: upper voltage level for the Layer 1 signaling (V_OH)
6) DI_VCC: digital supply of DNC interface (DI_VCC)
7) DI_VCC33ana:LVDS power supply of DNC interface (DI_VCC33ana)
8) LVDS common mode voltage (DI_Vbias)
9) VDD5: floating-gate readout supply (VDD5)
10) VDD12: floating-gate programming voltage (VDD12)

All the generated voltages are monitored by a micro-controller placed on the AuxPwr. In case of voltage problems it will shut down the output voltages and send a message to the Main System Control Unit.

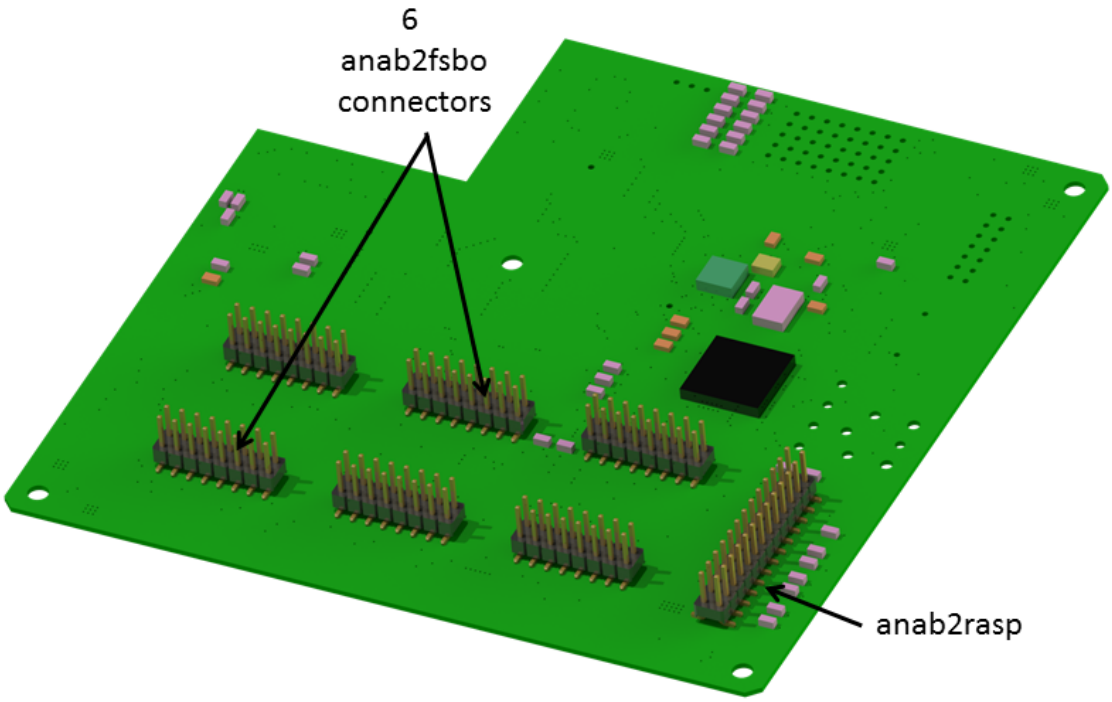


Figure 2.5.15: Angular top view of the Analog Breakout PCB.

2.5.3.8 Breakout PCBs for analog readout signals of the Wafer (AnaB)

Each of the 48 reticles of the Wafer delivers 2 analog signals which sums up to 96 signals for the Wafer. These signals are routed as 50 ohm terminated single ended lines to the corners of the MainPCB which are not occupied by the AuxPwrs.

Each of the two AnaB PCBs collects 48 analog readout and 6 trigger signals from the 24 FCPs. Always four adjacent FCPs are sharing one trigger signal. These signals are then delivered to the boards via 2 fine pitch connectors. The pin-out of these connectors is shown in the figures 2.5.9 and 2.5.10. The AnaB PCB which is located on the right-bottom corner of the MainPCB (see figure 2.5.7 is called master. This PCB connects the MainPCB to the Main System Control Unit (MaCU). The connection to the MaCU is established via the 26 pin connector for the connection of Analog Breakout PCB with Raspberry Pi (AnaB to Rasperry Pi connector) connector using a ribbon cable. It passes the 5 V standby supply voltage for the Wafer Module to the MaCU and the it generates the 5 V supply voltage for the Monitoring and Control PCB for Reticles. It also handles all the Inter-Integrated Circuit Link (I2C) buses for power and temperature control of the system (see section 2.9.2.1). Each AnaB collects the analog signals of 24 reticles and distributes them to 6 standard 16 pin SMD pin header connectors marked as 16 pin connector for the connection of Analog Breakout PCB with Flyspi Breakout PCB (AnaB to FsBo connector) in the fig. 2.5.15. Each of these connectors will deliver 8 analog signals together with one trigger signal via a shielded ribbon cable to the farm of Analog Readout Module placed outside the Wafer Module.

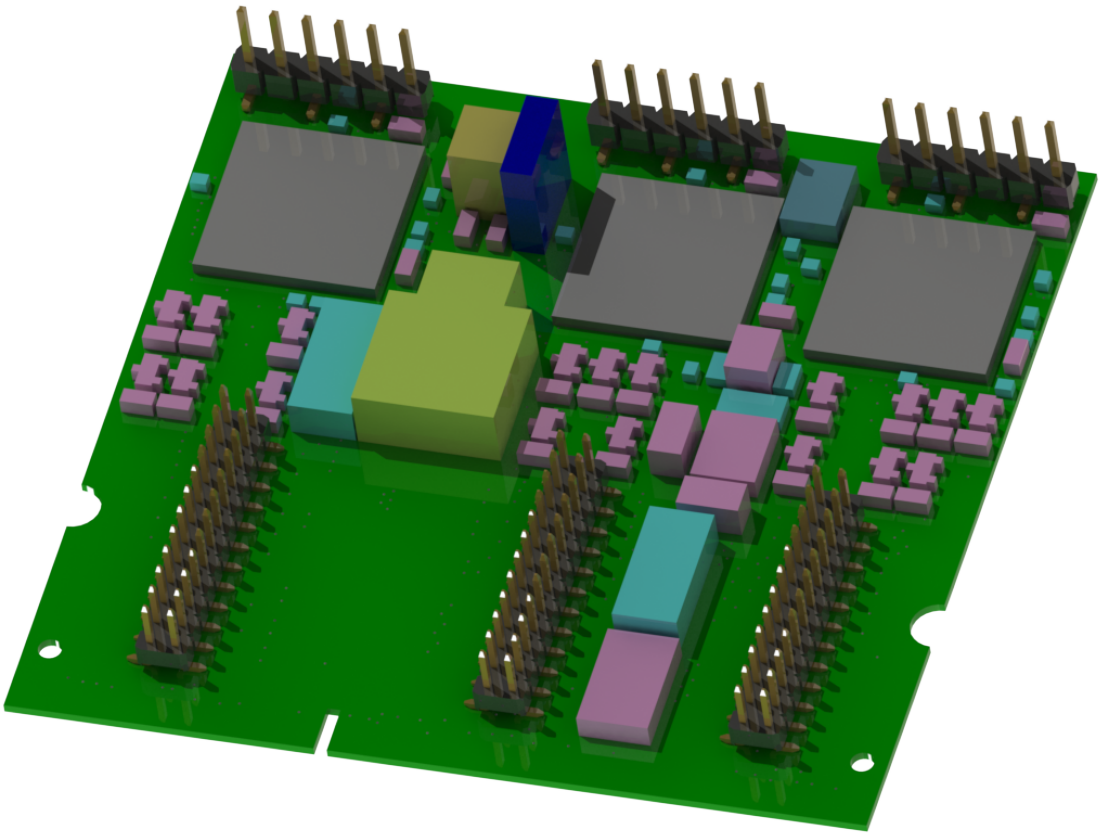


Figure 2.5.16: Angular top view of the Monitoring and Control PCB for Reticles. Computer-generated picture made with SolidWorks

2.5.3.9 Monitoring and Control PCB of Reticles (Cure)

To avoid defects resulting of voltage problems like power-shorts or over-voltages all the voltages of all 48 reticles on the Wafer are monitored. In case of a problem the affected voltage on the affected Reticle can be switched off individually (see section 2.9.2.4). Each Cure will monitor and control the voltage supply of 6 Reticles. This task is done stand alone by micro-controllers on the Cure in order to avoid latency and non reacting due to I2C connection problems. The Cures are controlled by the MaCU. In fig. 2.5.16 the top side of the Cure is shown. At the front side the cut-outs for the board edge connector which connects the board to the sodimm-connectors on the MainPCB are visible. The pads are not shown in the picture. At the back side the programming connectors for the micro-controllers in front of them are placed. The other pin headers which are placed vertically allow a direct connection to the voltage measuring points of the belonging Reticles (see fig. 2.5.18).

The pin-out of these MainPCB to Cure PCB connectors connectors is shown in figure fig. 2.5.17

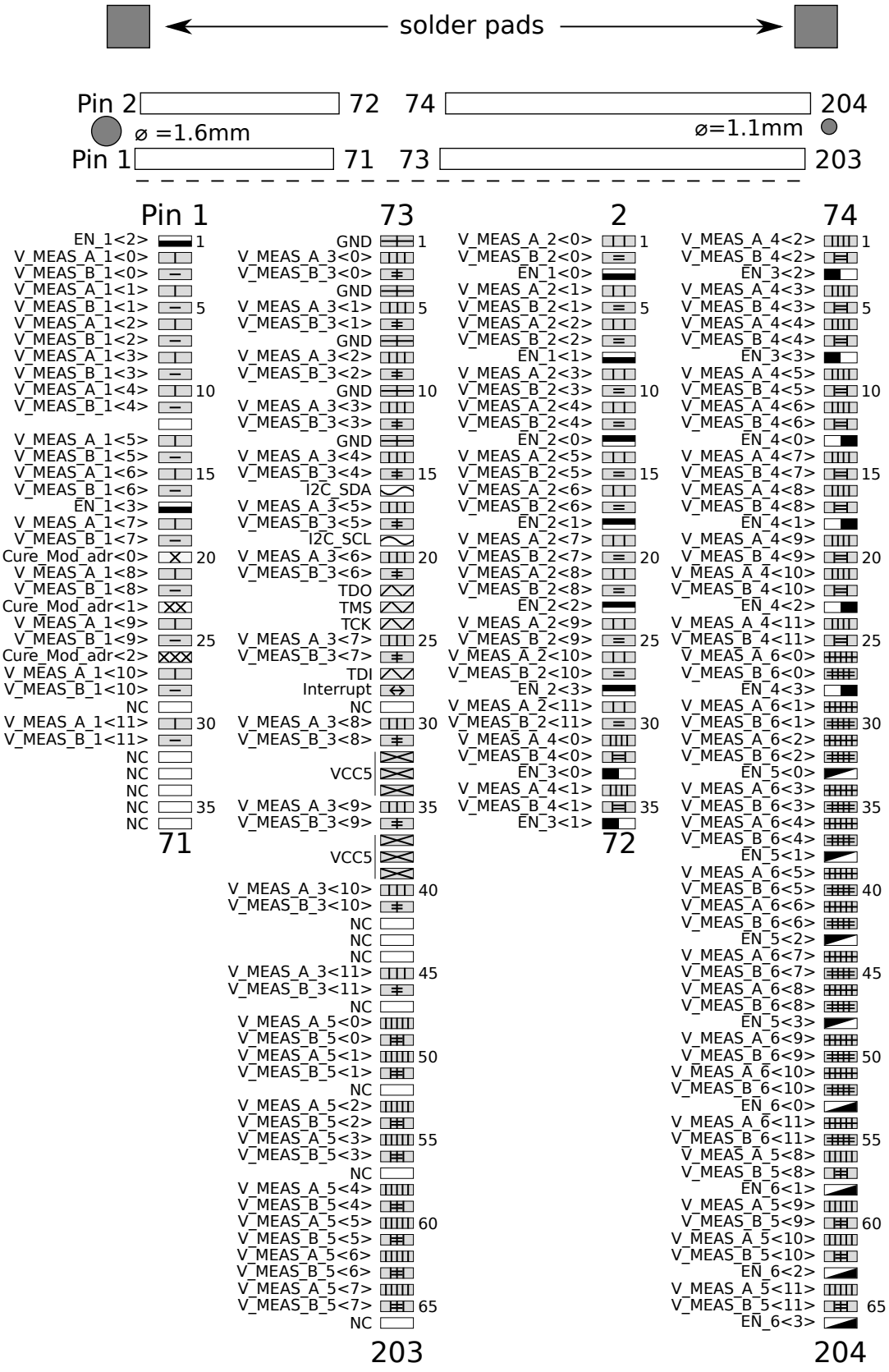


Figure 2.5.17: Pinout of the MainPCB to Cure PCB connector connector

EN_SIG...<X>		V_MEAS_B...<X>			EN_SIG...<X>		
5930	3	VDD_12	6	5	VDD5	3	5930
A912	0	DI..33ANA	7	4	VDD25	1	A912
7234	1	VDDBUS	8	3	VDD	2	7234
A912	2	VOL	9	2	DI_VCC	0	7234
A912	2	VOH	10	1	VDDA	1	7234
A912	0	D_VBIAS	11	0	DI..ANA	0	A912
A912	0	D_VBIAS	11	0	DI..ANA	0	A912
A912	2	VOH	10	1	VDDA	1	7234
A912	2	VOL	9	2	DI_VCC	0	7234
7234	1	VDDBUS	8	3	VDD	2	7234
A912	0	DI..33ANA	7	4	VDD25	1	A912
5930	3	VDD_12	6	5	VDD5	3	5930
MOSFET		corresponding V_MEAS_A					MOSFET

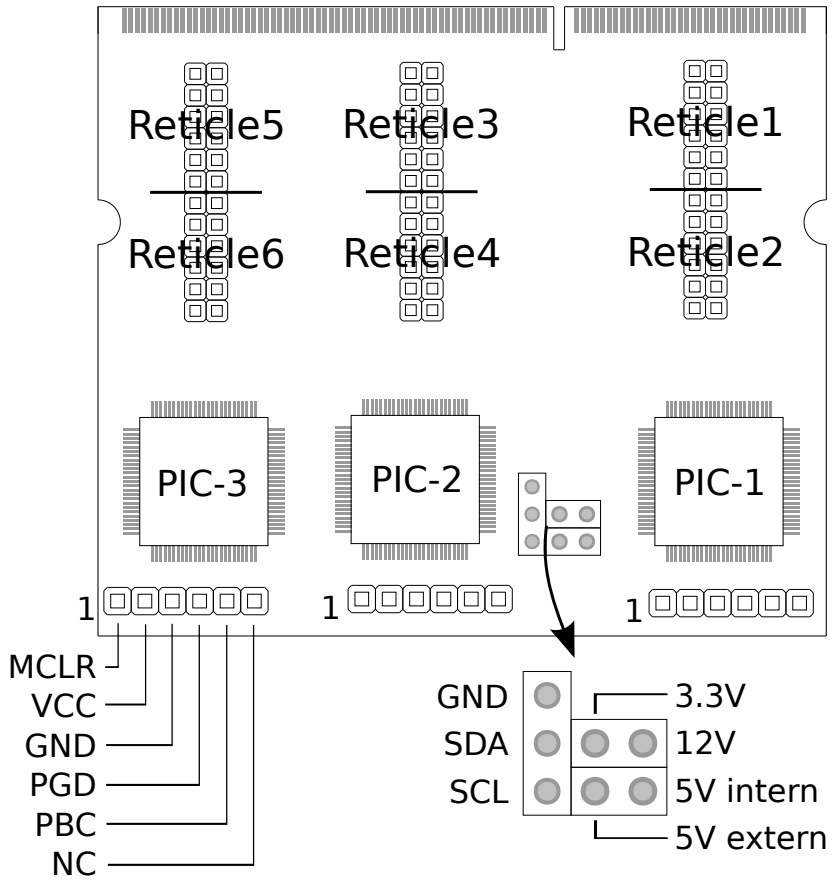


Figure 2.5.18: Pinout of the connectors on the Cure board

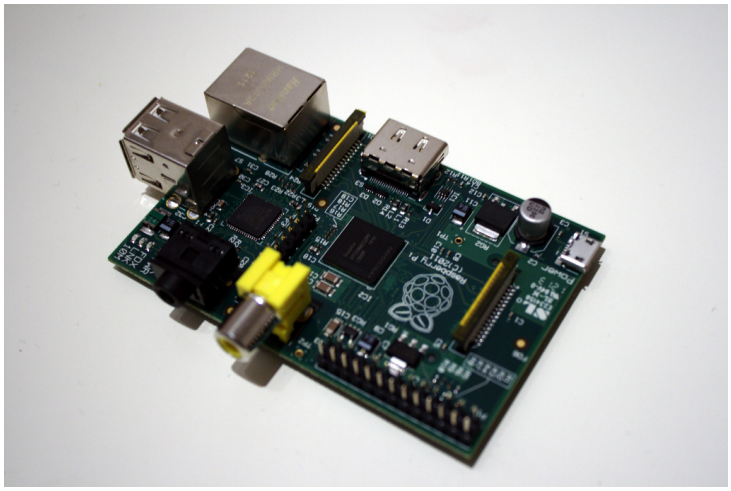


Figure 2.5.19: Top side of the Raspberry Pi

2.5.3.10 Main System Control Unit (MaCU)

The Main System Control Unit is realized by a standard Raspberry Pi [58] as control computer of the Wafer Module. It allows full system control via one Ethernet link. The Raspberry Pi (Raspberry Pi) communicates using I2C with the control units placed on the power supplies and the Monitoring and Control PCB for Reticless. It also collects all the temperature information of the system. The Raspberry Pi is connected using a ribbon cable to the AnaB on the right bottom side of the MainPCB. More detailed information is given in section 2.9.2.3.

2.5.3.11 Top Cover (ToCo)

Top and bottom view of the Top Cover is shown in fig. 2.5.21. The ToCo is milled out of a 20 mm thick planarized aluminum plate with a hight of 440 mm and a width of 440 mm. The inner structure is formed by an equilateral octagon which is merged over the diagonals to the outer frame. The distance between two sides of this octagon is 242 mm.

The ToCo does not only increase the stability of the system but is also used to fix peripheral electronic units described in section 2.5.3.5. In the bottom view of the ToCo a pattern of stamps in the middle of the octagon is shown. This stamps have three important functions.

- The first function is the constant distribution of the back pressure from the Wafer over the Elastomeric Stripe Connectors onto the MainPCB over the wafer-area.
- As the second function they work as a heat conductor. Because of its size the solid ToCo can absorb a large amount of heat and because of its high surface area it can be cooled easily.
- The third function of the stamps is their function as a current sink. Both the ToCo and the WBr are connected to electrical ground. To relieve the ground planes inside the MainPCB the stamps of the ToCo are electrical connected to the ground pads upon

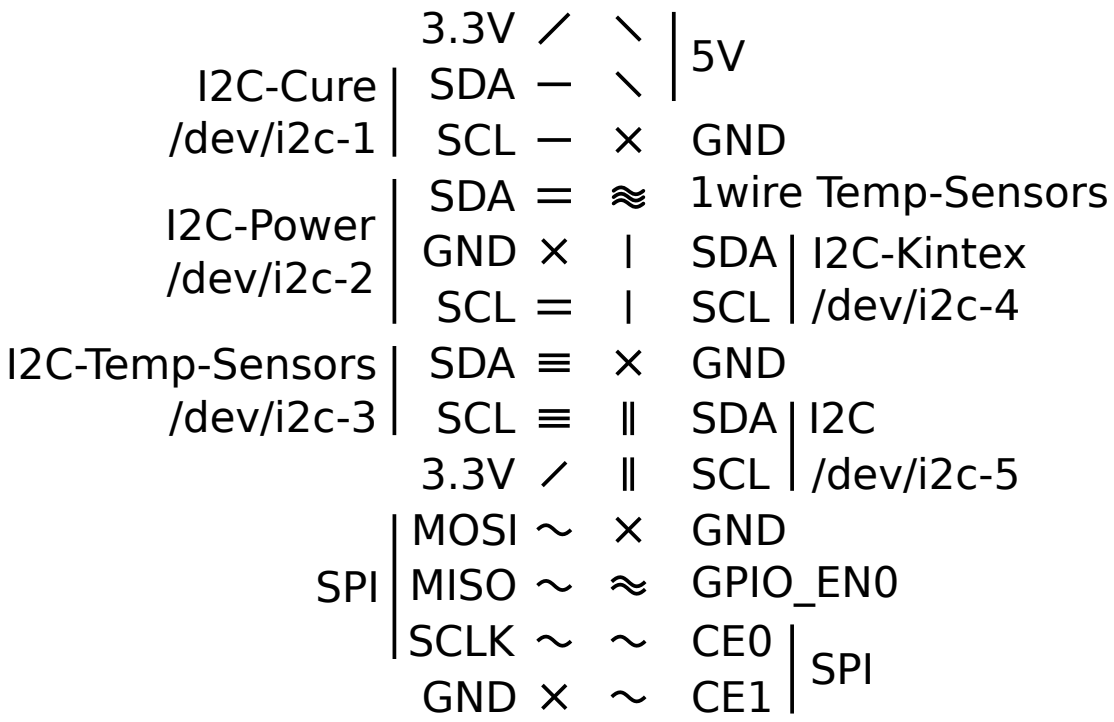


Figure 2.5.20: Pinout of the AnaB to Rasperry Pi connector

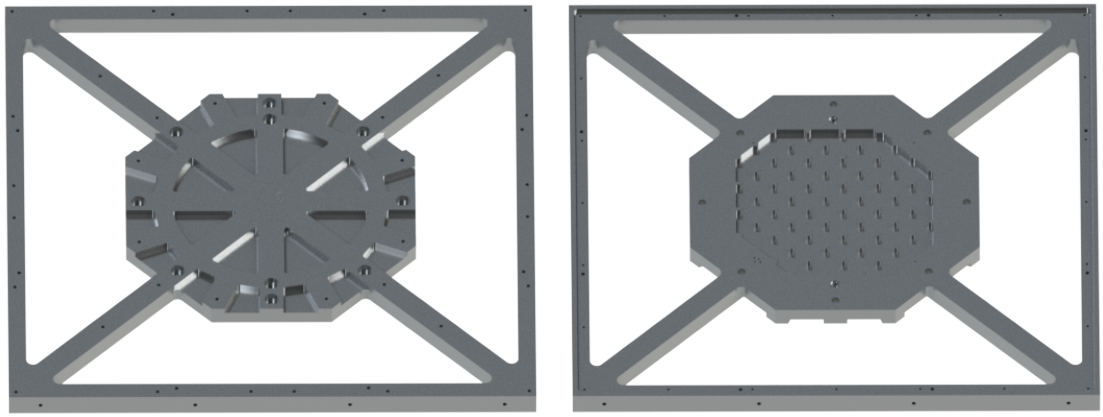


Figure 2.5.21: Top view of the Top Cover

the MainPCB. To achieve a good thermal and electrical conduction the ToCo is electro silvered.

2.5.3.12 Insertion Frame for mounting of additional PCBs (InFra)

The InFra is screwed to the ToCo in order to allow the insertion of the assembled Wafer Module into the rack. It is also used to mount the Wafer I/O PCBs. Figure 2.5.22 shows a computer-generated picture of the InFra. Mechanical details such as screw hole positions can be found in appendix A.

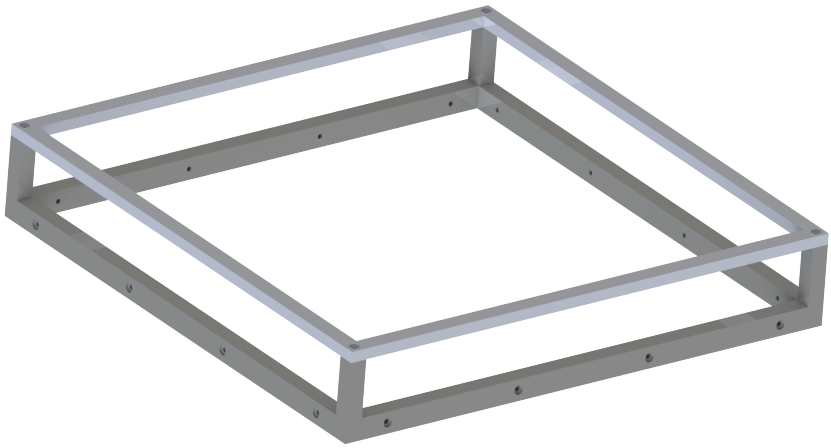


Figure 2.5.22: Angular view of the Insertion frame for mounting of additional PCBs

2.5.3.13 FPGA Communication PCB (FCP)

The configuration, the monitoring and the pulse stimulation of the HICANNs on the Wafer is done on daughter boards. These boards are called FCP and they are placed vertically

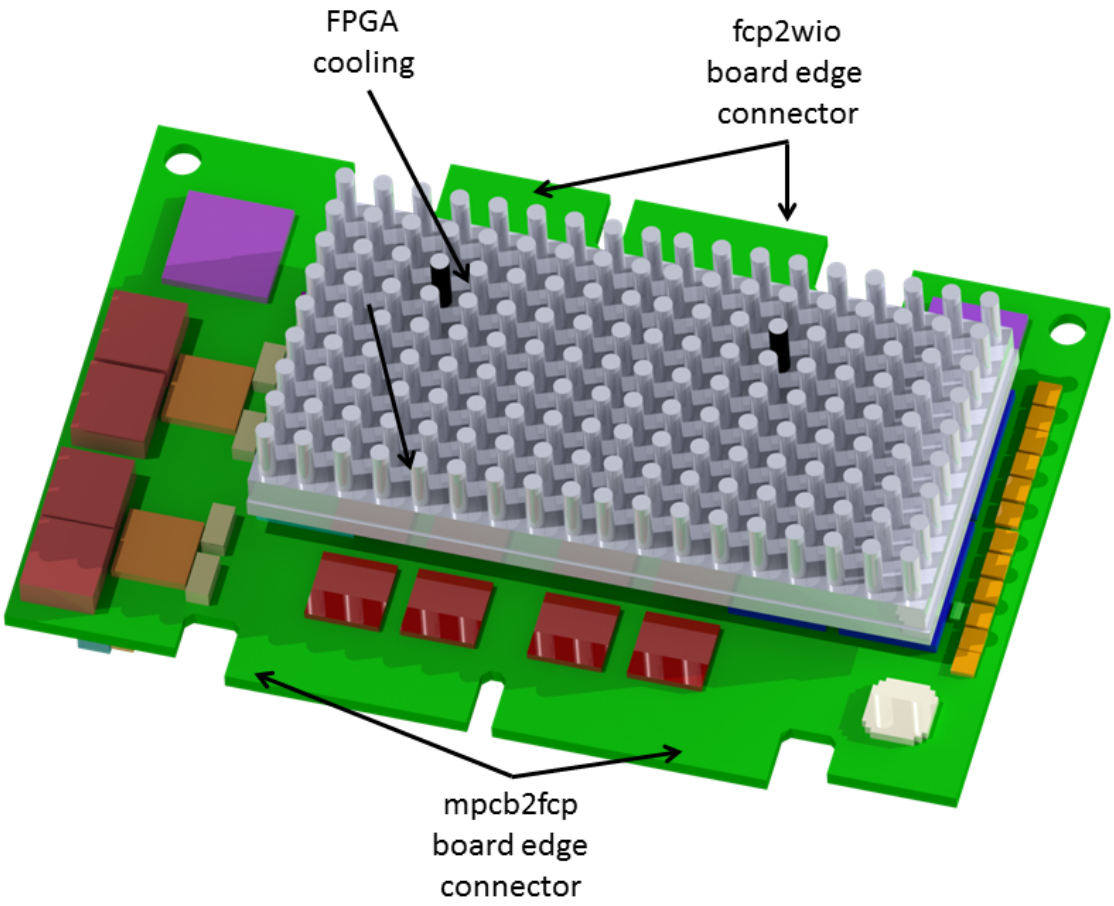


Figure 2.5.23: Angular top view of the FPGA Communication PCB. Computer-generated picture made with SolidWorks

on the MainPCB (see fig. 2.5.11). Each of these FCPs is responsible for 8 HICANNs that are placed on one Reticle. This sums up to 48 board whick are placed on the bottom side of the MainPCB. In fig. 2.5.11 the 48 connectors placed on the bottom side of the MainPCB are visible. Figure 2.5.23 shows a computer-generated picture of the top side of the FPGA Communication PCB. More details and a picture of the prototype can be found in section 2.6.2.1.

2.5.3.14 Wafer I/O PCB (WIO)

For the data exchange between the FCPs on the same or on different Wafer Modules the FCPs are requiring additional components. These components such as Gbit Ethernet PHYs and RJ45 Gigabit and USB3.0 connectors are placed on larger PCBs placed on top of the FCPs. Each of this so called Wafer I/O PCBs will provide the necessary components for 12 FCPs. This sums up to 4 WIOs for one Wafer Module. The necessity of cooling the current consuming components (specially the FPGAs) on the FCPs restricts the placement options of the FCPs on the MainPCB to one direction. This is the vertical upwards direction because the airflow in the Wafer Module is from bottom to top. Therefore two different types of WIOs have to

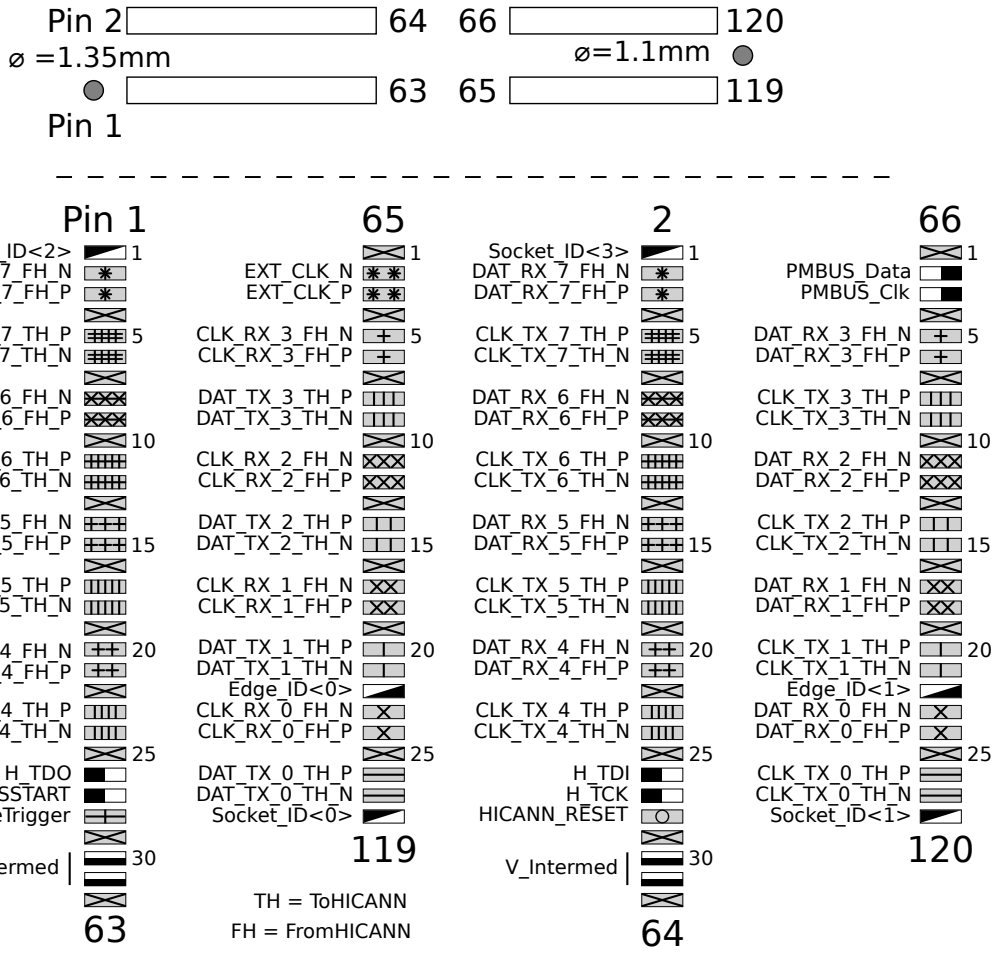


Figure 2.5.24: Pinout of the MainPCB to FCP connector connector

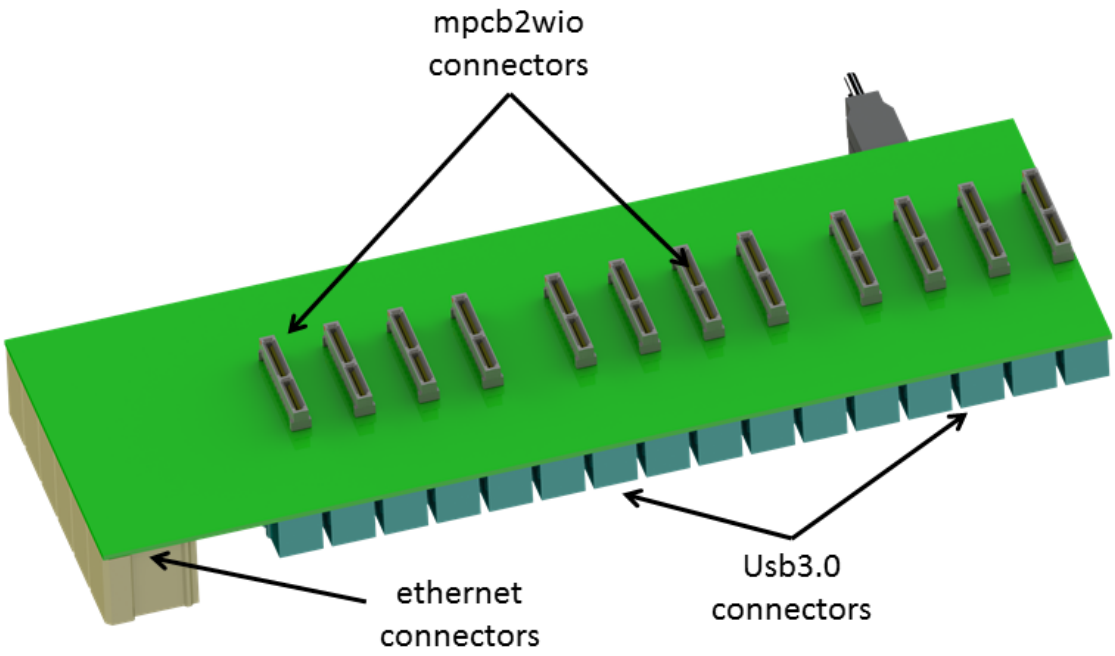


Figure 2.5.25: Computer-generated picture of the bottom side of the WIOH

be built which are called WIOH and WIOV referring their direction into the system. Detailed description about these PCBs can be found in section 2.6.2.2.

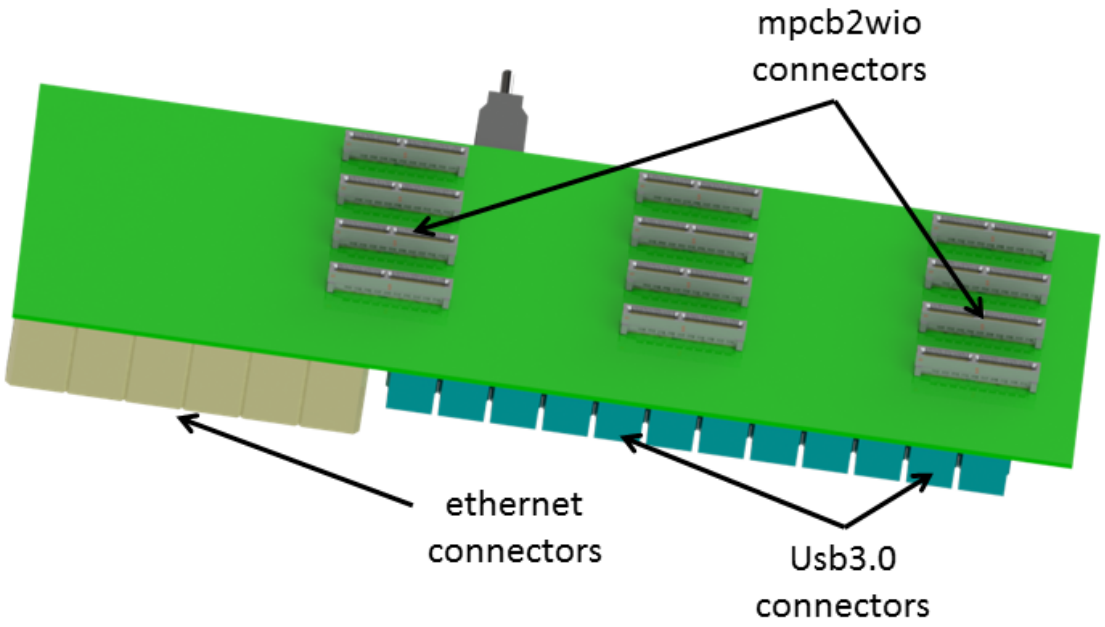


Figure 2.5.26: Computer-generated picture of the bottom side of the WIOV

2.6 Communication Modules

2.6.1 Overview

Configuration of the HICANNs, pulse stimulation and monitoring, and communication between wafers is done on FCPs, see Fig. 2.1.1. These boards contain a Xilinx Kintex7 FPGA (Part no. XC7K160T-1FFG676C) that executes the above tasks [74].

The interfaces of the FCP are shown in Fig. 2.6.1. One FCP is connected to eight HICANNs on the wafer. Communication is done via one LVDS interface per HICANN, which is a custom design on the HICANN and uses LVDS pins with standard serializer/deserializer modules on the FPGA side. The interface to the host uses one Gbit-Ethernet port. Seven Xilinx Gigabit Transceiver protocol (GTX) transceivers are available for pulse communication with other FCPs. From these, three links are used for connecting other FCPs on the same wafer module. The FCPs are arranged in groups of four, and the local inter-FCP links are used for realizing full connectivity between the FCPs in one group. The remaining inter-FCP links are employed for inter-wafer pulse communication.

Besides the communication interfaces, the FCP contains three Double Data Rate (DDR)3 memory interfaces. One of them is used as frame buffer for Ethernet communication, the other two are employed for pulse storage. One of the pulse memories is used as playback memory, storing pulses for stimulation. The other one is used as trace memory, storing pulse activity during an experiment, which can be read out afterwards by the host.

The following sections describe the design of the FCP and the firmware development for

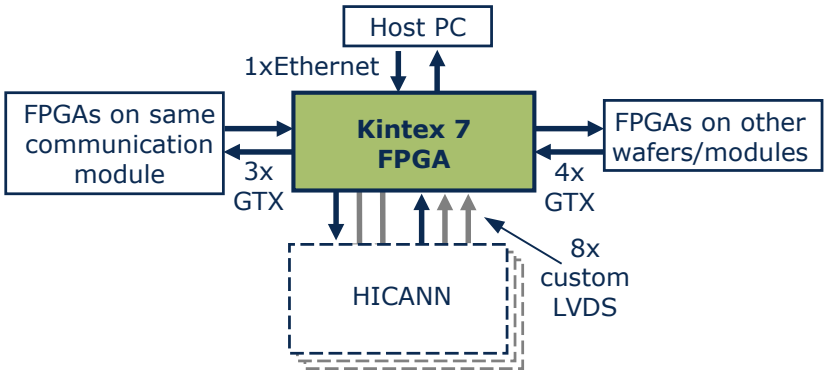


Figure 2.6.1: Communication channels and partners of the Kintex7 FPGA

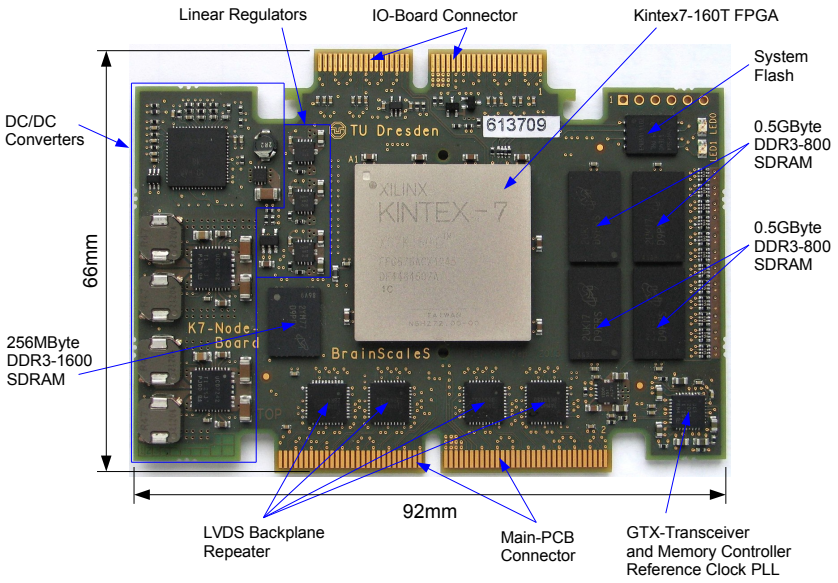


Figure 2.6.2: Photograph of the FCP.

the Kintex7 FPGA.

2.6.2 Board Design

2.6.2.1 Kintex7 board

Each FPGA board, as shown in Fig. 2.6.2 includes a Xilinx Kintex 7 FPGA as its core component, which is responsible for interfacing to eight HICANNs on the wafer, communication to the host via Gbit Ethernet and transmission of pulses to other FPGA boards via seven GTX transceivers. Furthermore, it provides external storage for Ethernet frames, stimulation pulses and pulse tracing. To achieve the maximum possible throughput the storage tasks are separated onto three independent DDR3 memories.

The board features the following components:

- Kintex7 FPGA speed grade 1
- 512MB DDR3-800 SDRAM for playback data (two 2Gbit chips, 32bit interface, 400MHz)
- 512MB DDR3-800 SDRAM for trace data (two 2Gbit chips, 32bit interface, 400MHz)
- 256MB DDR3-1600 SDRAM for Ethernet (one 2Gbit chip, 8bit interface, 800MHz)
- 32MB SPI Flash memory for multiple FPGA firmware images
- Card edge connector to Wafer-IO-Board with 80 pins (J1)
- 8 GTX transceiver connections via J1 used for:

  - Serial Gigabit Media Independent Interface (SGMII) link to Gigabit Ethernet PHY (1.25Gbit)
  - three connections to adjacent K7-Node-Boards (6.25 Gbit)
  - four links to USB3.0 connectors high-speed signal pair for inter-wafer communication hosted by Wafer-IO-Board (6.25 Gbit)
- Card edge connector to Main-PCB with 120 pins (J2)
- 8 HICANN connections via J2 buffered by back-plane LVDS transmitters with optional +6dB pre-emphasis
- On-board power supply with an input voltage of 6-13V
- Board power control by PMBus commands (default is power OFF)
- PMBus access to board status (voltages, PGOOD signals, currents and temperatures)
- Power Good and Done LED
- Socket-ID[3:0], Edge-ID[1:0] and PMBus addresses defined by slot on Main-PCB
- Wafer-ID[7:0] is provided via PMBus command
- Reference clock PLL for GTX transceivers (125MHz) and memory controllers (200MHz)
- 3.3V debug/GPIO header
- Two debug LEDs
- On-board 50MHz system reference clock generation for stand alone test (without Wafer-IO-Board) of HICANN connections
- Wired OR sample trigger signal generation

PMBus groups, Socket- and Edge-IDs Figure 2.6.3 depicts the Wafer-Module from the bottom side, showing the PMBus groups and the Edge- and Socket-ID predefinition. Within each PMBus group, the addresses of the UCD9246 power controllers and FPGAs are made unique, by determining them from Socket-ID bits [1:0]. All the identification bits together up to the Wafer-ID are used to calculate system wide unique MAC and IP addresses.

Power-up The FCP by default is switched off after the input supply voltage rises above 6V. To enable the board, all OPERATION registers inside the UCD9246 digital power controller must be written by a broadcast command. This causes the digital power controller to synchronously ramp up the four DC/DC converter controlled supply domains. The downstream placed linear regulators will ramp up the analog domains too, right after their internal power up circuit detects high enough supply voltage for proper operation. If all DC/DC domains are regulated within their limits the UCD9246 generates it's power good signal which causes the FPGA to load the firmware image from the 32MB SPI Flash. If a valid image was loaded the Done LED

Figure 2.6.3: PMbus group, Edge- and Socket-ID predefinition.

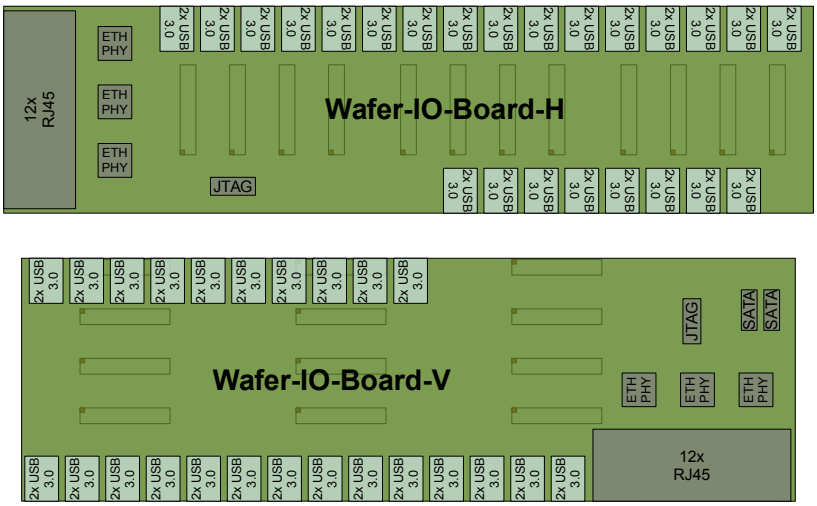


Figure 2.6.4: Board layout of the wafer IO boards.

lights up and the FPGA firmware starts operation. Towards this point the FPGA is reachable via PMBus, showing that it's up. The next step is to read out the FPGA status register (on-board PLL configured and locked) using the PMBus and writing the Wafer-ID into the corresponding register. After a valid Wafer-ID is detected by the FPGA firmware it calculates the MAC and IP address and initializes the Ethernet PHY. This process finalizes the start-up sequence.

2.6.2.2 Wafer IO boards

The Wafer-IO-Boards are designed to host all IO connectors of the 12 FCPs placed on one edge of the Wafer-Module. In addition to the connectors the three quad Gbit Ethernet PHYs are placed on the board too, supporting one SGMII MAC connection for each FCP.

The IO boards host the following components:

- 12 RJ45 Gigabit Ethernet connectors with integrated magnetics integrated as one 12x dual row connector block
- Three quad SGMII Gbit Ethernet PHYs
- 48 USB 3.0 ports realized as 24 stacked double connectors (the one 3.0 high-speed signal pair is directly connected to one GTX transceiver of a FCP)
- One free FPGA programmable LED per USB 3.0 port
- 50MHz system reference clock input connector and clock tree circuitry
- SYS_START signal input connector and distribution circuitrry
- Connector and circuitry to make each FCP reachable by JTAG
- Small PMBus debug header

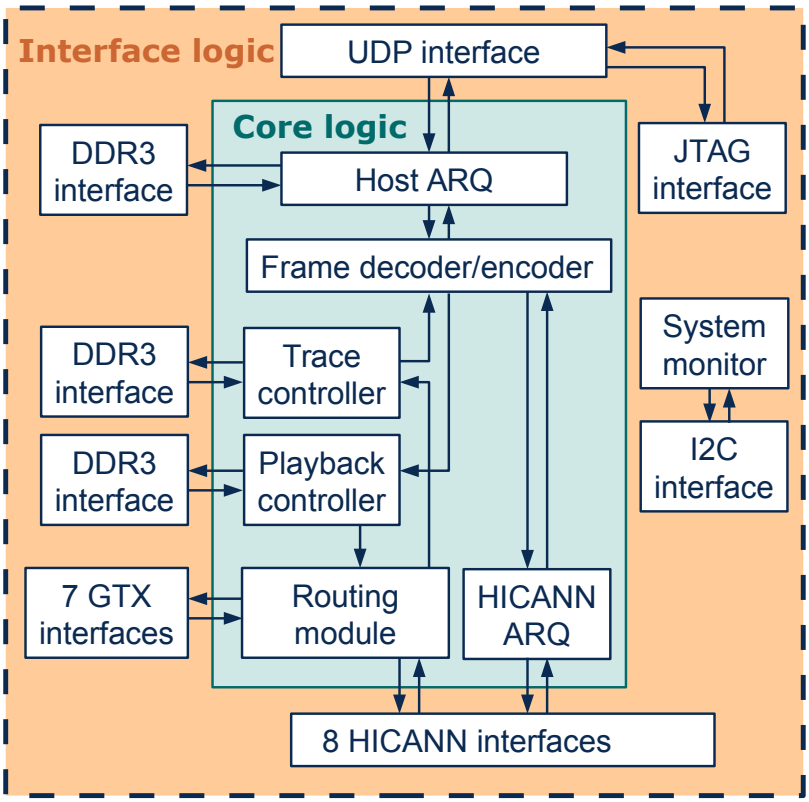


Figure 2.6.5: Main modules of the Kintex7 FPGA firmware

- On-board power supply for all active components with an input voltage of 6-13V

2.6.3 FPGA Firmware

2.6.3.1 Overview

The structure of the Kintex7 FPGA firmware is shown in Fig. 2.6.5. Its modules are split into interface logic and core logic.

The interface modules provide simple interfaces to the respective communication partners and encapsulate details of the interface control like generation of special clocks or instantiation of dedicated Xilinx Intellectual Property (IP) blocks. There are separate interfaces to the three DDR3 memories and seven separate GTX interfaces. Ethernet communication is provided by a custom designed User Datagram Protocol (UDP) interface. The eight HICANN interface modules not only provide access to the LVDS communication lines, but also contain arbitration and protocol handling for pulse and configuration packets. They also encapsulate pulse buffering and sorting functionality for pulses sent to the HICANNs, to ensure their release at a defined target time. The firmware also provides a UDP-JTAG adapter for enabling low-level access to the HICANN. A system monitor module reads out the FPGA device temperature and different supply voltages. It can be accessed from outside via a I2C

interface.

The core logic is responsible for processing and routing of the data from the different interfaces. A variant of the ARQ protocol is used for safe data transmission between host PC cluster and FPGA. Data for the host communication are handled and distributed by a frame decoder/encoder module. A trace controller manages writing of pulses to be traced in the dedicated DDR3 memory, and reading-out of the memory content to the host. The playback controller allows for release of pre-stored pulses and HICANN configuration packets from the corresponding DDR3 memory. A routing module handles inter-FPGA routing of pulses. Finally, safe transmission of HICANN configuration data is ensured by a separate module implementing another variant of the ARQ protocol.

In the following sections, the main modules are described in more detail.

2.6.3.2 Low-level interfaces

UDP interface

Ethernet communication in the Kintex7 FPGA is done via a custom-designed UDP interface. It implements the main features of UDP in a relatively light-weight design. The SGMII protocol is used to connect the FPGA to an external Ethernet physical layer chip. The UDP module was designed for Gbit-Ethernet usage, but is compatible with 100Mbit and 10Mbit modes as well.

The interface of the UDP module to the core logic is frame-based: Start-of-frame and end-of-frame signals are sent before and after the actual frame data. The receiving side does not wait when providing data of a frame. The core logic has to process it immediatly or implement a buffer. The same signals as for the receiving side are used for the interface on the sending side, but with different timing requirements. The core logic requests sending of a frame by a start-of-frame signal. In turn, the UDP module prepares the frame header. Afterwards, it provides a ready-signal, notifying that the next data word can be sent. The actual sending of the frame is initiated with the end-of-frame signal provided from the core logic.

DDR3 interfaces

Three independent memory controller interfaces have been implemented to access the external DDR3 memory chips [40]:

- 256MiB DDR3 Memory for Ethernet frame buffering
- 512MiB DDR3 Memory for playback
- 512MiB DDR3 Memory for tracing

A Memory Interface Generator (MIG) IP provided by Xilinx was used to implement each memory controller interface. The toplevel of the IP block was modified to reduce the number of hierarchy levels and to allow easy configuration for the different external memory chips with different physical interface widths. The memory interfaces provide a FIFO-like user interface instead of the Xilinx Advanced Extensible Interface (AXI) standard to maximize

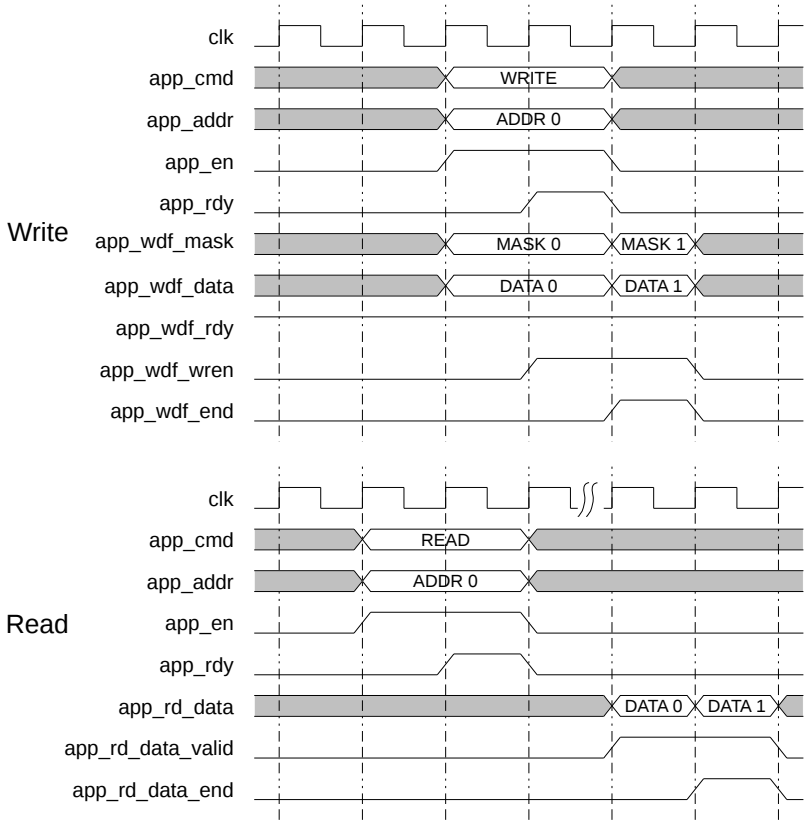


Figure 2.6.6: Address handling for the playback and trace memory interfaces

throughput. They both follow the specification in [75]. The Ethernet frame buffer uses a clock frequency ratio of 4:1, which is fully covered by this document. The other two memories use a 2:1 ratio, for which the address handling is not specified. Figure 2.6.6 shows the employed address handling scheme, which has been developed from the behavioural memory models provided by Xilinx. It has been successfully verified in simulation and tests on the FPGA.

GTX transceivers

The GTX transceiver modules are used for inter-FPGA pulse communication on one wafer and between wafers. It implements the Xilinx Aurora 8b/10b streaming interface. Currently, a raw data rate of up to 6.25Gbps is supported. The module provides a FIFO-like user interface in both directions.

System monitor

The system monitor periodically reads the FPGA device temperature. This value is provided to the DDR3 memory interfaces for internal calibration purposes. Furthermore, the FPGA device temperature can be read out via the I2C interface. The system monitor also provides

several measured voltage levels, which can also be accessed by I2C. The 13 register values to be read out are:

I2C Address	Content
0x0	Current FPGA temperature
0x1	Current FPGA Vccint
0x2	Current FPGA Vccaux
0x3	Current FPGA Vccbram
0x4	Minimum FPGA temperature
0x5	Minimum FPGA Vccint
0x6	Minimum FPGA Vccaux
0x7	Minimum FPGA Vccbram
0x8	Maximum FPGA temperature
0x9	Maximum FPGA Vccint
0xA	Maximum FPGA Vccaux
0xB	Maximum FPGA Vccbram
0xC	Current MGTVCCAUX voltage

The fixed-point conversion is already done in the FPGA. It has the format:

- Temperature: 10.6 format
- Voltage: 4.12 format

The system monitor is only operational if the initialization logic has detected a successful lock of the external PLL.

I2C interface

The I2C interface has been implemented to be SMBus compatible, including Read/Write Word and Read/Write Byte commands. It is operational shortly after initial FPGA configuration has completed. The I2C slave address is configured to 0x28 + (SOCKET_ID & 0x3) through the initialization logic. The following registers are available:

I2C Register	Content
0x00	Initialization Logic Status (16-Bit, readonly)
0x01	Wafer ID (16-Bit, RW), writable only if wafer ID locked bit is not set in Initialization Status
0x10	HICANN TX Interface Power Down (16-Bit, RW): 2 bits for each channel (Bit 0: TX Clock, Bit 1: TX Data)
0x11	HICANN TX Post-Emphasis Enable (8-Bit, RW): 1 bit for each channel
0x12	HICANN TX Equalization Enable (8-Bit, RW): Bit 0: EQ enable, Bit 1-7: Reserved, read-as-zero
0x80-0x8C	System Monitor Registers (16-Bit, readonly)

2.6.3.3 Layer 2 HICANN interface

Overview

The HICANN interface was initially designed in the DNC [61] [63]. As the current system structure merged functionalities of the DNC into the FPGA, its low-level communication components are adapted to run in the selected FPGA. The HICANN transmission control can be found 8 times in the FPGA design, one for each HICANN. It schedules the packet transmission towards the HICANN and modifies the received pulse packet events in their timestamps. A detailed block diagram can be found in figure 2.6.7. Transmission and receiving of packets and the configuration of one HICANN channel is presented in the following chapters. To ensure fast interaction with all connected HICANNs, each of these channels is completely independent from the other ones in all packet routing mechanisms.

One HICANN connection has 8 RAM cells, each with 64x24bit memory, corresponding with the L1 channels in the HICANN. So each neuron on the eight available busses has one entry in the memories. The RAM cells are dual port memories for speedup of the sorting algorithm that is used for timestamp prioritized routing [62]. The channel supports different packet types: two different event packets for different numbers of events, one communication packet for HICANN configuration and different control packets for flow control. The low level communication interface consists of two transmit LVDS [65] cells for clock and data and a serializer that generates the bitstream for transmission. On the receiving side, there are two LVDS cells, also for clock and data and a deserializer for bitstream to parallel data conversion. Each direction offers 1 Gbit/s data rate. Transmission is assured by error detecting, CRC, and optional error correction mechanism via acknowledge-resend packets.

The eight RAM cells need to be allocated to the receiving or the transmitting direction. This is done in the configuration phase and corresponds with the configuration of HICANNs L1 channels. For each L1 channel, one RAM block is available. If the FPGA transmits towards a L1 channel, the RAM block is in sorting heap mode to buffer and sort the event from the FPGA. If the FPGA receives events from a L1 channel, the corresponding RAM block is used as Look Up Table (LUT) for timestamp data that is required for time-based event routing.

Transmission towards the HICANN

The transmission towards the HICANN can be split into the handling of pulse event packets and of configuration data (ARQ packets). Both packet types need to be buffered in FIFO memories at the input, to allow reordering via scheduling mechanisms. The reordering is done using a custom developed heap sorting memory (see [62]). The target L1 channel is determined and the corresponding heap memory is prioritized selected. The memory orders all events in a heap tree structure so the top element always contains the next closest event in time. The top elements of all down-way configured heap memories are compared with the current system time. The event is released to the HICANN, if the difference of both is smaller than a preconfigured limit value. This combination of buffering and scheduling reduces the demand on buffering capabilities on the Wafer, where the focus is on analog neural structures. The limit for releasing can be set in increments of 32 clock cycles. This setting must account for the link delay down to the Hicann, which is dependent on the link traffic, i.e. on the number of neuron-neuron connections routed via this HICANN channel.

Configuration ARQ packets are not modified in their contents or controlled via a timestamp mechanism. They are forwarded to the HICANN immediately when the link is idle (i.e. no event packet is scheduled for transmission). So pulse events are privileged in submission before configuration ARQ packets to ensure timing requirements for neural connections.

Receiving from the HICANN

Incoming packets from the HICANN are immediately forwarded without further processing, reordering, etc. The event packets arriving from the HICANN contain the timestamp of the release time and a 9 bit wide identifier for the source neuron. The further packet handling is done with source-based addresses routing and target-based prioritization via the timestamp. The packet is forwarded towards the target by using the source identifier. At each communication point a LUT is used to generate one packet for each required connection. So the bandwidth requirement is reduced as the new packets are generated as late as possible depending on the source address. The time-based prioritization uses the target timestamp. Each packet contains the timestamp of the next closest release time. The sorting of the pulse events prioritizes the next required packets. When new packet copies are generated at crossing points, the new packet gets a modified timestamp with the next target release time. As one packet always contains the next release timestamp in time for all possible target points, the new release time is generated by simply adding a delta time to the existing time.

For further routing, when a event from the HICANN is received, containing the occurrence time, the timestamp of the release time at the next target in time is required. Each neuron-synapse connection is defined with a constant duration for transmission. The smallest duration of all target paths is added to the received timestamp. The value is contained in the routing memory of the L1 channel that has offered this event on the HICANN. The LUT uses the source neuron identifier. After correction of the timestamp the event is forwarded. If two event packets arrive at the same time, the second one is delayed until the first packet is modified and then processed itself. Further routing of the event packets can be done in the FPGA towards the targets via the source neuron id and prioritized by the timestamp via the calculated target timestamp as described before. Configuration ARQ packets are simply forwarded towards the FPGA without modification.

Configuration of a HICANN channel

For proper operation of routing mechanisms, the HICANN channel needs to be configured. One of the most important configuration registers is the one for the direction of the L1 channels. It defines whether a L1 channel of the HICANN is configured for sending towards the FPGA or for receiving data from the FPGA. Depending on this, the eight RAM cells need to be assigned to their functionality, either the heap sorted buffer for sorting events and sending them to FPGA, or the LUT for modifying the timestamp of a received event. In the transmission section of the channel, for the sorted events the start time towards the HICANN is defined by the difference between the event timestamp and the current system time. If this difference goes below a configurable limit, the event is released. The limit is set via a configuration register. The last required information before the experiment can start is the content of the look up memories for modifying the event occurrence time into the release

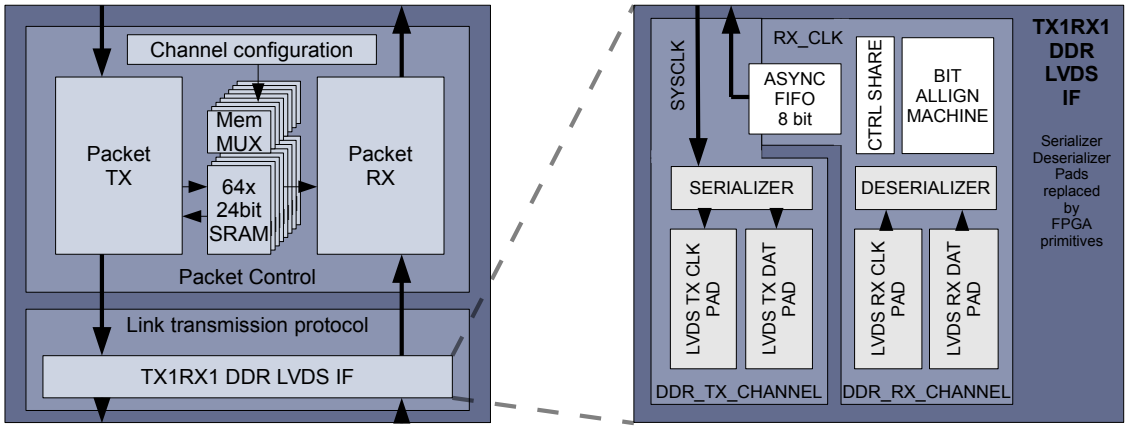


Figure 2.6.7: Block diagram of the L2 HICANN connection in the FPGA

time at the target HICANN.

All configuration data is transmitted by the host control to the FPGA. Each HICANN channel requires separate configuration data. For debug and testing purposes, all configuration data can also be set using the JTAG [1] interface. The implemented JTAG registers are presented in table 2.10.34.

2.6.3.4 Core logic

As shown in Fig. 2.6.5, the core logic contains all modules that are not directly related to the FPGA interfaces. They are divided into the Host ARQ protocol (HostARQ) module, which implements the ARQ protocol for safe date transmission between host and FPGA, and the remaining modules, which constitute the Application Layer for the host-FPGA communication.

Playback module

The playback module decodes the content of the playback memory and releases the stored pulses and configuration packets to the HICANN interfaces at the defined release times. It also writes the playback memory entries directly from the host interface to the playback memory, using the format defined in section 2.10.2, but with frame type header stripped. The playback memory is read block-wise with maximum burst size to optimize throughput. The FPGA release time of the next pulse group is continuously compared to the 15bit system time counter that is continuously running in the FPGA. Release of pulses is started when the release time is reached. The, pulses in the pulse group are sent to the addressed HICANN interface, one per clock cycle. In the current implementation, the playback module needs 6 clock cycles for reading and decoding of a pulse group with one pulse. This constitutes the minimum distance between FPGA release times. Long inter-spike intervals are handled via special waiting entries. Upon reading of such an entry, the playback module waits for a number of system time counter periods, defined in the entry.

After an experiment start trigger is received, the playback module waits for one overflow of the system time counter before releasing any pulses. By this, the starting time of the experiment is made reproducible. After releasing the last spike, the module automatically

goes into its idle state, being ready for the next experiment.

Trace module

The trace module is responsible for writing received spikes from the HICANNs to the trace memory, and sending the trace memory content to the host upon request. The module only starts writing pulses when it receives a start signal from the playback module, which the latter sends after it has waited for an initial overflow of the system time counter. By this, both playback and trace modules start operating synchronously after an experiment start. The trace module can receive four pulses in one clock cycle. An internal arbitration logic combines the incoming pulses to memory entries. One 64bit entry can be written to the memory in each clock cycle, which stores two pulses. For correctly handling spike intervals longer than one system time counter period, the trace module writes a special control word to the memory each time the system time counter overflows. Tracing of pulses is stopped via a control packet from the host.

2.6.3.5 HICANN ARQ

The FCP connects via a single DNC Interface to up to 8 HICANNs. All ARQ links are encapsulated in a module called the dnc_arq. It resides between the Application Layer (AL) and the DNC Interface. Figure 2.6.8 shows the top level view and the word formats on both sides. The dnc_arq is capable of receiving and transmitting one configuration word per clock cycle in both directions assuming sufficient network bandwidths. The additional header information at the DNC interface for the transport layer functionality is added and processed internally which explains the difference in the interface widths.

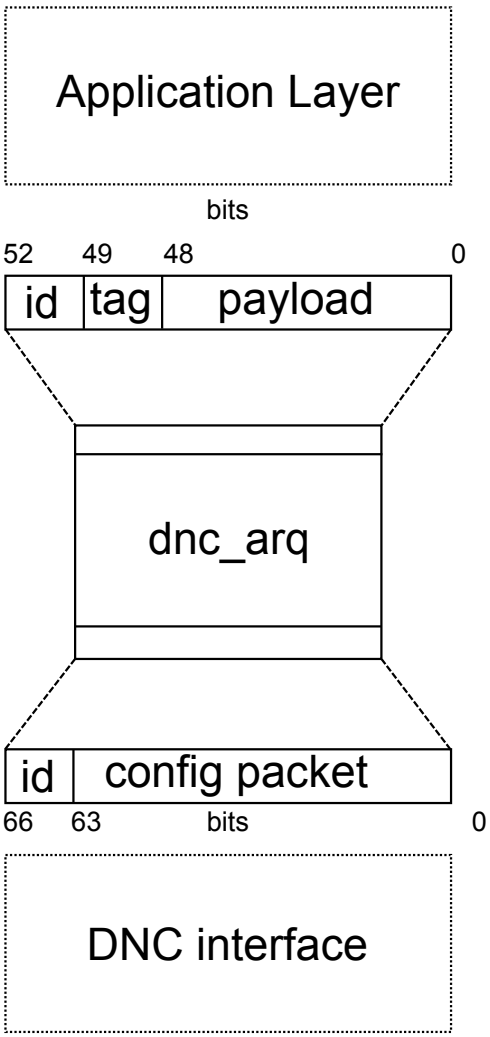


Figure 2.6.8: Top level view of the dnc_arq module acting as a buffer between the AL and DNC Interface in the FPGA of the FCP. The payload and tag fields at the AL side are encapsulated in the configuration packet, the ID field denotes which HICANN the packet belongs to.

2.6.3.6 HostARQ

The top level module host_arq_top implements the Transport Layer functionality between host PC and FCP over GbE. Subsection 2.10.1.1 gives a high-level overview of the protocol, a more detailed description can be found in [42]. Figure 2.6.9 shows a block schematic with the neighboring modules. The module serves as a bridge between the AL and the UDP layer providing full duplex data transfers which look as FIFO-like as possible to the AL. It runs at a single clock frequency of 125MHz synchronously to the UDP and AL modules. For the detailed interfaces and timing diagrams see Figure 2.6.3.6. Sending and receiving data is handled separately in two submodules called rx_link and tx_link with minimal non-

blocking communication between them to ensure independent functionality and thus full-duplex capability.

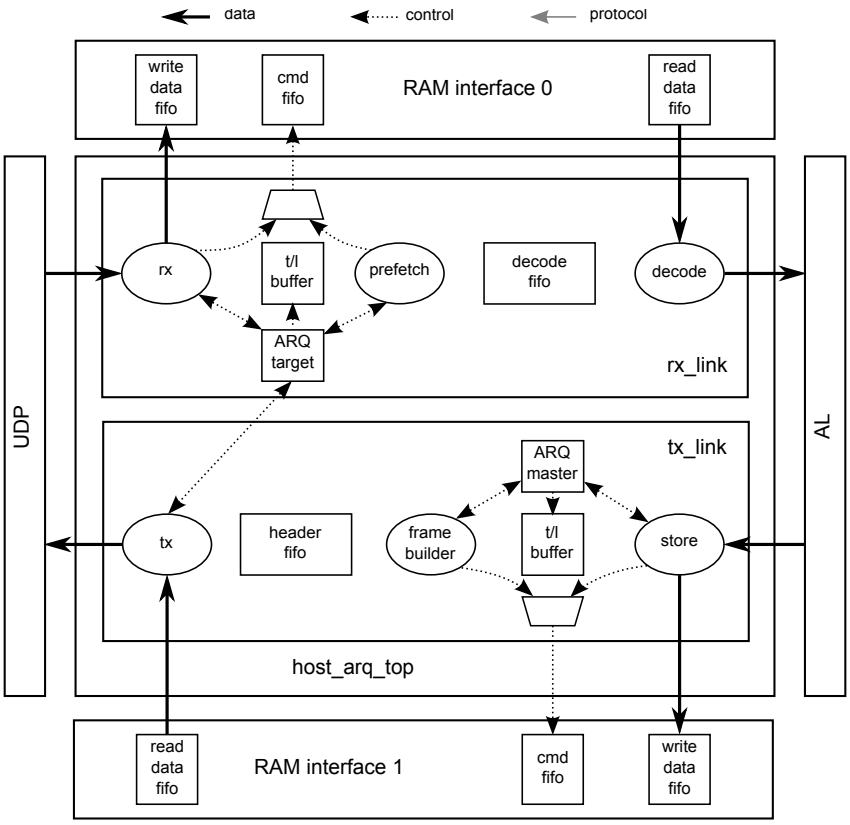


Figure 2.6.9: Structure of the host_arq module in context of its neighboring modules.

DRAM frame storage The ARQ window is implemented in the DDR3 DRAM available on the FCP. This enables configurable buffer sizes in the Megabyte range which relaxes timing requirements for the software. The tx_link and rx_link submodules manage their own memory spaces, that hold their respective windows, which they access via dedicated Xilinx MPMC ports. The DRAM access times dominate the protocol latency in the FPGA due to a prefetching scheme in the module implementation.

Interfaces

UDP rx side The start and end of a packet is announced via separate flags with data being marked as valid by raising a valid flag. Note that in the current implementation of the UDP layer the frame will be read out as a 32 bit word every two clock cycles which yields an effective bandwidth of 2 Gbit/s.

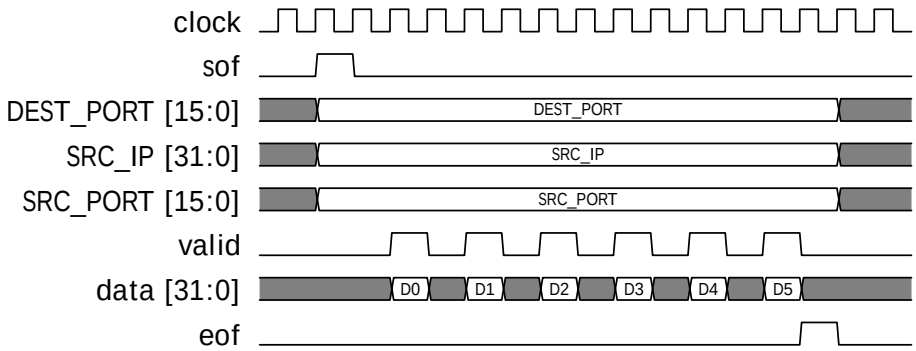


Figure 2.6.10: Timing diagram depicting the rx interface of the UDP module. The shown transmission is consistent with an ARQ packet that carries a single 64-bit payload word

Application Layer read interface The AL-read interface has been designed to look as FIFO-like as possible. The ARQ implements full flow control which means that the AL can stall for arbitrary periods of time between popping payload without loss of data. The decoder FSM shown in the rx_link module in Figure 2.6.9 takes care of pairing the payload words with their corresponding type. Figure 2.6.11 shows some examples for reading data to the AL. Signals with the suffix *_i denote input signals from the rx_link's point of view, signals marked with *_o are output signals driven by rx_link.

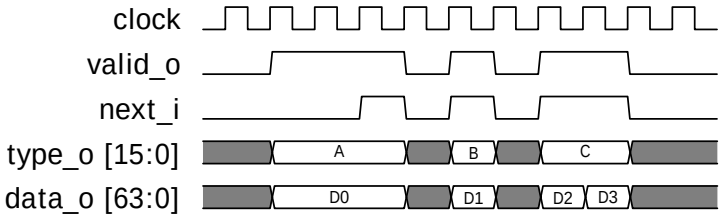


Figure 2.6.11: Timing diagram showing several examples of reading data to the AL from rx_link's point of view. First, a single word transaction with type *A* where the AL responds after a delay. Second, a single word transaction with type *B* where the AL acknowledges the data in the same clock period. Lastly, a two-word back-to-back transfer with type *C*

Application Layer write interface Writing data to the tx_link is very similar to the read interface as discussed earlier. A data word is presented to the tx_link by the AL together with its type by raising the al_write_valid flag. The tx_link will acknowledge the reception of that word by raising the al_write_next flag which completes the handshake. This handshake is necessary because it depends on many factors whether a data word can be stored in memory at a particular time. Figure 2.6.12 shows some example writes.

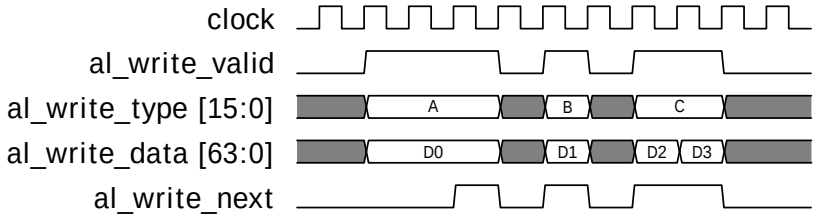


Figure 2.6.12: Timing diagram showing several examples of writing data to the tx_link. Depending on the situation data can be written within a single clock, back to back or after a certain delay.

The UDP tx interface Transmitting a frame through the UDP requires two handshakes where the UDP acknowledges both with the ready flag. The first handshake represents the request to send a frame which is indicated through raising the sof flag. After the UDP responded by raising the ready flag the frame data can now be pushed into the UDP with the push flag. To end a frame the tx_link raises the sof flag after the last data word has been pushed. Note that the current implementation of the UDP module buffers the frame entirely before it is sent to the Ethernet MAC, thus there are no timing requirements on the length of the pause between two pushes of data. This is useful because while the ARQ header data is quickly available to be pushed into the UDP there is a significant delay before the payload data is returned from memory. However, the tx FSM can be modified to adhere to more strict timings which would require a maximum delay between two pushes if necessary. An example timing diagram is shown in Figure 2.6.13.

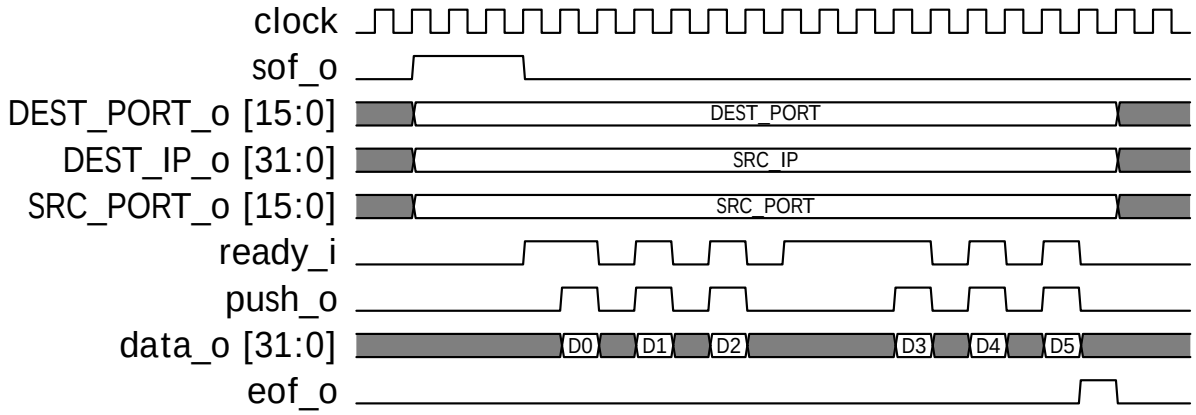


Figure 2.6.13: Timing diagram demonstrating the interface between the tx_link and the UDP module.

2.7 Analog Read-Out

The Analog Readout Modules (AnaRMs) are used to sample the membrane voltages of select neurons on the Wafer Module. Every reticle on a Wafer Module has two analog outputs (c.f. 2.4.2). A Wafer with 48 reticles thus provides a maximum of 96 membrane voltage signals. Each AnaRM has eight inputs, which are multiplexed to the input of its ADC. In the NM-PM1 each rack with four Wafer Modules will contain one analog readout subsystem with 12 AnaRMs. This allows to connect all analog outputs from one Wafer Module to an AnaRM input and sample 12 voltages simultaneously. The input multiplexers on the AnaRM support a switching frequency of 20MHz. The analog bandwith of the AnaRM is greater than 300MHz.

Cables run from every Wafer Module to the analog readout subsystem. The inputs of the 12 AnaRMs can be freely configured by plugging in the appropriate cables.

The Flyspi is described in section 2.7.1 and the AnaFP is described in section 2.7.2.

The follwing enumeration lists all components involved in the analog read-out system:

Analog Readout Module (AnaRM) 12 AnaRMs are connected to the Wafer Modules mounted in the same rack.

Control Computer Intel NUC based Linux system provides the USB 2.0 resources for connecting the AnaRMs to the Compute Cluster.

Mechanical Assembly 3U rack-mount for the 12 Flyspis, the Control Computer and four USB 2.0 hubs.

Each AnaRM consists of three PCBs:

Flyspi FPGA PCB (Flyspi) small data aquisition PCBs containing a fast ADC, an FPGA and 512MiB DRAM memory.

Analog Frontend PCB (AnaFP) Each Flyspi carries one AnaFP containing multiplexers and one pre-amplifier to connect the analog readout channels from the Wafer Module to Flyspi.

2.7.1 Flyspi FPGA PCB (Flyspi)

The analog readout is based on a small custom FPGA board designed in Heidelberg. Its main components can be seen in Fig. 2.7.1:

FPGA Spartan 6 FPGA providing the necessary logic resources

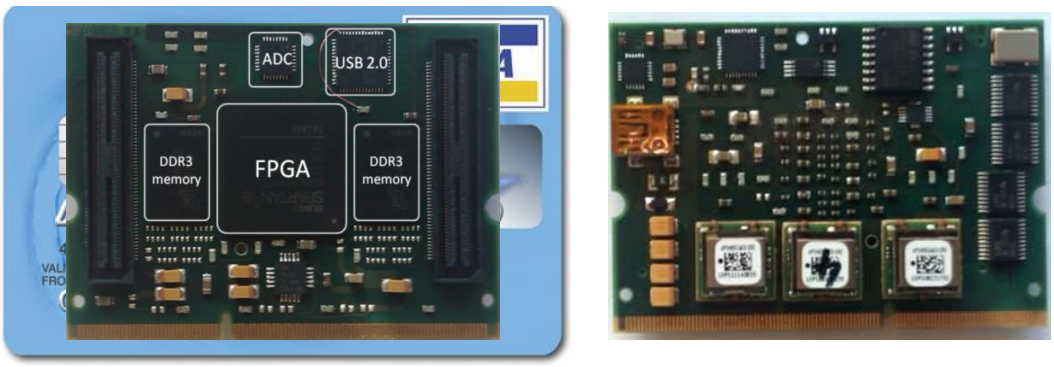


Figure 2.7.1: Photographs of top (left) and bottom (right) side of the Flyspi FPGA PCB (Flyspi).

ADC 12bit ADC with 125MHz sampling rate (Texas Instruments ADS6125).

DRAM 512MiB memory, sufficient to store 7.5hours of membrane voltage traces with full sampling rate in biological model time using an acceleration factor of 10^4.

USB 2.0 controller The USB 2.0 controller can sustain up-to 40MB/s transfer rate to the host. The USB 2.0 cable also supplies power to the AnaRM.

2.7.2 Analog Front End Board

The analog front end board contains an analog front end for the Spartan6 FPGA board. An analog front end comprises signal multiplexing, signal termination and pre-amplification to match the ADC's input voltage range. The front end board described here is connected to the Spartan6 FPGA board via two Samtec BTH-060-02-L-D-A (0.5 mm pitch) connectors.

Basically, this board selects one out of eight analog input signals, connects the selected signal to ground via a 50Ω termination and pre-amplifies the signal to match the ADC's input voltage range.

The input signal voltage range at the input of the pre-amplifier (after termination) is specified to be between 0 V and 0.9 V. The input signal is connected to the front end board via an 8x2 male pin header with a pitch of 2.54 mm. Three multiplexers with three inputs each can select one out of three input signals. Their outputs are connected to drive one common net that is connect to the input of the pre-amplifier and the termination resistor. The signal assignment for controlling the multiplexers from the point of view of the FPGA can be found in section 2.10.3.

The pre-amplifier circuit is a modified version of the one that is shown in the data sheet of the ADC (Texas Instruments ADS6125, Figure 95 in revision A, March 2008). It has been adopted to match the 2 V differential input voltage range of the ADS6125 with a single ended input voltage range of 0 V to 0.9 V.

According to the data sheet of the ADS6125 both input signals of the differential input the ADC should be changed between -0.5 V and 0.5 V around a common mode voltage of 1.5

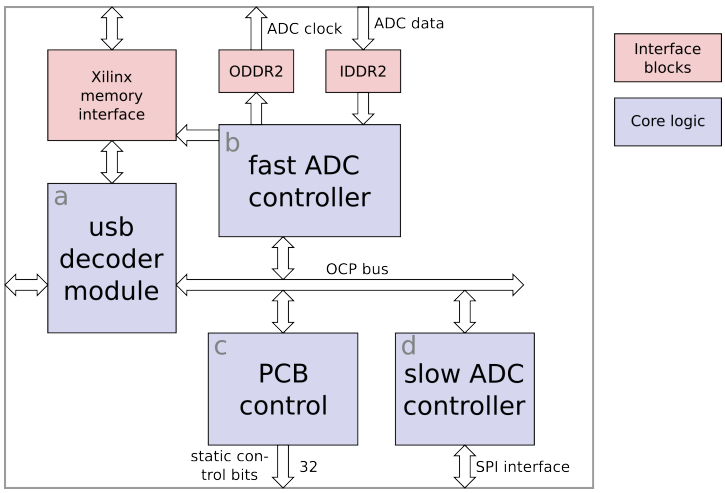


Figure 2.7.2: block diagram of the FPGA firmware for the analog readout board

V. This results in a peak-to-peak voltage rangen of $2V_{pp}$. This input voltage range is then mapped to the 12 bit digital output range between 0 and 4095.

In the case of the pre-amplifier used here, this results in a conversion between digital output of the ADC and a single-ended input voltage as described in the following equation, where ADC stands for a 12 bit unsigned integer value.

$$V_{inp} = 2.0 - 6.6 \cdot 10^{-4} \cdot \mathrm{ADC} + 5.7 \cdot 10^{-9} \cdot \mathrm{ADC}^2 \qquad (2.7.1)$$

2.7.3 FPGA Firmware and Software interface

The FPGA firmware consists of 4 modules that are depicted in figure 2.7.2 and will be described in subsection 2.7.3.1. The user's view on the configuration of the analog readout module will be described in subsection 2.7.3.2.

2.7.3.1 FPGA Firmware

All data to and from the analog readout board are sent via a USB interface in the FPGA firmware (module a on figure 2.7.2). This module can read and write data from and to the DDR memory on the Spartan6 board. It can also send write and write commands via the on-chip bus fabric that is based on the OCP[1]. The other modules which are described from here on are connected to the USB decoder via this on-chip bus. The bus has an address with of 16 bits and a data width of 32 bits. An overview of the base address of the different modules that are discussed in this section can be found in section 2.10.3.

A sampling run comprises the recording of `endaddr-startaddr` samples from the ADC and the storage of the resulting 12 bit data words to the DDR memory. A sampling run can

[1]Open Core Protocol, http://www.ocpip.org

be started by:

- a trigger signal that is connected to the analog readout FPGA and the Wafer-scale neuromorphic system or by
- an asynchronous start signal that can be sent via USB to the FPGA.

To start a sampling run, the FPGA and the ADC itself have to be configured. This configuration comprises three steps (all register definitions for these steps can be found in subsection 2.10.3):

1) To enable the power supply and to set the multiplexer on the analog front end board (c.f. section 2.7.2) several static control signals have to be set. These signals are controlled by control module c on figure 2.7.2.
2) To configure the actual ADC chip, there is an ADC controller module (module d in figure 2.7.2). This module sends configuration data via the SPI interface in order to set the internal reference, and to select the default analog input (or for debugging purposes the internal test pattern generator). Since this is a generic SPI module the details for this configuration step can be looked up in the data sheet of the Texas Instruments ADS6125.
3) Module b in figure 2.7.2 controls the sampling frequency of the ADC, reads back the parallel data stream from the ADC and writes it into the DDR memory via the Xilinx memory interface. This module is called "fast ADC controller" in this section and also controls the start and end time of every sampling run. Before starting a sampling run, the length of the sampling run (in terms of number of start address in memory and end address in memory) has to be set in the fast controller. Furthermore, the trigger configuration has to be set. There are two trigger input pins to the FPGA of which one can simultaneously be selected and which can be set to de-activated, always active or as single pass triggers.

2.7.3.2 Software Interface

To configure the Analog Readout Board, the user first has to acquire an object of the type `Handle::ADCHw`. This object can be initialized without parameters to acquire the first board in the list that libusb returns that matches the manufacturer and product ID. If the object is initialized with a `Coordinate::ADC` object as a parameter the handle tries to acquire a board with the given serial number.

All configuration information (for all three configuration steps explained above) is held in an object of type `ADC::Config`. This information comprises the trigger channel (`Coordinate::TriggerOnADC`), one out of eight measurement channels (`Coordinate::ChannelOnADC`) and the number of samples that the FPGA should store during one sampling run.

To prepare the board for measurement it first has to be configured by calling `ADC::config(Handle::ADCHw,ADC::Config)`. The `ADC::Config` object holds the trigger channel as `Coordinate::TriggerOnADC`, the measurement channels as

`Coordinate::ChannelOnADC` and the number of samples that the FPGA should store during one sampling run.

Now the trigger can be activate by calling `ADC::prime` or recording can be started immediately by calling `ADC::trigger_now`. The recorded data can be read by calling `ADC::get_trace`. Calling `ADC::get_trace` does not ensure that valid data is in the ADC memory. The user can check if the trigger has been activated by calling `ADC::get_status` or, when using `ADC::trigger_now`, has to wait an appropriate time. It can be read arbitrarily times without reconfiguring.

2.8 Compute Cluster and Networking

Author: Eric Müller

For configuration purposes, data analysis and execution of closed-loop experiments (cf. section 2.2.3) a conventional compute cluster complements the NM-PM system. Due to the nature of the NM-PM system experiment execution speed mainly depends on configuration and runtime data (i.e. spike or membrane traces) throughput. Maximizing experiment throughput is one of the major goals for compute node and network design. Another key feature is low-latency capability for closed-loop operation of the neuromorphic system and one or multiple cluster nodes.

2.8.1 Node architecture

During both, configuration and experiment execution phase, a single compute node handles up to 48×1-GbE [33] data streams[1] section 2.10.1.1 to a single Wafer Module. To achieve acceptable performance within the financial budget, as many desktop computer components as possible are deployed. At present, the Intel® Core™ i7-4770 CPU offers state of the art single-thread performance at reasonable cost. Per CPU a LINPACK benchmark performance of approximately 180 GFlops can be reached.

A single compute node consists of:

	#	Component
CPU	1	Intel i7-4770
RAM	≥ 16 GB	DDR3-1600 (or better)
Main board		Q87-based, KVM support via Intel AMT
	1	PCIe ×16 Gen3 (16 lanes) slot
	4	memory slots, DDR3-1600 (or better)
	≥ 4	USB 2.0 connectors
NIC	≥ 1	10GbE port(s)
	≤ 6 µs	low-latency, MPI with RDMA-support
Case		1U including 2×-redundant PSU

One or more I/O nodes additionally contain 40GbE PCIe-based Network Interface Controllers (NICs), 2TB (or more) fast storage (SSD-backed), 20 TB (or more) HDD-backed cluster

[1] one data stream per FCP

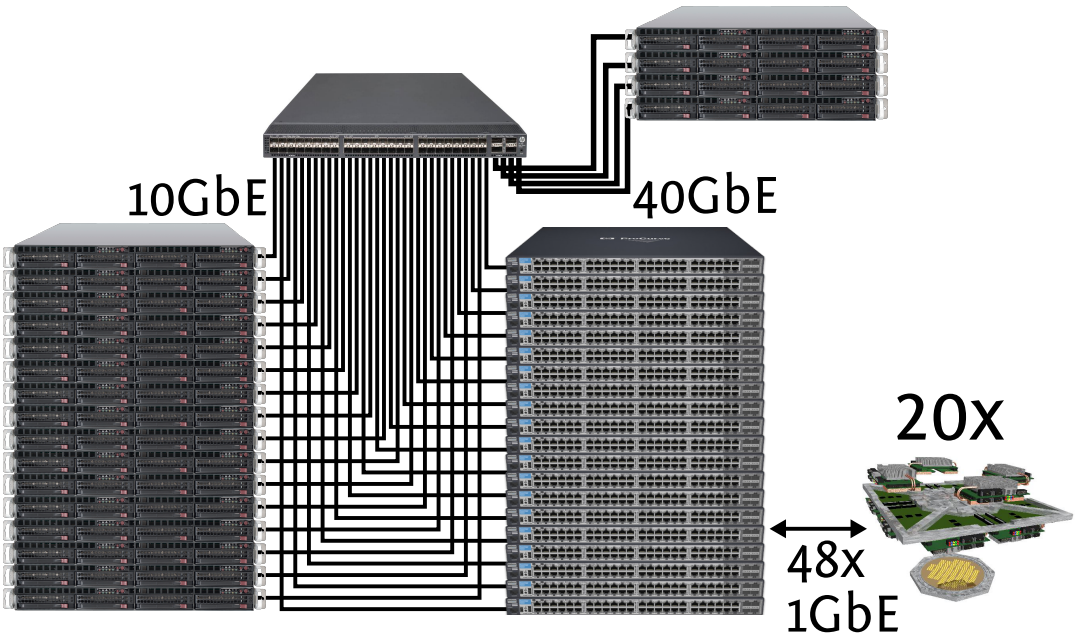


Figure 2.8.1: The network topology of the NM-PM1 system: 20 Wafer Modules, 20 Compute Nodes and one (or more; 4 are shown) I/O Nodes communicate via multiple switches. The Compute Nodes are directly connected to the central switch via 10GbE. The I/O Nodes are connected via 40GbE. 48 × GbE connections from each Wafer Module to the central ToR switch are aggregated into 10GbE.

storage.

2.8.2 Network architecture

Connectivity between wafers systems and hosts is established using multiple Ethernet standards. The FCPs controlling the neuromorphic system support a single GbE connection each. This sums up to 48×1-GbE upstream connections. An aggregation switch combines all connections into one (or more) 10GbE (cf. [36]) connections. Together with the 10GbE host links, all aggregated wafer connections are switch in a single ToR switch. This network topology can be upgraded to a complete fat-tree network. Control connections between frontend/ login and compute nodes are performed on a dedicated GbE network. Figure 2.8.1 shows the network topology.

	#	Component
Experiment switch	20	48-port GbE, 1-port 10GbE (or more) low-latency (< 3 µs)
Top-of-the-Rack switch	≥ 1	48-port 10GbE, 1-port 40GbE (or more) low-latency (< 3 µs)
Control switches	1	32-port GbE

For all connections between Compute Node, ToR and Wafer Module switches SFP+ or QSFP

direct-attach copper cables are used. External connectivity for the NM-PM1 system is provided by one or more fiber-based (10GBASE-SR) links to the Kirchhoff-Institute for Physics (KIP).

2.9 System Control and Power Supply Infrastructure

This section has been written by Maurice Güttler.

It is not feasible to use a neuromorphic hardware platform, when the user has to think about turning the power on or handling hardware errors. These tasks have nothing to do with the experiment and have to be hidden from the user. Therefore a control system is developed, which works in the background.

2.9.1 Power Supply

The complete Wafer Module is supplied from the main input voltage of the Wafer Module (-48 V) (V_MainIn). V_MainIn is only connected with the PowerIt board. On the PowerIt the V_MainIn generates the intermediate voltage for the Wafer Module (7-13.5 V) (V_intermed), which is the main supply voltage in the Wafer Module. The flow diagram of the system is illustrated in fig. 2.9.1.

5 V *Standby voltage* The V5_Stby is immediately turned on with the V_MainIn voltage and cannot be turned off. The only component supplied by the V5_Stby voltage is the Raspberry Pi. The Raspberry Pi is the master control unit, which has to be powered at first.

2.9.1.1 HICANN Voltages

The HICANN voltages are generated on the PowerIt and the AuxPwr boards and only supply the Wafer. The voltage regulators for the 1.8 V digital power supply voltage for the Wafer (1.8 V) (VDD) and 1.8 V analog power supply voltage for the Wafer (1.8 V) (VDDA) are on the PowerIt board. All other voltages are created on the AuxPwr boards.

The VDD and VDDA voltages from the PowerIt-Board supply the wafer as a whole, whereas, the voltages of the AuxPwr only supply one half of the Wafer (see fig. 2.9.2). This way no current sharing between the voltage regulators on the AuxPwr boards is needed, therefore, the complexity of the boards and the failure proneness is reduced.

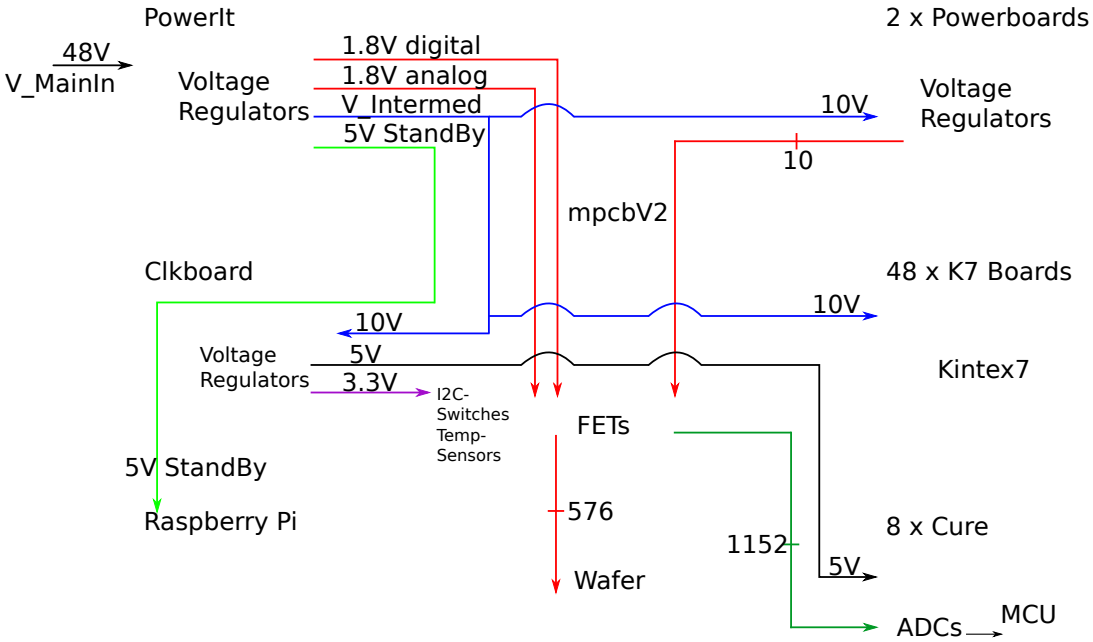


Figure 2.9.1: Power infrastructure of the Wafer-Scale Integration System, red lines represent the wafer power supply, the V5_Stby voltage(green) is reserved for the Raspberry Pi, the V_intermed voltage(blue) supplies the FCP, AuxPwr and the AnaB boards

Voltage name	Voltage/V	Current/A	Board
VDD	1.8	200	PowerIt
VDDA	1.8	200	PowerIt
DI_VCC	1.8	10	AuxPwr
DI_VCCANA	1.8	2	AuxPwr
DI_VCC33ANA	3.3	7.8	AuxPwr
DI_VBias	1.25	0.1	AuxPwr
V_OL	0.6 - 1.1	19.2	AuxPwr
V_OH	0.6 - 1.1	19.2	AuxPwr
VDD25	2.5	15	AuxPwr
VDD12	11	0.1	AuxPwr
VDD5	5	0.1	AuxPwr
VDDBUS	1.2	115.2	AuxPwr

Table 2.9.1: List of the HICANN voltages with voltage and current values. The current values represent the consumption of the entire Wafer.

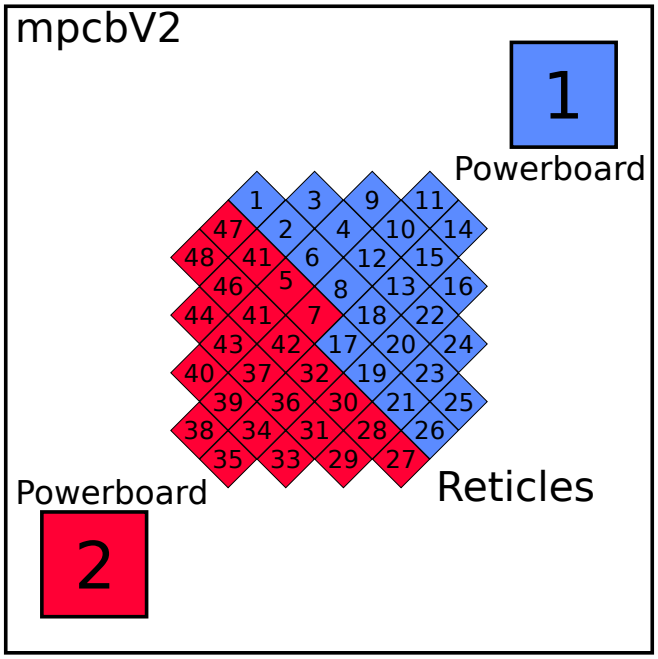


Figure 2.9.2: Reticle power supply from the two AuxPwr boards

2.9.1.2 Reticle Power Supply

Each reticle on the Wafer can be switched individually. This is done by using Power-FETs on the MainPCB as High Side Switches. Figure fig. 2.9.3 shows the schematic of one Power-FET. On the MainPCB three different Power-FETs from Vishay Siliconix are used. The Si5903DC [7] is a p-channel Power-FET, which is needed to switch VDD12 and VDD5. The other Power-FETs are the SiA912DJ [6] and the Si7234DP [5], both are n-channel Power-FETs. The Power-FET gate is controlled by the Cure board. The 12 voltages are divided into four groups, so one Enable-line goes to several gates. The following list shows the four groups with the corresponding voltages:

1) DI_VCC, DI_VCCANA, DI_VCC33ANA, DI_VBias: LVDS common mode voltage (DI_VBias)

2) VDDA, VDD25, VDDBUS

3) VDD, V_OL, V_OH

4) VDD5, VDD12

Furthermore, there are certain dependencies between voltages, for example, between VDD12 and VDD5. The floating gate cells on the Wafer could be destroyed by turning VDD12 on without VDD5. So using one enable line for both voltages, makes the wafer handling easier and safer.

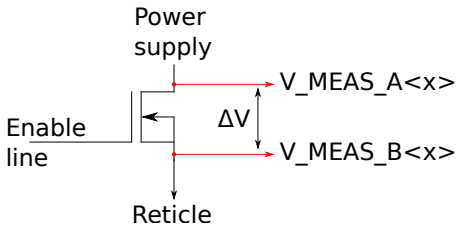


Figure 2.9.3: An example of a Power-FET on the MainPCB. The drain is connected with the power supply, the Source pin goes to the Wafer and the gate and the V_MEAS lines are connected to a Cure board.

To measure the current flowing into the wafer, there are additional measuring lines(V_MEAS_A and V_MEAS_B) before and after the Power-FET. The measuring lines are routed to the Cure boards. The current measuring method is described in section 2.9.2.4 later.

2.9.2 Control System

The concept of the control system consists of small low-level components and one high-level MaCU. The low-level components, e.g. the Cure boards, are only responsible for a specific function and a spatial part of the system. They follow a command from the system control unit, but can also act on their own under certain circumstances. Whereas the MaCU is coordinating the low-level components and gathering all system parameters, like temperatures, power supply state etc.

2.9.2.1 Communication Channels

The system has three types of communication channels. Figure 2.9.4 shows all communication buses. There are four I2C buses, one 1wire bus and one JTAG chain.

1-wire The 1-wire (1-wire) bus is only used for 1-wire temperature sensors, which are placed in the WBr. The advantages of 1-wire is, that it only needs three lines VCC, GND and Data[2].

JTAG A JTAG chain on the MainPCB board links all Cure boards together. So it is possible to debug or program the microcontrollers on the Cure boards within the system. The programming time increases with the chain length. But the programming time is negligible, since it happens not very often.

I2C

This section describes the structure and tasks of the I2C chains in the Wafer Module. For a technical description of the I2C Bus see [52].

The system components communicate over several I2C buses. For every chain the Raspberry Pi is the I2C-master.

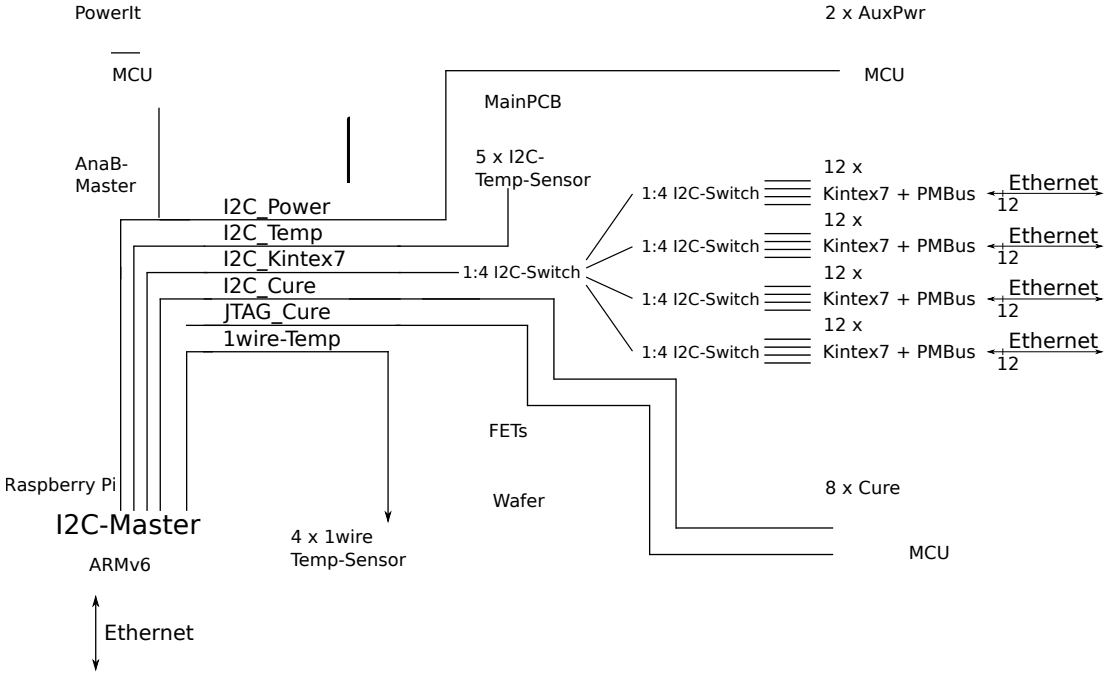


Figure 2.9.4: Communication Channels in the Wafer Module

The buses are separated, because it increases the system reliability. If one I2C slave spontaneously stops working and blocks the bus by pulling down the lines, the other buses are not affected. Thus allowing the Raspberry Pi to bring the system in a state, where a defect cannot damage more parts.

The following list contains the I2C chains and the components they are connect to.

- **I2C_Power** : PowerIt, Auxiliary Power Supply PCB
- **I2C_Temp** : I2C-Temperature sensors
- **I2C_FCP** : FPGA Communication PCB
- **I2C_Cure** : Monitoring and Control PCB for Reticles

I2C_FCP The I2C_FCP bus is different from the other three buses. The bus is used to communicate with FCP boards.

On each of the FCP boards is a Power Management Bus (PMBus) controller. Unfortunately, the PMBus controller has a limited address region, there are only four I2C addresses available. So not more than four FCP boards can be attached to a I2C bus at the same time.

Instead of creating 12 different I2C buses, I2C multiplexer[4] are used to divide the bus into separate parts. The I2C multiplexer splits one bus into four buses. With two stages of multiplexers it is possible to reach every FCP board over one I2C bus (see fig. 2.9.4).

The Raspberry Pi has to know the routing table to all FCP boards. From the Raspberry Pis point of view the multiplexers are ordinary I2C devices, which are controlled via the I2C bus.

I2C-Temperature Sensor	I2C-Address
1	0x38
2	0x98
3	0xB8
4	0xF8
5	0xD8

Table 2.9.2: I2C-Addresses of the temperature sensors, see also Figure fig. 2.9.5

Hence, the I2C-master has to configure the I2C multiplexers, before he can communicate with one of the FCP boards.

2.9.2.2 System Monitoring

The System uses three physical measurands to determine its current state. These are temperature, voltage and current. Each of them gets measured at certain locations in the system. In addition to these measurands the system checks the availability of the components at regular intervals.

Temperature Measurement The WBr contains four 1wire temperature sensors [2]. The sensors measure almost the correct Wafer temperature, because there is only 2.68 mm of aluminium between the Wafer and the sensors. Each 1wire temperature sensor has a unique address, which has to be readout before the placement in the WBr.

Furthermore there are five I2C-temperature sensors [47] on the bottom side of the MainPCB (see fig. 2.9.5). Each of these sensors can measure an extra temperature with an external diode. The extra diodes are used to measure the temperature under the ToCo. The circles in fig. 2.9.6 mark the positions of the measurements. The gathering of the data is done by the Raspberry Pi(see section 2.9.2.3).

Voltage and Current Measurement The voltages and currents get measured on two different system levels. On the top level the PowerIt and the AuxPwrs measure the voltages and currents, before they flow into the MainPCB. This represents the global power consumption of the entire system.

On the reticle level, the power consumption of a single reticle is measurable. This is possible, because of the dual use of the Power-FETs on the MainPCB (see section 2.9.1.2). The main purpose is to use them as switches, to turn the reticle on or off. Furthermore, the voltage before and after the Power-FETs is measured. The subtraction of the two values returns the voltage drop over the Power-FET and with the Drain-Source On-State resistance, the current flowing through the PowerFET can be calculated.

The measurement and calculation is done on the Cure board (see section 2.9.2.4).

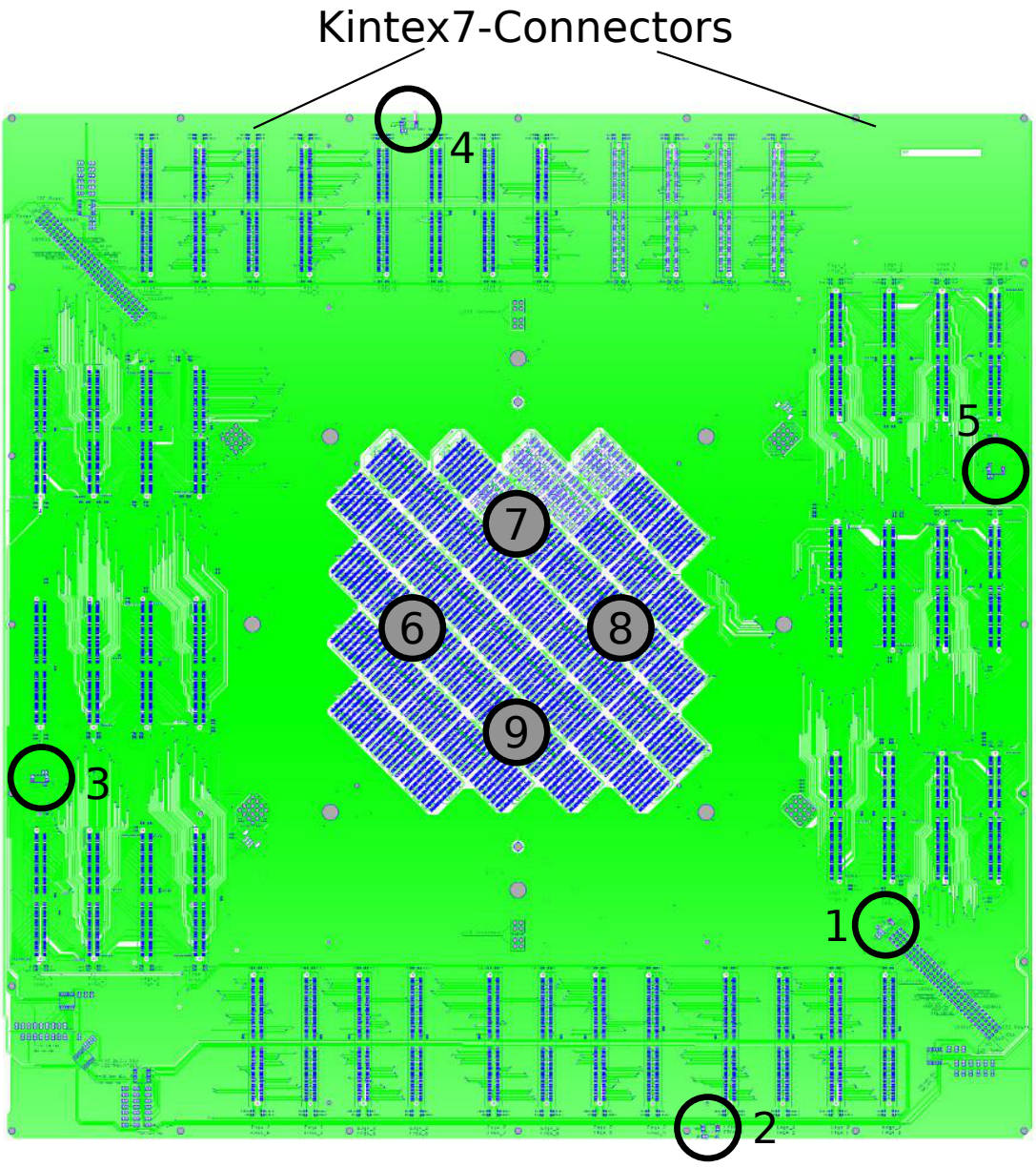


Figure 2.9.5: Position of the temperature sensors on the bottom side of the MainPCB. The I2C temperature sensor are at the board edges (no. 1-5). The 1wire temperature sensor are placed in the WBr(not visible in this image) (no. 6-9).

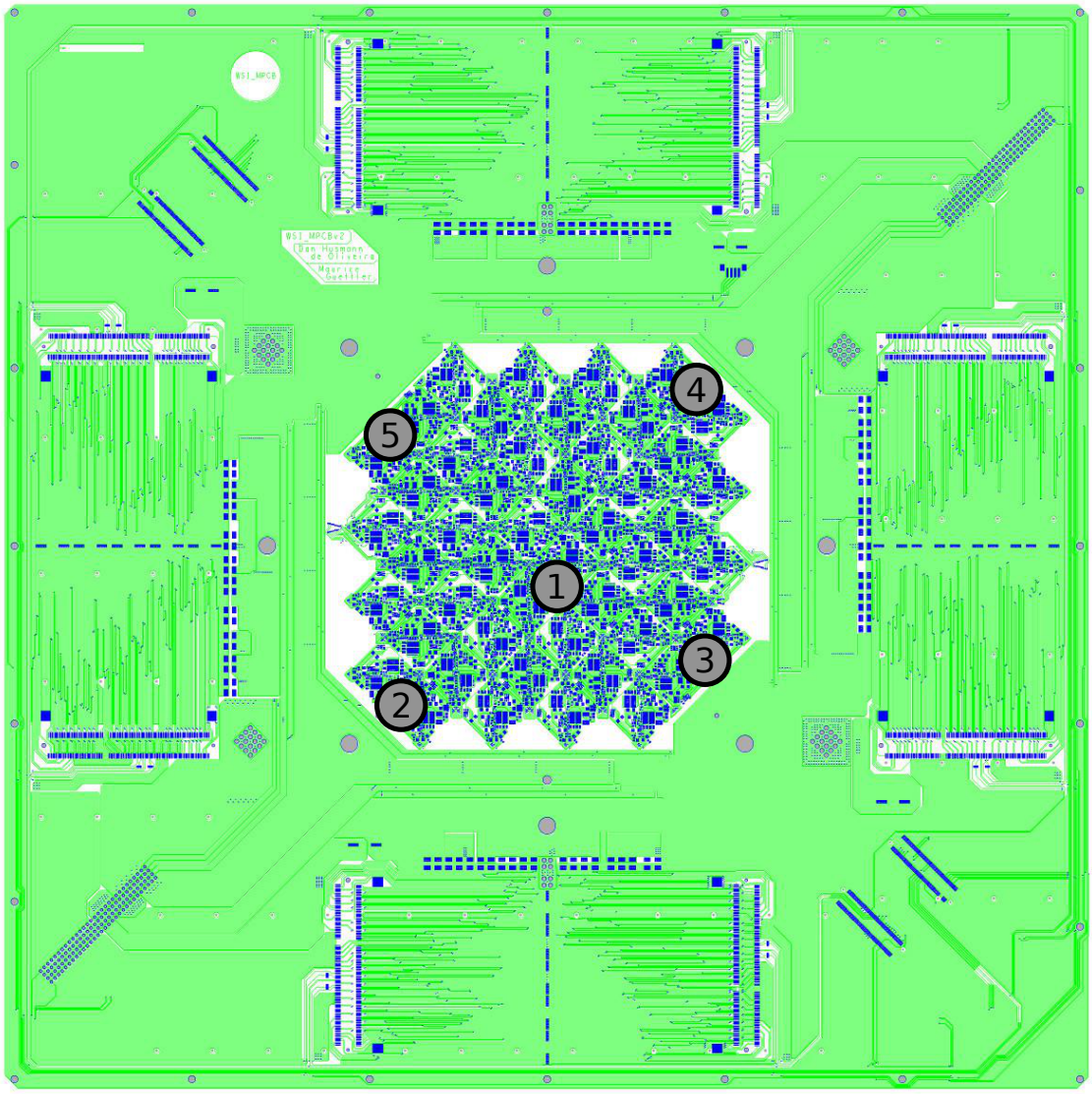


Figure 2.9.6: Temperature sensor under the Top-Cover. The numbers correspond to the I2C-temperature senors on the bottom side(no. 1-5)

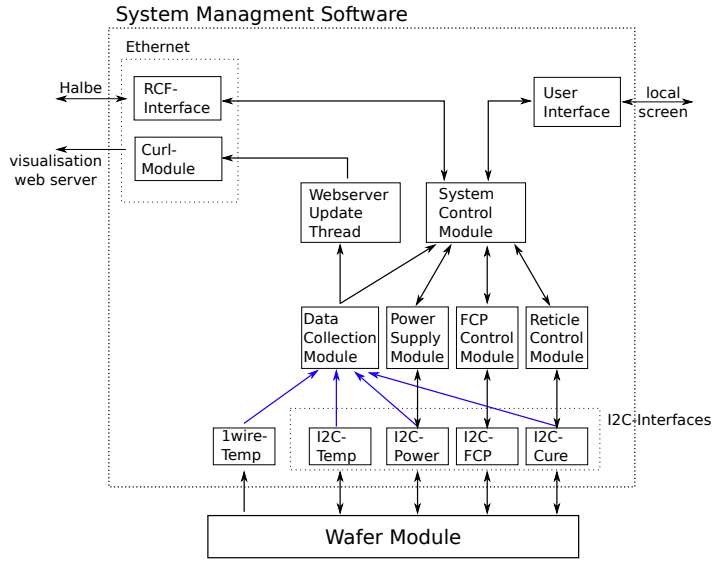


Figure 2.9.7: Internal structure of the System Managment Software

2.9.2.3 Raspberry Pi - Main System Control Unit

The Raspberry Pi is the central control unit of a Wafer Module. It has a standard Ethernet connector for the communication with the computer hosts and additionally, there are enough communication channels to the Wafer Module available. Its low price and big user community make it the ideal interface between the hosts and the Wafer Module.

The connector definition with the Wafer Module can be found in section 2.5.3.10. This section looks into the software on the System Control Unit.

On the Raspberry Pi runs a linux operation system, called raspbian [58]. The underlying operating system handles the hardware control and presents an user-friendly API. Thus, the development of the System Managment Software (SMS) is easier and more independent from the actual hardware platform.

The SMS is a C++ program, which handles incoming commands from the HALbe, the readout of temperature sensors, the web server updates, the communication with different I2C-devices and automated tasks like power up. In fig. 2.9.7 the internal structure of the SMS is illustrated.

There are only two ways of interaction with the program. The primary access goes over the HALbe/Remote Call Framework (RCF)-interface. Over this interface the correct usage of the Wafer Module is granted. Whereas, the interaction over the local screen is less restricted. It is designed for debug and maintenance purpose only and not for continuous operation.

For the monitoring of the Wafer Module there is the web server update thread. It gathers the information from the components, like temperature or current, and sends them via the "Curl-module" to the web server.

Another important module of the software is the "system control module". It is responsible for the coordination of system-wide tasks, e.g. the system shutdown. In section 2.9.2.5 the sequence plans for the power up and shutdown can be found. In addition the module works

Cure-Board-ID	Reticles
0	1, 2, 3, 6, 45, 47
1	4, 8, 9, 10, 11, 12
2	13, 14, 15, 18, 16, 22
3	17, 19, 20, 23, 24, 25
4	21, 26, 27, 28, 29, 30
5	31, 32, 33, 34, 35, 36
6	37, 38, 39, 40, 42, 43
7	5, 7, 41, 44, 46, 48

Table 2.9.3: Reticle-Cure board mapping

Cure-Board	Socket-IDs			I2C-address		
	<2>	<1>	<0>	PIC1	PIC2	PIC3
0	0	0	0	0xC2	0xC4	0xC6
1	0	0	1	0xE2	0xE4	0xE6
2	0	1	0	0xD2	0xD4	0xD6
3	0	1	1	0xF2	0xF4	0xF6
4	1	0	0	0xCA	0xCC	0xCE
5	1	0	1	0xEE	0xF0	0xF2
6	1	1	0	0xDA	0xDC	0xDE
7	1	1	1	0xFA	0xFC	0xFE

Table 2.9.4: I2C-addresses of the microcontrollers on the Cure boards

as a security layer, which checks the correct usage of the system. For example, the module only allows to turn the reticles on, if the power supplies and Cure boards are running.

Another security feature is the regular availability check of all system components. If a component has a problem and is not responding, the module has to prevent the others from being affected or damaged. A detailed description of the emergency plans is in section 2.9.2.6.

2.9.2.4 Monitoring and Control PCB for Reticles - Cure

One Cure board monitors and controls six reticles on the MainPCB. It has 24 enable lines and 144 voltage measuring lines from 72 Power-FETs (see fig. 2.9.3). Obviously, this number of lines can not be handled with one microcontroller. Therefore, three Microchip dsPic33FJ microcontrollers [8] are placed on the Cure board, so that every microcontroller is responsible for two reticles. Table 2.9.3 shows the reticle-Cure board mapping.

Every microcontroller has a 7-bit I2C address, which is a combination of fixed bits, a Cure board number and a microcontroller number. In table table 2.9.4 all microcontroller I2C-addresses are listed.

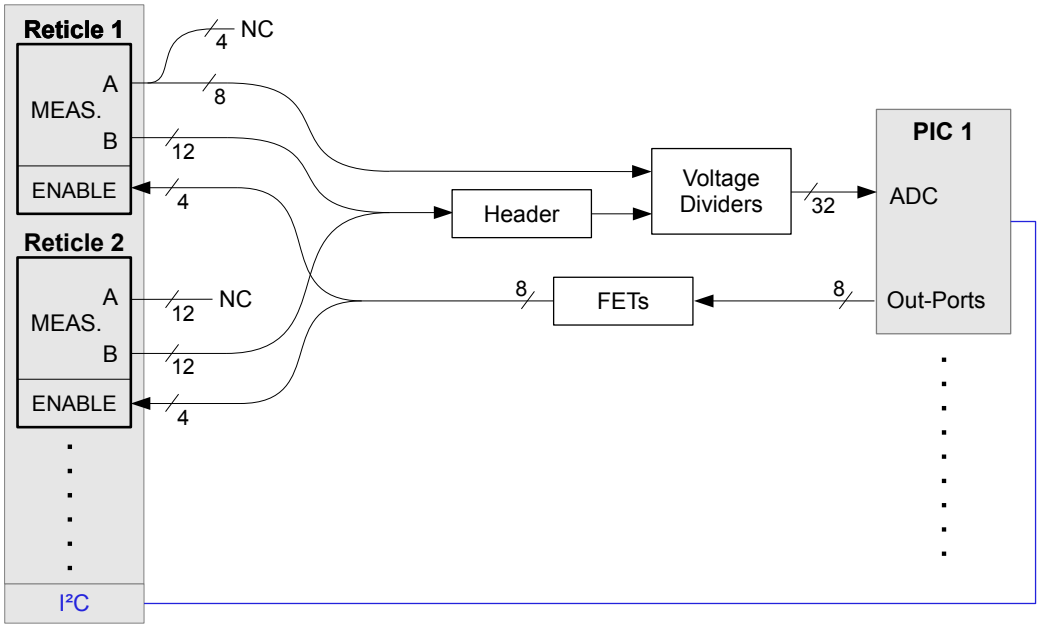


Figure 2.9.8: Block diagram of the Cure board. Image created by Joscha Ilmberger.

Current measuring Every microcontrollers has two ADCs with 16 ADC inputs each. So in total there are 96 inputs. However, this is not enough to digitize all the 144 measuring lines from the reticles.

The voltage after the Power-FETs has to be measured by the microcontroller. Because only with these voltages it is possible to determine, if the reticle receives the applied voltage and if the Power-FET is working properly. As a result, this leaves only eight free inputs at each microcontroller.

The first and third microcontroller use the free inputs to measure eight voltages before the Power-FETs from one reticle. These voltages are VDD, VDDA, V_OL, V_OH, VDDBUS, DI_VCC, DI_VCC33ANA and VDD25. The second microcontroller measures the supply voltages VCC33, VCC5, VCC12 and the ADC reference voltage. Figure 2.9.8 shows schematically the routing of the measuring lines for the first microcontroller.

Although only one third of the HICANN supply voltages are measure before and after the Power-FETs, the current flowing through the Power-FETs can be estimated. The voltage values before the Power-FET are replace by the values from the PowerIt and the AuxPwrs boards. The value of the Drain-Source On-State resistance, which is needed for the calculation, is taken from the datasheet (see [7], [6], [5]).

$$I = \frac{U_{pwrboard} - U_{pwrfet}}{R_{ds\_on}} \tag{2.9.1}$$

Of course, the current value received from eq. (2.9.1) is not very precise. It does not consider

the voltage drop from the power boards to the Power-FET, which means the calculated current is always higher than the actual value. Nevertheless, this offset does not affect the overcurrent protection. It only means, that the threshold for overcurrent has to be set higher.

2.9.2.5 System Sequence Plans

Power up sequence

The power up sequence is split into two steps. At first the wafer module gets the V_MainIn voltage, which initiates all the components needed for a controlled system start, e.g. the Raspberry Pi. Afterwards the systems waits for the command to power up everything else.

Initialisation Phase When the 48V power supply is turned on, the Raspberry Pi, the AuxPwr boards , the Cure boards and the PMBus controller on the FCP boards start and run a series of self-tests. Figure 2.9.9 shows this process. The results of the self-tests are evaluated with the SMS on the Raspberry Pi. Only if all tests pass, the system can proceed with the next phase.

PowerUp Phase After the initialisation phase is finished, the system waits for the start command from the user. When the start command is received, the system follows the PowerUp-Sequence shown in fig. 2.9.10.

The first modules to start are the AuxPwr PCBs. The microcontrollers on the AuxPwr boards turn the voltage regulators on one by one and start monitoring immediately. If a problem with one of the regulators occurs, the microcontroller stops the start-up sequence and turns off all regulators. Then the master controller detects the error and sends the command to turn off all voltages on the second AuxPwr too.

After all voltage regulators on the AuxPwr boards are on, the VDD and VDDA are the next voltages to be turned on.

The last step is to turn on the FCP modules. Each FCP checks itself for proper operation. If every test is passed, the system is ready and reticles can be turned on.

Power down sequence

The shutdown of the system is in general the power up sequence in reversed order. It is not recommended to shutdown the system at once. Because there can be reverse currents with high voltage peaks, which could damage sensitive electronics. Therefore, the reticles are turned off one by one with a small delay.

2.9.2.6 Error Management

Although, all components are tested before the assembly of a Wafer Module, it is possible that an error occurs during the system operation. Therefore, the System Managment Software and the low-level components have integrated error handling routines.

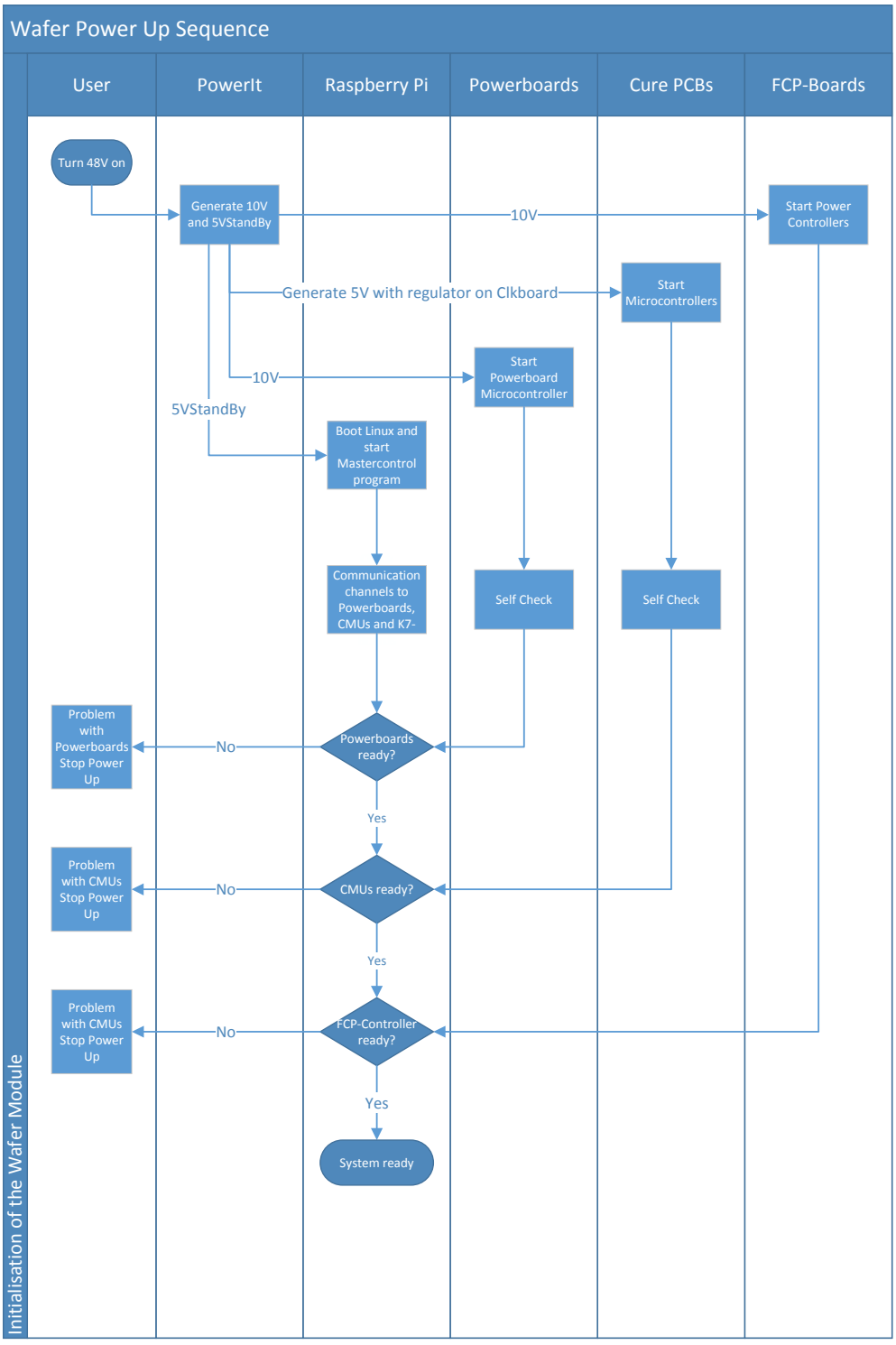


Figure 2.9.9: Initialisation of the wafer module

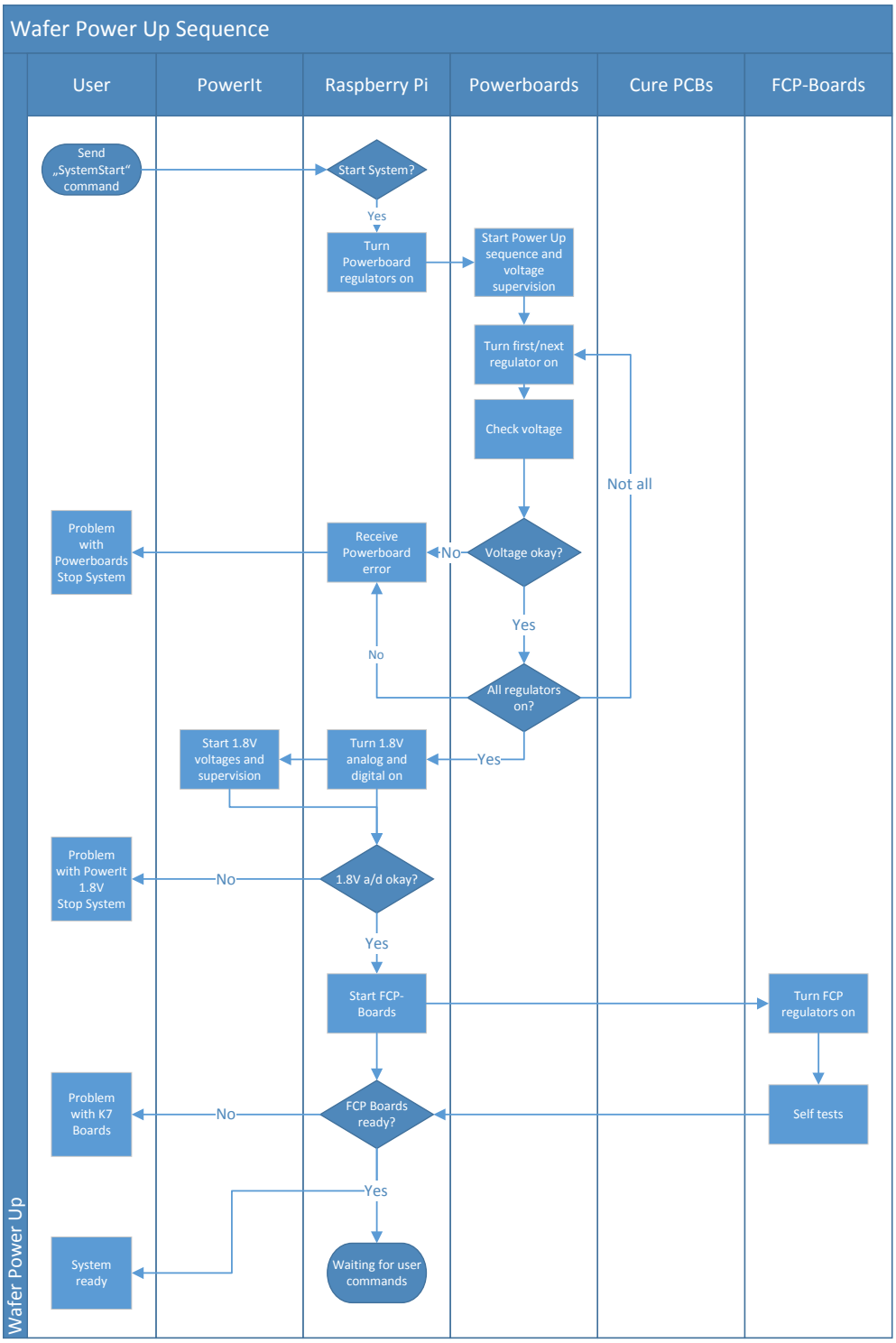


Figure 2.9.10: Power up the wafer module

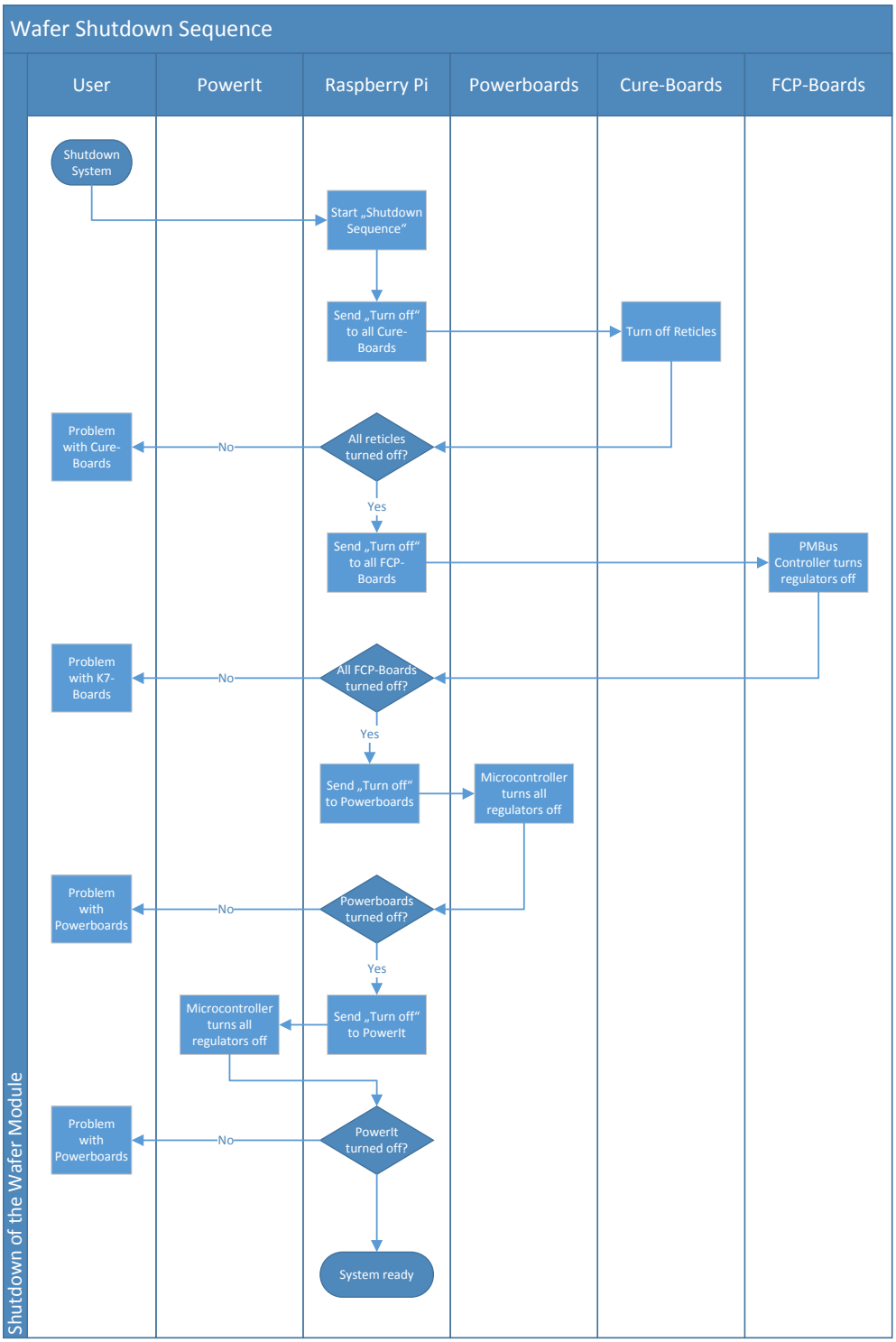


Figure 2.9.11: Flow chart of the wafer shutdown sequence

Overcurrent problem - Reticle level

Parts: Monitoring and Control PCB for Reticles
Problem: A Cure board detects one or more voltages with a current value above a defined threshold
Sequence plan:

- Turn off all Power-FETs of the reticle
- Set Overcurrent-Flag for this reticle

A problem with one reticle does not require a shutdown of the entire system. This way other experiments, which are using other reticles, are not disturbed and can finish their run.

Over-/undervoltage problem - Reticle level

Parts: Monitoring and Control PCB for Reticles
Problem: One or more HICANN voltages are not in their correct working region.
Seqence plan:

- Turn off the voltages of this reticle
- Set OV/UV-Flag

The Power-FETs cannot regulate the HICANN voltages, they only act as switches (see section 2.9.1.2). Therefore, the problem is either on the Wafer or on the PowerIt/AuxPwr boards. Nevertheless, for safety reasons the reticle is turned off.

Overcurrent problem - Power supply level

Parts: PowerIt Main Power Supply PCB, Auxiliary Power Supply PCB
Problem: One or more voltage regulators exceed a user-defined current threshold.
Sequence plan:

- Turn off all reticles
- Turn off all HICANN voltages, but keep the other supply voltages on
- Set Overcurrent-Flag of the voltage regulator

Over-/undervoltage problem - Power supply level

Parts: PowerIt Main Power Supply PCB, Auxiliary Power Supply PCB
Problem: One or more voltages are not in their correct working region.
Sequence plan:

- Turn off all reticles
- Turn off all power supplies
- Send e-mail to hardware maintenance mailing list

Overtemperature problem (soft limit)

Parts: 1wire Temperature sensor in the Wafer Bracket, I2C Temperature sensor on the Wafer Module Main PCB
Problem: The temperature of one or more sensors is above a user-defined soft limit.
Sequence plan:

- Send e-mail with system information to the hardware maintenance mailing list

The soft limit is not a critical point for the system. Nevertheless the hardware maintenance group should look for possible defects, like a failure of the air-conditioning system.

Overtemperature problem (hard limit)

Parts: 1wire Temperature sensor in the waferbrackt, I2C Temperature sensor on the Wafer Module Main PCB
Problem: One or more temperature sensors measure values above a user-defined hard limit
Sequence plan:

- Turn off all reticles
- Turn off all HICANN voltages
- Send e-mail with information about system status to the hardware maintenance mailing list

In this case, the temperature has risen above a threshold, where a hardware damage is imminent. Therefore, the system ends its operating phase and goes into a state, where only the Raspberry Pi, the Cure boards and the microcontroller on the power supply boards are running.

Level-1 component not reachable

Parts: PowerIt Main Power Supply PCB, Auxiliary Power Supply PCB, FPGA Communication PCB
Problem: A component is not responding to any request.
Sequence plan:

- Shutdown the system, like in Figure fig. 2.9.11 However, the sequence is not interrupted, if an error occurs.
- Send e-mail with information to the hardware maintenance mailing list

Level-1 components are essential parts for the system operation, like the PowerIt and the Cures boards. The Raspberry Pi tests in regular intervals the communication channels to these Level-1 components. If an error occurs, the system gets shutdown down immediately.

Level-2 component not reachable

Parts: 1wire Temperature sensor in the waferbrackt, I2C Temperature sensor on the Wafer Module Main PCB
Problem: A temperature sensor returns no data anymore.
Sequence plan:

- Remove the sensor from data acquisition list
- Send e-mail to hardware maintenance mailing list

Level-2 components are not important for the system operation. Only if all temperature sensors of the Wafer Module are not responding, the Raspberry Pi turns the system down.

2.10 Hardware-Software Interface

2.10.1 Host to FCP Communication

This section specifies the packets used for the communication between Host computers and FCPs. This communication channel utilizes Ethernet, UDP/IP and a custom ARQ-style protocol. For a specification of the latter see section 2.10.1.1. Different data types (e.g., configuration data, routing and pulse data) are specified in application layer (cf. section 2.10.2).

As the Ethernet frames, which belong to the Link Layer (cf. [38]), are handled by the FCP's Media Access Controller (MAC) unit these are omitted. For a specification of the utilized standard packet headers (i.e. Ethernet, Internet Protocol version 4 (IPv4) and UDP headers) see [24, 56, 55].

2.10.1.1 Transport Layer Protocol

Within the OSI model [38] the transport layer provides end-to-end communication channels. The protocols can be further separated into connection-oriented or connection-less models. Error detection and correction, flow control and ordering of data are additional features of transport layer protocols.

The NM-PM1 contains many concurrent data streams per wafer unit: 48 FCPs connected to the cluster node via 48 GbE links, multiple Analog Readout Modules (AnaRMs) (cf. chapter 2.7) are connected via their intermediate control computers using USB 2.0 and GbE. Requirements are high throughput and reliable communication channels for system configuration and most operational modes (i.e. all operation modes except for real-time communication, cf. section 2.2.3).

A simple protocol implementing the requirements is the ARQ protocol realizing the go-back-N packet-based sliding window method [70]. Messages, i.e. data frames, carry one sequence number identifying the packet itself and one acknowledge number confirming the successful reception of remote data. Packets that have not been acknowledged by the remote endpoint will eventually, after timing out, be resent to the remote endpoint. Incoming data is handled in order; missing packets interrupt data handling until the corresponding resends will resume the data stream handling.

An easy measure to estimate the required memory or window size given a protocol delay is the bandwidth-delay or throughput-delay product:

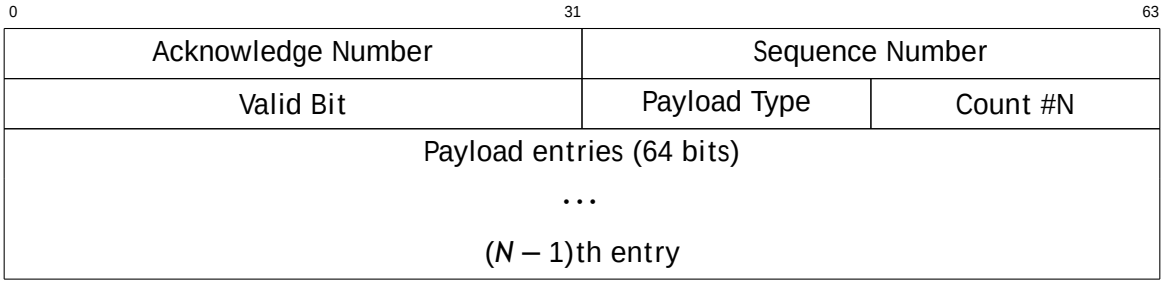
$$T \times D = C$$

The link capacity C represents the bits-in-flight. The delay D comprises mostly protocol handling times but also link delays (wire, network hardware). Typical software delays (without real-time constraints) are in the order of milliseconds. Thus:

$$10\,\mathrm{GBit/s} \times 1\,\mathrm{ms} = 10 \cdot 10^6\,\mathrm{Bits} \approx 1.2\,\mathrm{MiB}$$

Assuming standard-sized Ethernet frames (i.e. 1500 Bytes) this yields approximately a window size of 1.19 MiB/1500 ≈ 800 frames distributed over 48 streams. The FPGA firmware uses 512 frames per connection by default.

The packet format (omitting Ethernet, IP and UDP headers) looks like this:



The sequence field marks the packet number within the sender's window. The acknowledge field indicates the last successfully received packet from the communication partner. Packets carrying a non-zero valid-field are valid data packets if the length field indicates a size > 0. Payload type and the number of payload entries are stated in the 16-bit type/length fields.

2.10.2 Host to FCP Payload Data Formats

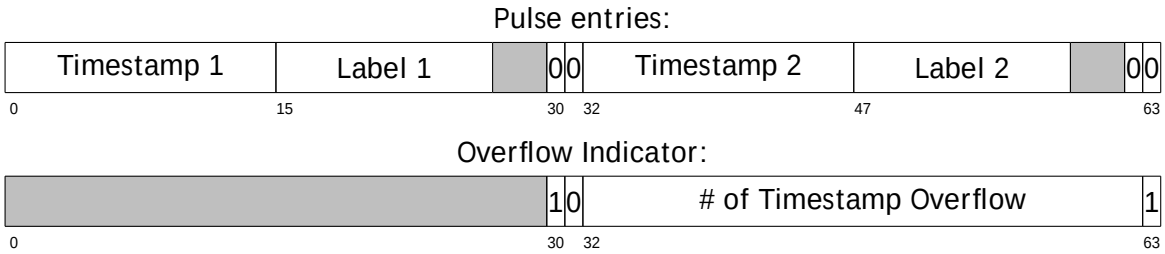
All payload data types are listed in Table 2.10.1. The payload data is aligned to 64 bit for more efficient handling. It can have a maximum length of 1456 Byte (maximum IP frame length minus IPv4, UDP and ARQ headers).

Hex. Code	Type
0x0CA5	FPGA Trace/Pulse
0x0C5A	FPGA Playback Memory
0x0C1B	FPGA Configuration
0x2A1B	HICANN Configuration

Table 2.10.1: Application Layer Packet Types

2.10.2.1 FPGA Trace / Pulse Data

Payload Type: 0x0CA5
Format:



Field	Length [Bits]	Description
Timestamp	15	timestamp of the received pulse
Label	12	address of the pulse, from MSB to LSB: 3 bit HICANN ID 9 bit neuron ID
Overflow	31	overflow count since experiment start

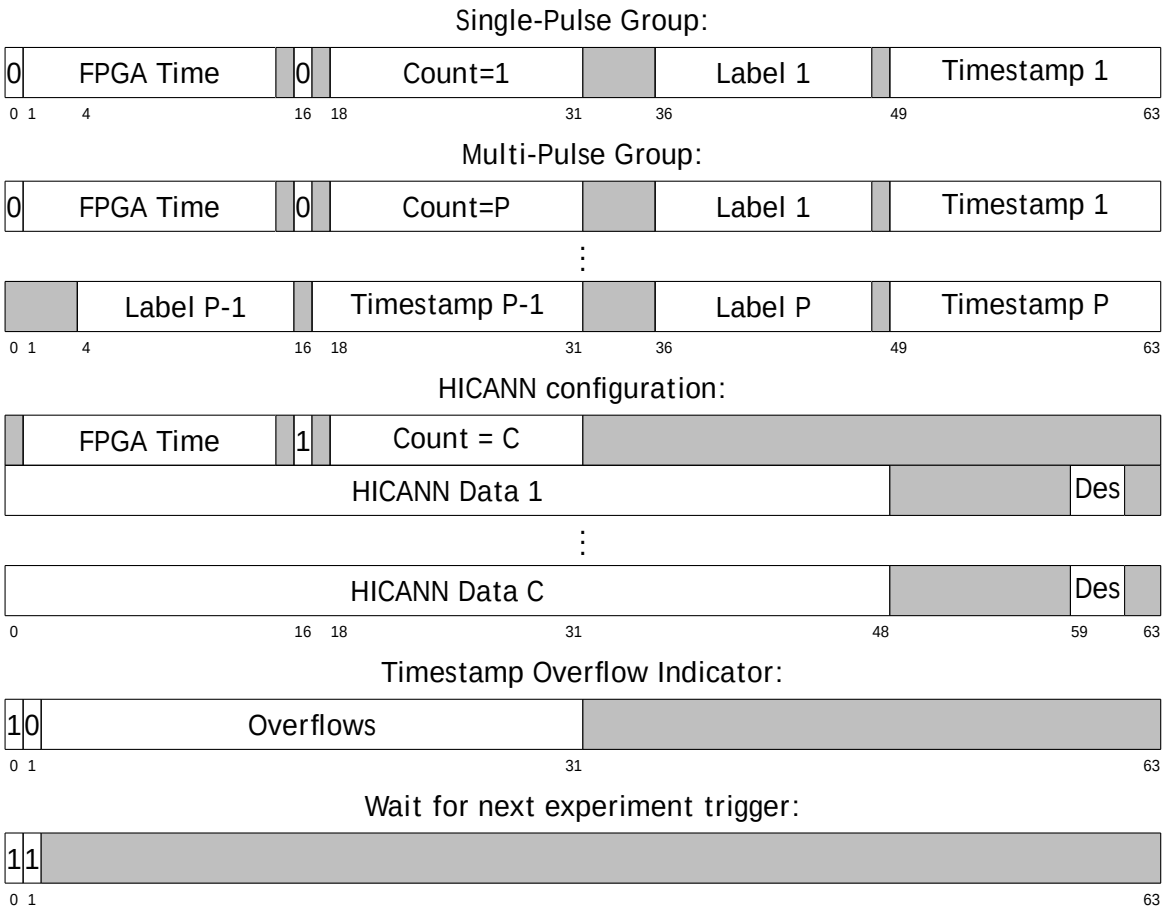
Description:
This frame is used for sending pulses stored in the trace memory to the host. In principal, the layout of the frame (except for the frame header) is the same as in the trace memory.

Handling:
The timestamp, the HICANN ID and the neuron ID is contained in the pulse packet coming from the HICANN. Overflows in the timestamp counter of the FPGA are stored as separate trace memory entries with a leading 'high' bit. Such an entry is generated always at an timestamp overflow. Due to the higher width of memory entries (currently 64bit), an empty pulse entry may be following an overflow indicator, denoted by MSBs '01'. Once the host requests the reading of the trace memory via an FPGA configuration packet, frames with the above format are generated in the FPGA and sent to the host.

2.10.2.2 FPGA Playback Data

Payload Type: `0x0C5A`

Format:



Field	Length [Bits]	Description
FPGA Time	14	release timestamp of the group in the FPGA
Pulse Entries P	14	number of pulse packets that should be generated at the FPGA release time
Timestamp	15	timestamp field of the packet
Label	12	destination address of the pulse, as defined in FPGA trace/pulse data
Config Entries C	14	number of config. packets that should be generated at the FPGA release time
Des	3	destination HICANN ID of the config packet
HICANN Data	49	content of the sent HICANN config packet (lower 49 bit)
Overflows	30	number of timestamp overflows to wait before continuing

Description:

This packet type is used for transmitting stimulus data to the playback memory of the FPGA. It can carry both spike data and HICANN configuration data, allowing for synchronisation between the two. A 'high' bit at position 48 in the next entry indicates a control command

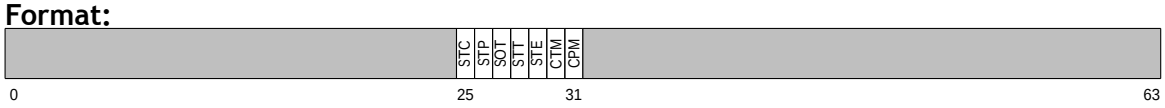
for the playback state machine. Currently, there are two such commands, separated by the following select bit (position 15):

- **select='low'** The execution should be stopped until the specified number of timestamp overflows occured. This is required for correct release of packets that are separated by more than half the time span of a timestamp counter cycle.
- **select='high'** The execution should be stopped until the next global experiment start signal is received (via an FPGA configuration packet). This command is used to synchronize the beginning of an experiment run over the whole system.

Handling:
All packets of this type are forwarded directly to the playback memory of the FPGA and stored there consecutively. Once an experiment start signal is received via an FPGA configuration packet, the content of the playback memory is read out and pulses or configuration packets sent to the HICANN. This process is only stopped when all stored entries have been played back or when a wait command in the playback memory is read. In both cases, the playback state machine waits for the next experiment start signal. Additionally, if the end of the stored entries has been reached, the read and write addresses for the memory are reset.

2.10.2.3 FPGA Configuration

Payload Type: `0x0C1B`
Format:



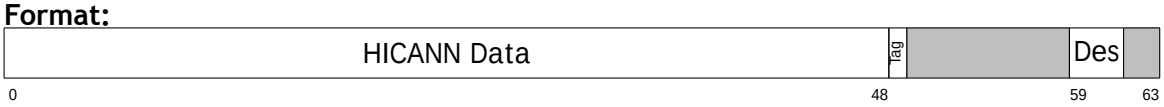
Field	Length [Bits]	Description
CPM	1	Clear Playback Memory
CTM	1	Clear Trace Memory
STE	1	STart Experiment
STT	1	STart Trace
SOT	1	StOp Trace
STP	1	Start reading Traced Pulses
STC	1	Start reading Traced Configuration packets

Description:
This packet is used for global settings in the FPGA and for controlling its behaviour. Most importantly, it contains a set of control flags that control the execution of an experiment and the handling of traced pulses and configuration data.
Handling:
The above control flags are directly stored in hold registers that are connected to the correspondent inputs of the playback and trace control modules.

2.10.2.4 HICANN Configuration Data

Payload Type: `0x2A1B`

Format:



Field	Length [Bits]	Description
Des	3	specifies HICANN address for the packet
Tag	1	specifies ARQ tag on HICANN
HICANN Data	49	HICANN configuration data

Description:
This frame type is used for providing a single packet of raw HICANN configuration data to the ARQ modules in the FPGA.

Handling:
HICANN configuration data is forwarded to the HICANN ARQ module.

2.10.2.5 Sideband Data

For low-level control of the FPGA, UDP packets with 32bit payload have been defined that are sent to a specific UDP target port on the FPGA. Two types are defined, system time counter start/stop trigger, and FPGA reset and reprogramming. These are defined in the following.

System Time Counter Control

Port: 1800

Format:

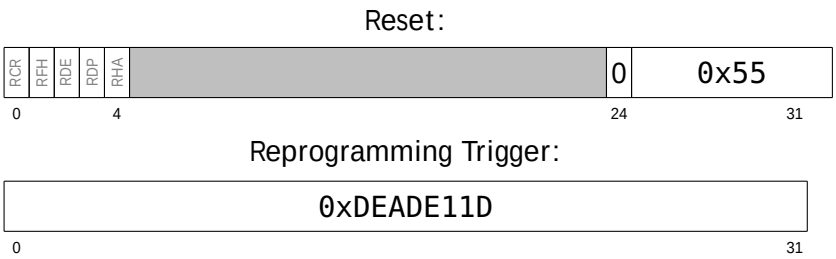
Field	Length [Bits]	Description
TCC	1	enable signal for system time counter (active high)

Description:
The TCC flag in the packet is stored in a register in the FPGA whose output enables counting of the system time counter in the FPGA. If the register is activated, i.e. set from 0 to 1, the system time counter is reset to 0, a trigger pulse on the sys_start output is generated to start the system time counters on the HICANNs and the FPGA system time counter starts running. If the TCC flag is set back to 0, the FPGA system time counter stops.

FPGA Soft Reset and Reprogramming

Port: 1801

Format:



Field	Length [Bits]	Description
RCR	1	reset of core logic (active high)
RFH	1	reset of FPGA-HICANN interfaces (active high)
RDE	1	reset of DDR3 frame buffer memory (active high)
RDP	1	reset of DDR3 pulse memories (active high)
RHA	1	reset of HICANN-ARQ module (active high)

Description:
Each reset flag in the reset packet is stored in an internal FPGA register. Each register output is merged with the global reset from the external reset pin via an OR-gate and connected to the reset input of the corresponding modules.

The reprogramming trigger packet stops operation of the FPGA immediatly and induces a re-load of the FPGA firmware from the external Flash memory.

2.10.3 Analog Readout

2.10.3.1 Host-to-Analog Readout Module USB protocol

The AnaRM is connected to the host PC via a USB connection. This section describes the commands that can be sent to the Flyspi of the AnaRM. There are 11 different commands for which their packet structure is depicted in the following figures.

Except for the commands `READBURST`, `WRITEBURST` and `WRITEOCPBURST`, all commands use a fixed packet size of 512 bytes. The following register diagrams only show the first 16 bytes of these packets and further bytes of the packet are irrelevant unless otherwise stated. The first row in each diagram contains the most significant bits of the data stream, continuing to less significant bits in the lower rows.

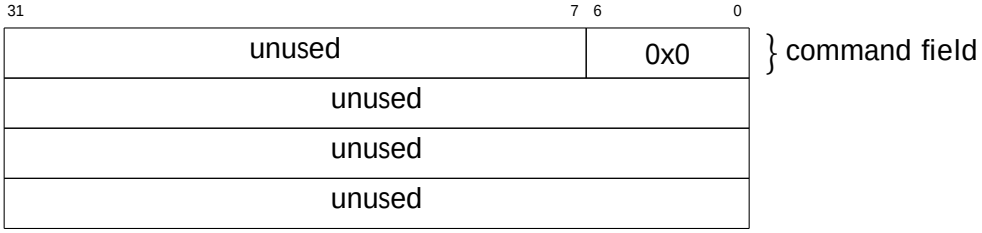


Figure 2.10.1: NOP (0x0)

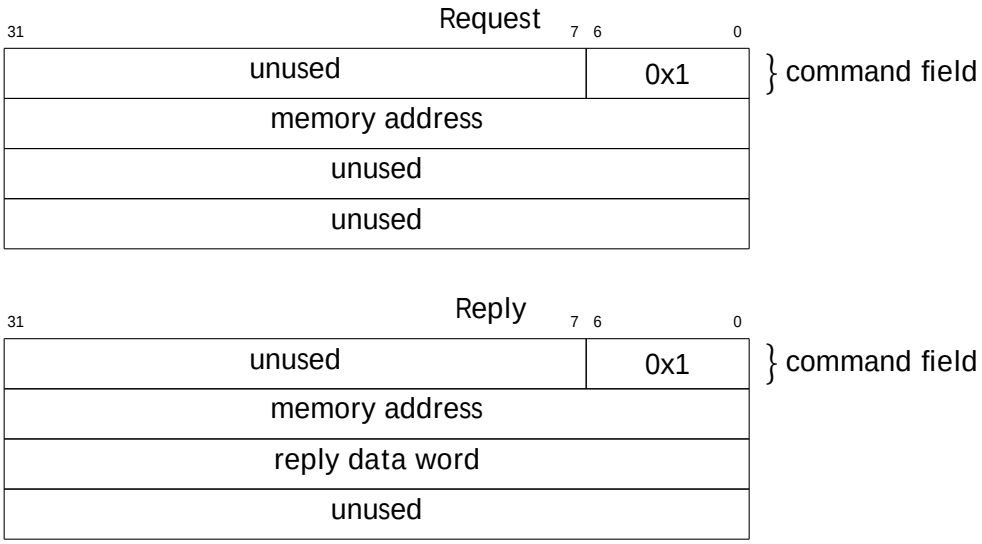


Figure 2.10.2: READMEM (0x1)

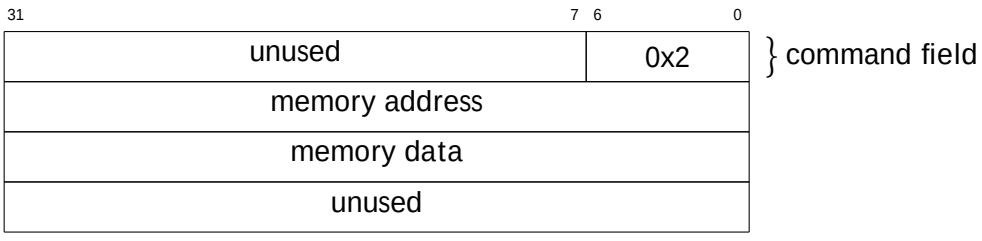


Figure 2.10.3: WRITEMEM (0x2)

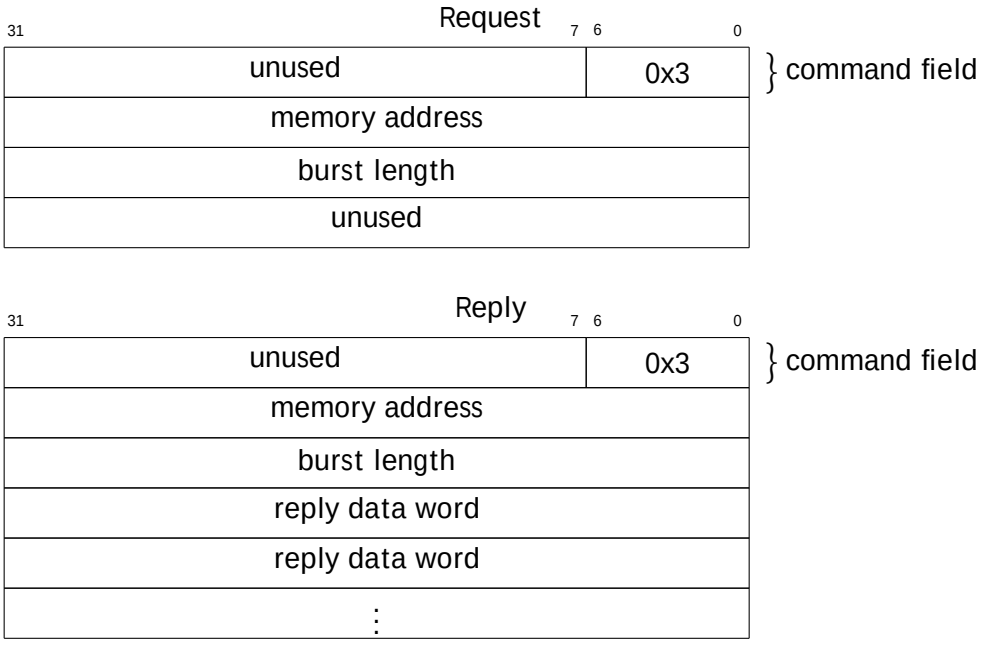


Figure 2.10.4: READBURST (0x3)

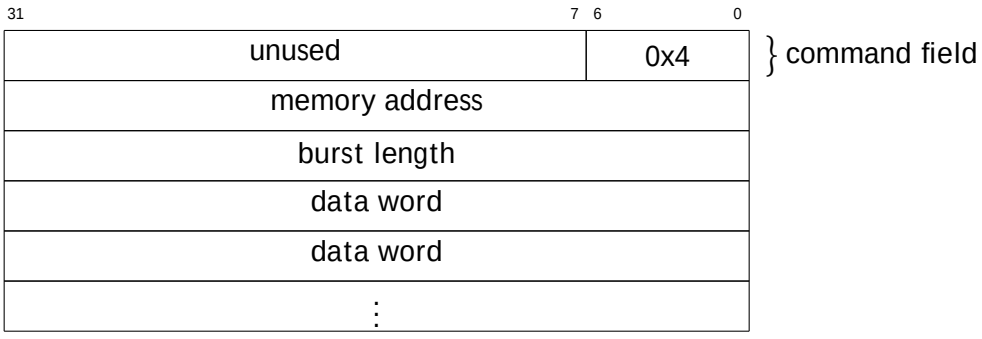


Figure 2.10.5: WRITEBURST (0x4)

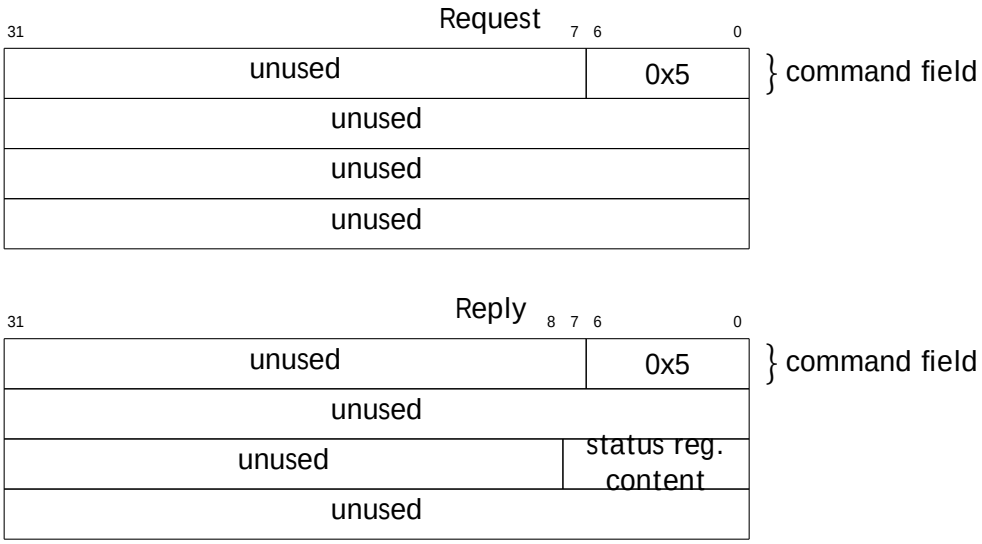


Figure 2.10.6: READSTATUS (0x5)

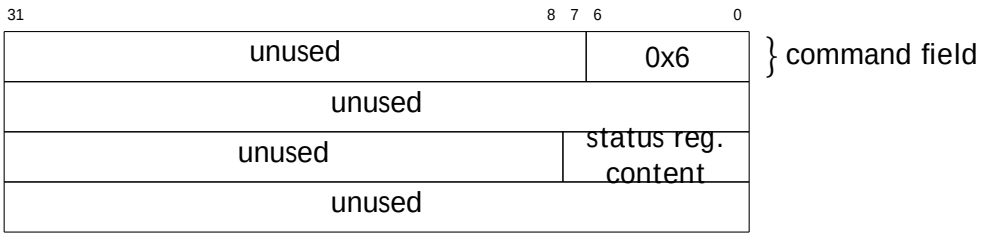


Figure 2.10.7: WRITESTATUS (0x6)

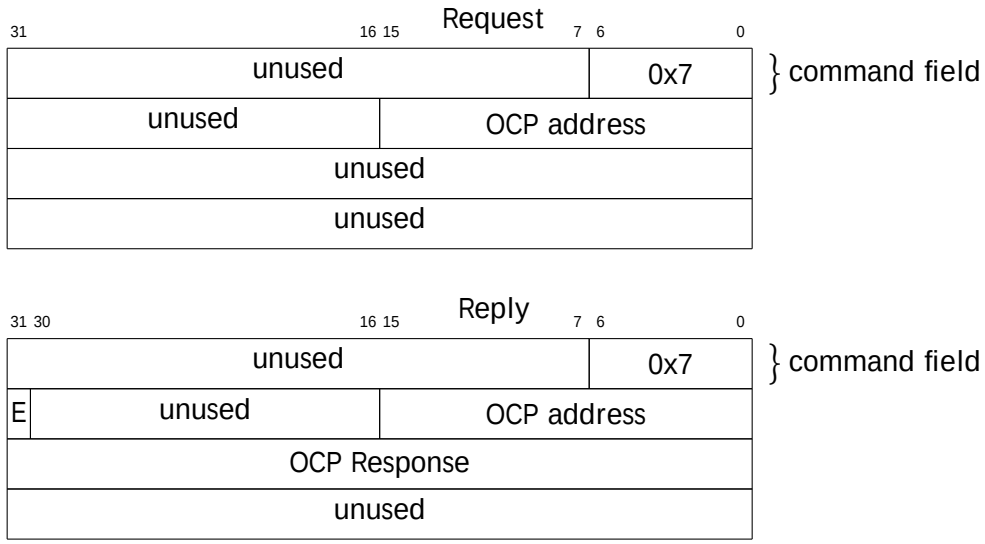


Figure 2.10.8: READOCP (0x7), here the field 'E' indictes whether the on-chip bus fabric FIFO was empty after the OCP request

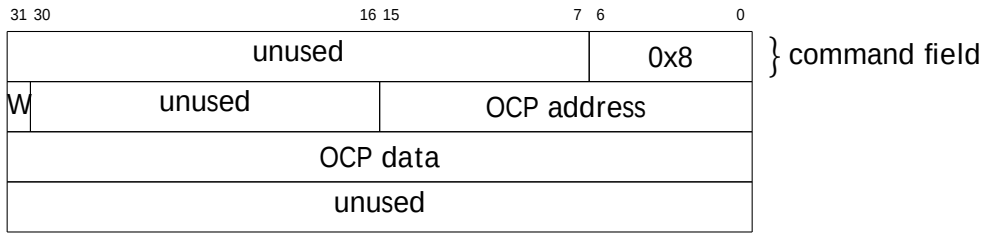


Figure 2.10.9: WRITEOCP (0x8), here the field 'W' selects whether the OCP request is a write request (1) or a read request (0).

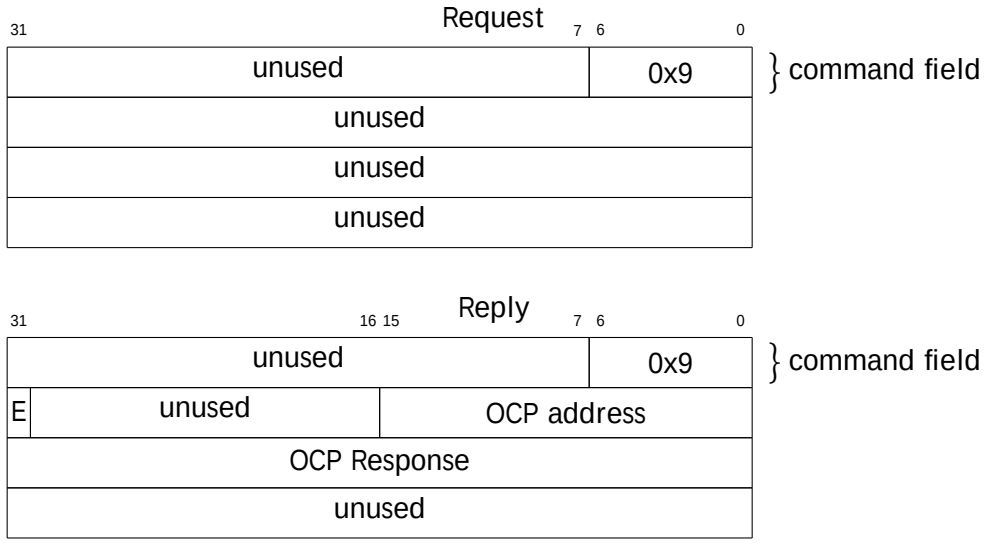


Figure 2.10.10: READOCPFIFO (0x9), here the field 'E' indicates whether the on-chip bus fabric FIFO was empty after the OCP request

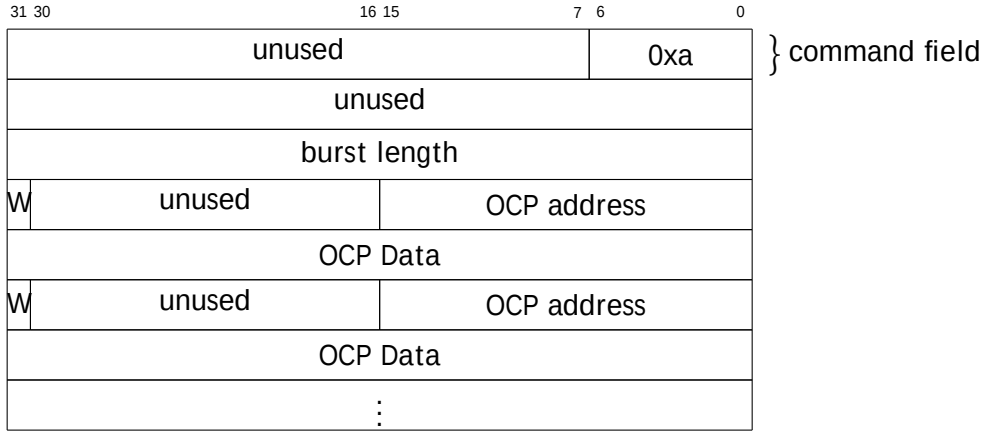


Figure 2.10.11: WRITEOCPBURST (0xa), here the field 'W' selects whether the OCP request is a write request (1) or a read request (0).

2.10.3.2 Pin assignment for analog input header

The input signal to the AnaFP is connected via an 8x2 male pin header with a pitch of 2.54 mm. The configuration of this pin header is depicted in figure 2.10.12. The individual signals can be selected via a multiplexer on the AnaFP whose configuration is described in the next subsection.

7	6	5	4	3	2	1	0
GND	6	5	GND	3	1	GND	TRG0
GND	7	GND	4	2	GND	0	TRG1

Figure 2.10.12: Configuration of the analog signal pin header on the AnaFP

2.10.3.3 FPGA registers for ADC board configuration

Register 2.10.1: ADC BOARD CONFIGURATION (0x8000)

unused	power_enable	unused	MUX0	unused	MUX1	MUX2	unsued
31 25	24	23 16	15 14	13 12	11 10	9 8	7 0
unused	0	unused	0 0	unused	0 0	0 0	unused

MUX0 switches between analog channels 0, 1 and 2. Both 0 means Mux 0 is in high output impedance mode.

MUX1 switches between analog channels 3, 4 and 5. Both 0 means Mux 1 is in high output impedance mode.

MUX2 switches between analog channels 6 and 7 and local ground input. Both 0 means Mux 2 is in high output impedance mode.

power_enable switches linear power regulators on the analog frontend board.

2.10.3.4 FPGA registers for Fast ADC controller

Successive registers starting at on-chip bus address `0x3000`.

7	6	5	4	3	2	1	0
unused				startaddr[27:24]			
startaddr[23:16]							
startaddr[15:8]							
startaddr[7:0]							
unused				endaddr[27:24]			
endaddr[27:24]							
endaddr[23:16]							
endaddr[15:8]							
endaddr[7:0]							
ADCPD	ADCRS	M1	M0	CB1	CB0	HI1	HI0
unsused			TRCH	TREN	TRSN	STOP	STRT

ADCPD **Invertedly** connected to physical power-down pin of ADC.

ADCRS **Invertedly** connected to physical reset pin of ADC.

M1 Controls in which order the single words that are sampled from the double data rate bus of the ADC are stored in the internal registers.

M0 Controls whether the clock phases that are sampled via the IDDR2 clock receiver should be written into the bit positions directly after the ADC data that are written to memory.

CB1 Controls the bit value that is applied in the first phase of the clock output to the ADC that is generated with an ODDR2.

CB0 Controls the bit value that is applied in the second phase of the clock output to the ADC that is generated with an ODDR2.

HI1 Most significant bit that will be written to memory together with a single ADC sample.

HI0 Second-to-most significant bit that will be written to memory together with a single ADC sample.

TRCH Selects one of two trigger channels that are physically connected to the board.

TREN Selects whether the FPGA listens for a trigger signal at its input pins. 1=enabled.

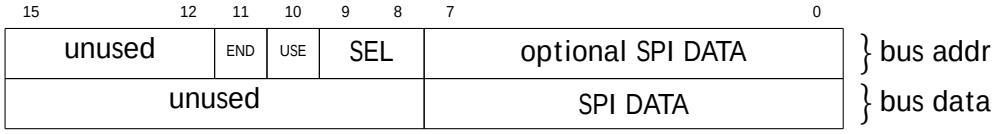
TRSN Selects whether the FPGA should only trigger once and then not listen for trigger signals any longer. 1=single trigger mode.

STOP Asynchronously stops an ADC sampling run. Normally the sampling run stops when `endaddr` is reached.

STRT Asynchronously starts an ADC sampling run. If the sampling run stops because the `endaddr` has been reached and this signal is still enabled, the controller will start another sampling run.

2.10.3.5 FPGA packet format for SPI-based ADC controller

The SPI configuration to the ADS6125 can according to the part's data sheet can be set via on-chip bus commands to the FPGA. The module uses only the lower 8 bits of the bus data field but can make use of the lower 8 bits of the address field.



USE selects whether the optional SPI DATA field in the lower 8 bits of the bus address field should be prepended to the SPI DATA field in the lower 8 bits of the bus data field.

END defines whether the SPI chip select signal should be de-asserted after the SPI transfer. If the signal is set to 0, multiple requests to this bus address can be combined to build larger concatenated SPI transfers.

SEL Controls which of four chip select signals should asserted during the SPI transfer. One module instantiation of the on-chip-bus-to-SPI converter in the FPGA firmware can control up to four different chips via SPI.

2.10.3.6 FPGA bus base addresses

Address	Module name
0x1000	ADC configuration via SPI
0x3000	Fast ADC controller
0x8000	Analog readout board controller

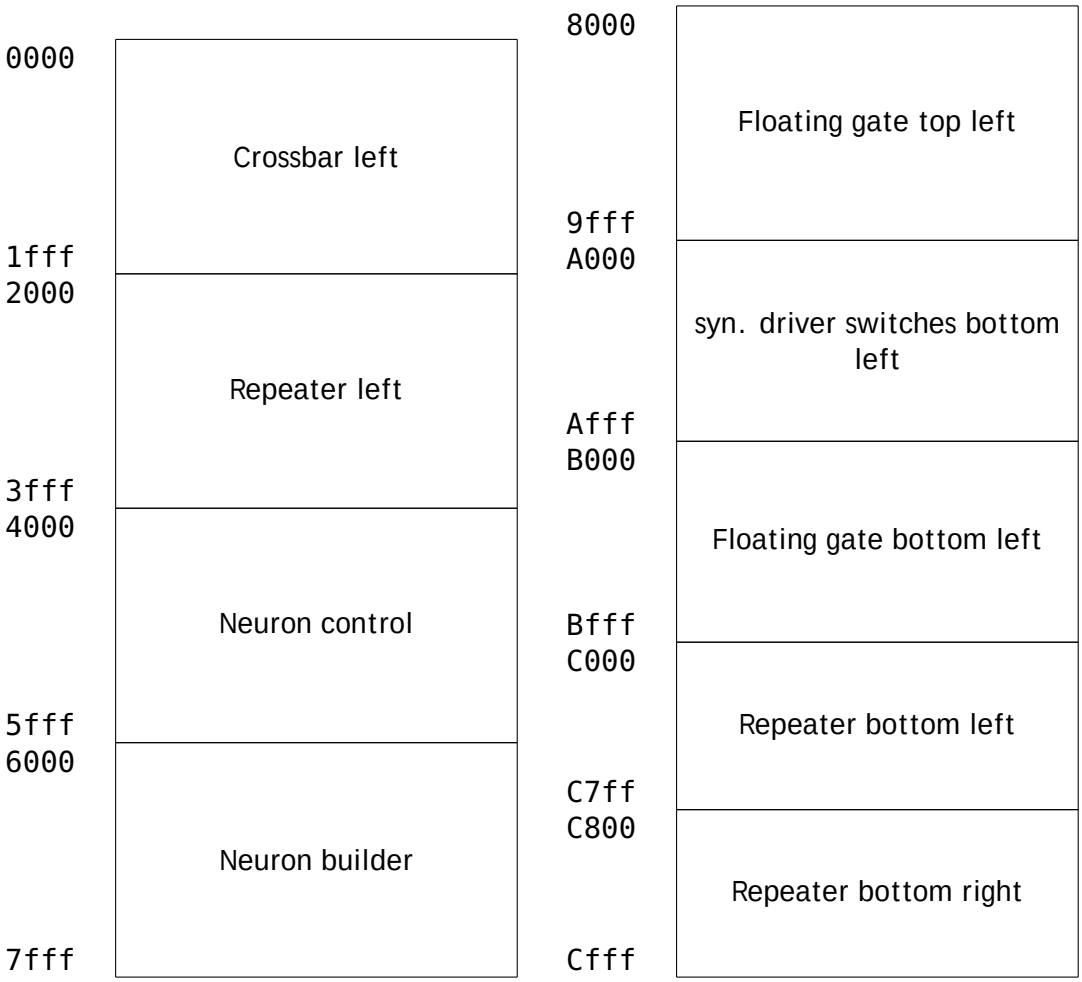


Figure 2.10.13: Address map of HICANN configuration registers (Part 1 of 2).

2.10.4 HICANN Configuration Registers

Figures 2.10.13 and 2.10.14 given an overview of the address space of the internal hicann bus. Not all addresses in the designated spaces are necessarily valid. Refer to the documentation of the module in the following sections for the actual used addresses.

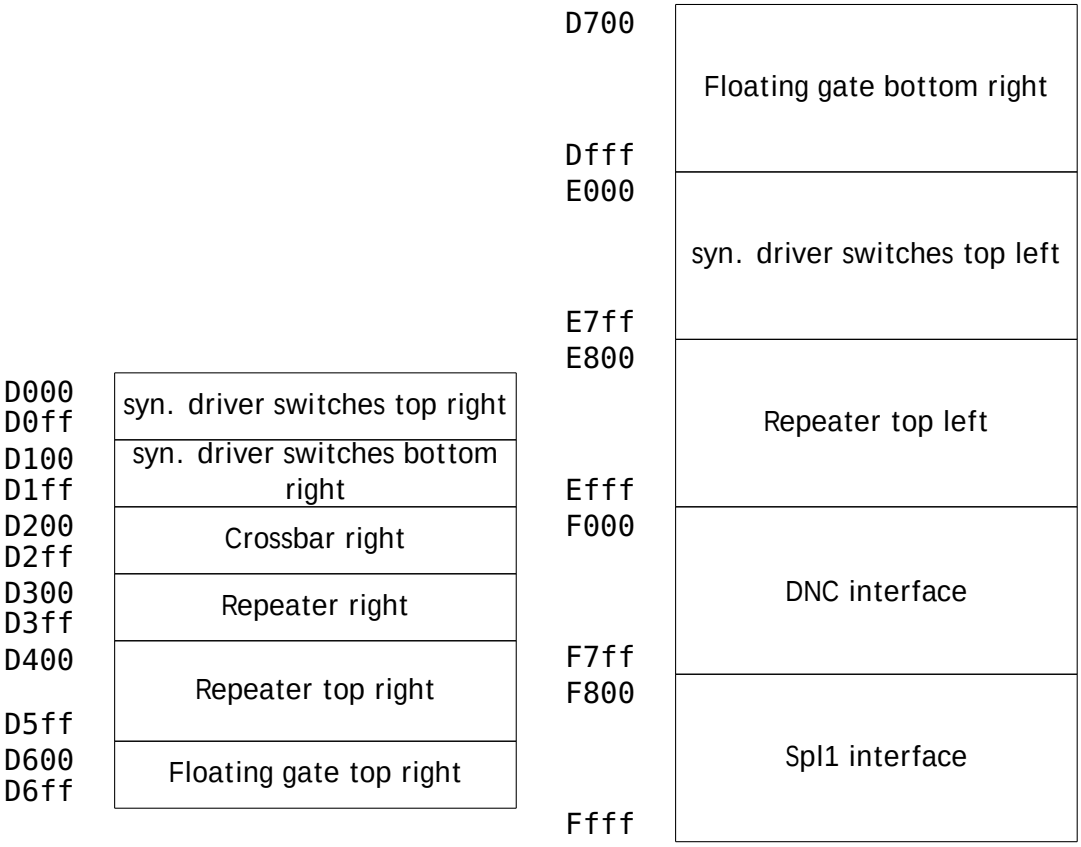


Figure 2.10.14: Address map of HICANN configuration registers (Part 2 of 2).

2.10.4.1 Hicann SRAM controller

source	name	description
dw	data width	width of sram data bus (min 8, max 32)
aw	address width	width of sram address bus+1, i.e. maxint(log_2(sram depth))
tc	timing control	width of timing control field (2^{tc} max delay in clk units)
numinp	number of inputs	number of input ports (with dw width)
numoutp	number of ouptuts	number of output ports (with dw width)
cnfgadr	conifg address	address bit selecting configuration registers, below is the IO-port address space
resval	reset value	output port register array reset value

(a) Parameters of the SRAM controller module. These parameters can only be controlled a compile time during Verilog instantiation. They are listed here because the SRAM controller is instantiated multiple times on the HICANN with different parameter settings for these values.

cnfgadr	cnfgadr-1:0	name	data bits	description
1	0	trd	tc-1:0	read delay, data valid after enable
1	1	tsu	7:4	setup time for address or data before enable -1, i.e. setting tsu=0 results in a setup time of 1
1	1	twr	3:0	enable pulse width -1
0	IO port	dw-1:0	IO	read/write IO data

(b) Timing control registers of the SRAM controller

Table 2.10.2: SRAM controller configuration.

2.10.4.2 Hicann neuron builder

name	value
dw	25
aw	10
numinp	3
numoutp	2
cnfgadr	'h10
resval	0

Table 2.10.3: SRAM controller settings in neuronbuilder.sv

adr	name	r/w	bit assignment [msb:lsb]
0	out1	r/w	n_resetb,spl1_resetb,cc.bigcap,cc.slow,cc.fast
1	out2	r/w	cc.aout_en,cc.aoutselect
2	inp	r	constant 1 (chip version)

Table 2.10.4: SRAM controller static registers in neuronbuilder.sv

name	bits	funciton
n_resetb	2	neuron reset top block=[0] , bottom block=[1]
spl1_resetb	1	neuron spl1 output reset
cc.bigcap	2	neuron control bits (see subsubsection 2.3.4.4)
cc.slow	6	neuron control bits (see subsubsection 2.3.4.4)
cc.fast	6	neuron control bits (see subsubsection 2.3.4.4)
cc.aout_en	2	enable 50 Ohm output buffer top=[0], bottom=[1], when top buffer is disabled, biasTG is open
cc.aoutselect	20	select analog output source top=[9:0], bottom=[19:10] (see subsubsection 2.3.4.4)

Table 2.10.5: Content of the static control and reset registers in neuronbuilder.sv

name	bits	address
spl1	6	Out of every group of 4 denmems, bottom left denmem address is *adr* (mod 0), top left is *adr* (mod 1), bottom right is *adr* (mod 2) and top right is *adr* (mod 3)
nmem top	8	3+4i
nmem bottom	8	3+4i
nb_vertical	1	2i
nb_fireen_top	1	2i + 1
nb_fireen_bottom	1	2i

Table 2.10.6: Content of the bit lines controlled by the SRAM controller in neuronbuilder.sv

2.10.4.3 Hicann denmem configuration

slow	fast	divisor
0	0	32
0	1	8
1	0	160
1	1	40

(a) I_{radapt}

slow	fast	divisor
0	0	3
0	1	1
1	0	27
1	1	9

(b) $I_{gladapt}$

slow	fast	divisor
0	0	3
0	1	1
1	0	27
1	1	9

(c) I_{gl}

Table 2.10.7: Scaling factors of the current mirrors for three different currents.

Number	function
0	inout
1	inout
2	inout
3	interconnect neuron pair
4	activate firing of left neuron
5	enable fire input of left adjacent neuron + membrane interconnection
6	enable fire input of right adjacent neuron + membrane interconnection
7	activate firing of right neuron

(a)

bit 2	bit 1	bit 0	aout right	aout left	currentin right	currentin left
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	1(v1 0)	0(v1 1)
1	0	1	1	0	1(v1 0)	0(v1 1)
1	1	0	0	1	0(v1 1)	1(v1 0)
1	1	1	0	0	0(v1 1)	1(v1 0)

(b) Meaning of the inout control bits

Table 2.10.8: SRAM control bits for denmem interconnect and analog output

2.10.4.4 HICANN analog output configuration registers

bit	op0	op1	description
0	Vctrl0<0>	Vctrl0<1>	Dll control voltage of row 0 syndriver (top/bottom)
1	prebuf<0>	prebuf<1>	output of inner left syndrivers
2	ft<0>	nc	fire line of neuron 0
3	fg_out<0>	fg_out<1>	Floating gate output of left arrays
4	n_out<1>	n_out<1>	Membrane readout even denmems top
5	n_out<3>	n_out<3>	Membrane readout even denmems bottom
6	n_out<0>	n_out<0>	Membrane readout odd denmems top
7	n_out<2>	n_out<2>	Membrane readout odd denmems bottom
8	nc	nc	not connected
9	fg_out<1>	fg_out<3>	Floating gate output of right arrays

Table 2.10.9: Analog output multiplexer configuration

command	val	function
read	0	sets floating gate array to readout mode and connects a dedicated cell to the analog output until another command is executed
writeUp	1	programs one floating gate line up
writeDown	2	programs one floating gate line down
getNextFalse	3	gets column address of next cell which did not reach the programmed value during a programming process
stimulateNeurons	4	stimulates a neuron using the values in one RAM bank as input; stops at last ram address.
stimulateNeurons Continuous	5	stimulates a neuron using the values in one RAM bank as input continuously until another instruction is executed

Table 2.10.10: Floating gate control instructions

2.10.4.5 HICANN floating gate controller instructions

2.10.4.6 HICANN merger tree configuration

name	function
enable	activate merger function, if 0, the merger acts like a static pipelined multiplexer
select	if enable=0, selects the input (see Fig.2.3.15) for input assignement
slow	set slow=1 for each merger connecting to a driving repeater. This ensures the output rate is limited to clk/2 which is necessary for spl1 data

Table 2.10.16: Merger tree configuration signals

parameter	width	function
maxcycle	8	Maximum number of cycles for programming
currentwritetime	6	length of a current write pulse
voltagewritetime	6	length of a voltage write pulse
readtime	6	Time waited for stable value during read in write process
accleratorstep	6	Number of cycle after which intern write time is doubled
pulselength	4	Multiplicator between in tern controller clock and slow HICANN clock

Table 2.10.11: Floating gate parameters

parameter	bits	description
fg_bias	[3:0]	Bias for upper floating gate programming voltage
fg_biasn	[7:4]	Bias for source followers. Maximum value if set to zero
pulselength	[11:8]	Clock cycle multiplicator
groundVm	[12]	shorts V_m to ground
calib	[13]	activate calibration mode

Table 2.10.12: Bit Configuration of biasRegister

adr	name	function [bit15-8]	[bit7-0]
0	enable	nc l2 l1_1 l1_0 l0_3 l0_2 l0_1 l0_0	bg_7-0
1	select	nc l2 l1_1 l1_0 l0_3 l0_2 l0_1 l0_0	bg_7-0
2	slow	nc l2 l1_1 l1_0 l0_3 l0_2 l0_1 l0_0	bg_7-0
3	dncloopb	select dnc for dnc merger 7-0	loopback enable 7-0
4	randomreset	background generators random 7-0	reset_n 7-0
5	phase	nc	input sample reg clock phase 7-0
6	seed	bg common seed 15-8	7-0
7-14	period	bg period 15-8	7-0
15	slowenablednc	slow for dnc merger 15-8	enable for dnc merger 7-0
16	bg neuron	bg1 neuron number 13-8	bg0 neuron number 6-0
17	bg neuron	bg3 neuron number 13-8	bg2 neuron number 6-0
18	bg neuron	bg5 neuron number 13-8	bg4 neuron number 6-0
19	bg neuron	bg7 neuron number 13-8	bg6 neuron number 6-0

Table 2.10.17: Merger tree routing configuration

parameter	bits	description
maxCycle	[7:0]	Maximum number of programming cycles.
readTime	[13:8]	Time waited until value to be read is stable
acceleratorStep	[19:14]	Number of cycles after which the write time is doubled
voltageWriteTime	[25:20]	Length of a write pulse
currentWriteTime	[31:26]	Length of a write pulse

Table 2.10.13: Bit Configuration of operationRegister

parameter	bits	description
lineNumber	[4:0]	line number (0:23) in code
columnNumber	[12:5]	column number (0:128) in code
bankNumber	[13]	Ram bank
instruction	[16:14]	Instruction for programming machine

Table 2.10.14: Bit Configuration of addressInstructionRegister

2.10.4.7 HICANN background event generator configuration

name	function
random	1: poisson event interval distribution, 0: fixed period between events
period	random=0: event period in sysclk (4ns) cycles, random=1: mean isi
seed	seed value for random generators (one common seed value for all generators)
reset_n	reset_n=0: disable event generators, reset_n=1: load seed and start event generation
my_nn	neuron number that is emitted by the background event generator

Table 2.10.18: Background event generator configuration signals

parameter	bits	description
slaveAnswer	[7:0]	Column number of wrong programmed cell
busy	[8]	High when state machine is active
error	[9]	Active, when a cell did not reach its value

Table 2.10.15: Bit Configuration of slaveAnswerData

2.10.4.8 HICANN reapeater SRAM controller configuration

name	value
dw	8
aw	8
numinp	13
numoutp	13
cnfgadr	'h20
resval	0

Table 2.10.19: SRAM controller configuation in the Repeater Controller module

name	normal repeater function	spl1 sending repeater function
recen	activates repeater	activates ext-driver, receiver is always on
dir	0:int->ext, 1:ext->int	input: same as standard repeaters, output: 1:activates int-driver
touten	activates test output	same
tinen	activates test input	connects spl1 output from neuroncontrol
ren,len	FEXT compensation	

Table 2.10.20: Function of the static configuration registers of the SRAM controller in the Repeater Controller module

adr(hex)	name	r/w	bit assignment [7:0]
0:repmax-1	repeater SRAM cells	rw	touten, tinen, recen, dir, ren[1:0], len[1:0]
80	config	w	fextcap[1:0], drvresetb, dllresetb, start TDI [1:0], start TDO [1:0]
80	status	r	reserved [7:2],TDI full flag [1:0]
81:86	test data out	w	test data out (TDO) channel 0:
81	entry 0 [7:0]	w	time [7:0]
82	entry 0 [15:8]	w	neuron number[5:0],time[9:8]
83	entry 1 [7:0]	w	time [7:0]
84	entry 1 [15:8]	w	neuron number[5:0],time[9:8]
85	entry 2 [7:0]	w	time [7:0]
86	entry 2 [15:8]	w	neuron number[5:0],time[9:8]
81:86	test data in	r	test data in (TDI) channel 0
87:9c	test data out	w	test data out channel 1
87:9c	test data in	r	test data in channel 1
A0:A1	sram timing	rw	standard sram timing registers

Table 2.10.21: Address map of the static configuration registers of the SRAM controller in the Repeater Controller module

2.10.4.9 HICANN DNC interface and Layer 2 circuit configuration

component	address	configuration value
dnc_if	Write 0	data[1:0] := init_ctrl
		data[2] := dc_bal_ctrl
		data[3] := proto_cfg
		data[4] := proto_plsl
		data[5] := 500MHz mode
		data[6] := crc_reset
		data[7] := channel_reset
	Read 0	data[15:8] := CRC error counter
		data[7:0] := Interface state
spl1_if	Write 0	data[7:0] := layer 1 bus connection enable
		data[15:8] := layer 1 bus direction
		data[16] := use timestamp of events
	Write 1	data[0] := reset expired / ignored event counters
	Read 0	Expired events layer 1 bus 0-3
	Read 1	Expired events layer 1 bus 4-7
	Read 2	Ignored events layer 1 bus 0-3
	Read 3	Ignored events layer 1 bus 4-7

Table 2.10.22: Configuration address space of layer 2 components

2.10.4.10 Digital Synapse Control

The functionality and implementation of this module is described in Section 2.3.4.5. There are two instances of DSC at the top and bottom of the chip. The memory range of the top block starts at address 0. The bottom block start at address 0x8000. All addresses given in this section are relative to these offsets. Both blocks are accessed using tag 1 of the internal bus.

Reading and Writing of Synaptic Weights and Decoder Addresses

Register 2.10.2: SYNIN (0x4200)

31 28	27 24		7 4	3 0	
W_0	W_1	· · ·	W_6	W_7	0x4200
W_8	W_9	· · ·	W_{14}	W_{15}	0x4201
W_{16}	W_{17}	· · ·	W_{22}	W_{23}	0x4202
W_{24}	W_{25}	· · ·	W_{30}	W_{31}	0x4203

Register 2.10.3: SYNOUT (0x4300)

31 28	27 24		7 4	3 0	
W_0	W_1	· · ·	W_6	W_7	0x4300
W_8	W_9	· · ·	W_{14}	W_{15}	0x4301
W_{16}	W_{17}	· · ·	W_{22}	W_{23}	0x4302
W_{24}	W_{25}	· · ·	W_{30}	W_{31}	0x4303

Register definitions The 128 bit registers SYNIN (Register 2.10.2) and SYNOUT (Register 2.10.3) are data registers used for the transfer of synaptic weights and decoder addresses into and out of the synapse array. Weights and decoder addresses are 4 bit in size and arranged in the registers from low to high.

Register 2.10.4: CREG (0x4000)

31	30	29	28	27	26 24	23 16	15 8	7	6	5	4	3 0
reserved	idle	sca	scc	without_reset	sel	lastadr	adr	reserved	newcmd	continuous	encr	cmd
-	0	1	1	0	0	0	0	-	0	0	0	IDLE

Opcode	Value (binary)	Description
IDLE	0000	Do nothing
START_READ	0111	Open one row for reading of synaptic weights
READ	0001	Read weights from one column set into SYNOUT register
WRITE	0011	Write weights from SYNIN register into one column set
RST_CORR	1010	Reset correlation capacitors according to the SYNRST register
START_RDEC	0010	Open one row for reading of decoder addresses
RDEC	0110	Read decoder addresses from one column set into SYNOUT register
WDEC	0101	Write decoder addresses from SYNIN register into one column set
CLOSE_ROW	1001	Close current row after it was opend with START_READ or START_RDEC
AUTO	0100	Start the automatic weight update process

Table 2.10.24: Valid opcodes for the cmd field of Register 2.10.4.

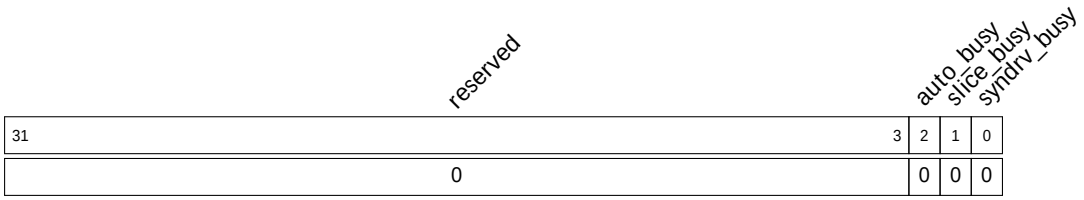
The 32 bit control register CREG (Register 2.10.4) is used to trigger operations on the synapse array, such as writing weights or enabling STDP. The field cmd contains one of the operation codes defined in Table 2.10.24.

Register 2.10.5: CFGREG (0x4001)

reserved	wrdel	oedel	dllresetb	gen	predel	endel	pattern0.cc	pattern1.cc	pattern0.ca	pattern1.ca	pattern0.ac	pattern1.ac	pattern0.aa	pattern1.aa
31 29	28 27	26 23	22 21	19 16	15 12	11 8	7	6	5	4	3	2	1	0
-	2	2	0	0	15	15	0	0	0	0	0	0	0	0

The 32 bit configuration register CFGREG (Register 2.10.5) holds additional information for the operation of DSC. The fields wrdel, oedel, predel, and endel configure the timing of synapse array operations. The STDP evaluation patterns are given in fields pattern0 and pattern1. The bits from fields dllresetb and gen are directly given to the synapse array analog block.

Register 2.10.6: STATUS (0x4002)



The read-only 32 bit register STATUS (Register 2.10.6) provides monitoring information for the state of DSC. The bits are set to 1, if the automatic weight update controller is active (`auto_busy`), a synapse driver memory access in ongoing (`syndrv_busy`), or an synpase array access operation is ongoing (`slice_busy`).

Configuring for operation Reading and writing of synaptic weights and decoder addresses should be possible after reset with the default timings provided in Register 2.10.5. To improve performance, timing parameters can be set to smaller values until reading after a write returns incorrect data.

Access operations There are four use cases:

- Read synaptic weights
- Write synaptic weights
- Read decoder addresses
- Write decoder addresses

For all of them the assignment of 4 bit fields in Registers 2.10.2 and 2.10.3 to synapses in the array is controlled by the contents of Register 2.10.4: Field `CREG.adr` selects the row of the synapse array. Field `CREG.sel` selects the column assignment according to the following mapping from the register index i to the column index j:

$$s \leftarrow \texttt{CREG.sel} \tag{2.10.1}$$

$$i \in \{0, \ldots, 31\} \tag{2.10.2}$$

$$j \in \{0, \ldots, 255\} \tag{2.10.3}$$

$$j = f_{\text{colset}}(i, s) = \begin{cases} i + 8s & \text{for } 0 \leq i < 8 \\ i + 8s + 64 & \text{for } 8 \leq i < 16 \\ i + 8s + 128 & \text{for } 16 \leq i < 24 \\ i + 8s + 192 & \text{for } 24 \leq i < 32 \end{cases} \tag{2.10.4}$$

The columns addressed by one configuration of `CREG.sel`

$$\{f_{\text{colset}}(i, s = \texttt{CREG.sel}) \mid i \in \{0, \ldots, 31\}\} \tag{2.10.5}$$

are referred to as column set of this configuration.

All accesses are performed by writing an operation code from Table 2.10.24 to `CREG.cmd`. In order for hardware to execute the operation, `CREG.newcmd` must be set to 1.

For read operations the pre-charge of the SRAM bitlines is separated from the read access. This means, that a row has to be "opened" before reading and "closed" before switching to a new row. Writes can be performed irrespective of whether the row is open or not.

Writing synaptic weights In the simplest case a write requires only one operation with opcode `WRITE`.

1: `SYNIN` ← weight data W_i
2: `CREG.adr` ← row address r
3: `CREG.sel` ← column set selection s
4: `CREG.cmd` ← `WRITE`
5: `CREG.newcmd` ← 1
6: **repeat**
7: b ← `STATUS`
8: **until** $b = 0$

This will write the following weights of the synapse-array:

$$\text{SYNARRAY}[r][f(i,s)] \leftarrow W_i \quad \text{for} \quad 0 \leq i < 32 \tag{2.10.6}$$

Reading synaptic weights For reading it is necessary to first open the row using the `START_READ` operation and afterwards close it with `CLOSE_ROW`.

1: `CREG.adr` ← row address r
2: `CREG.sel` ← column set selection s
3: `CREG.cmd` ← `START_READ`
4: `CREG.newcmd` ← 1
5: **repeat**
6: b ← `STATUS`
7: **until** $b = 0$
8: `CREG.cmd` ← `READ`
9: **repeat**
10: b ← `STATUS`
11: **until** $b = 0$
12: W_i ← `SYNOUT`
13: `CREG.cmd` ← `CLOSE_ROW`
14: **repeat**
15: b ← `STATUS`
16: **until** $b = 0$

This will read the following weights from the synapse-array:

$$W_i \leftarrow \text{SYNARRAY}[r][f(i,s)] \quad \text{for} \quad 0 \leq i < 32 \tag{2.10.7}$$

Reading and writing decoder addresses Read and write operations on decoder addresses are identical to those on weights, but use different operation codes:

Operation	Delay worst case
WRITE	8 cycles
START_READ	34 cycles
READ	18 cycles
CLOSE_ROW	3 cycles

Table 2.10.25: Worst case latencies for synapse array operations.

- START_RDEC replaces START_READ
- RDEC replaces READ
- WDEC replaces WRITE

Operation timings DSC is clocked with the system-clock divided by four. So for a nominal system-clock frequency of 250 MHz, one clock cycle of DSC is 16 ns long. The interface to the synapse array is operated at this frequency.

DSC is connected to Tag 1 of the internal bus. This bus is fully pipelined and can take one read or write request in every cycle. The delay on this bus is 6 cycles for both the top and bottom controller [1]. A register access takes one cycle. Therefore, the total delay for a register access is $2 \cdot 6 + 1$ cycles. For operations on the synapse array the additional delay is given by Table 2.10.25 [2]. The delay can be configured by changing the timing parameters CFGREG.*del in the configuration register.

Reading and Writing of Synapse Driver Configuration Memory

The synapse driver configuration memory is read and written by an sramCtrl module. The memory is separated into three regions: config, predrv, and gmax. The memory can be read and written. Due to the pipelined, non-blocking bus interface to DSC, two read requests have to be sent in order to read a memory location. (Yes, I know this is stupid. I won't do it again.)

An address map of the configuration memory is shown in Figure 2.10.15.

Reading Reading synapse-driver configuration memory requires two consecutive reads on the same address. Only the second one will return valid data. The timing of the sramCtrl module is configured in its timing setup address section (Section 2.3.4.4).

1: $a \leftarrow$ configuration memory address
2: $t_{\text{syndrv}} \leftarrow$ duration of sramCtrl access
3: y $\leftarrow$ SYNDRV[a]
4: Wait(t_{syndrv})
5: y $\leftarrow$ SYNDRV[a]

[1] Measured from port pktin on the Tag1 FIFO adapter to the control register CREG.
[2] The time measured is the duration during which the STATUS.slice_busy signal is asserted.

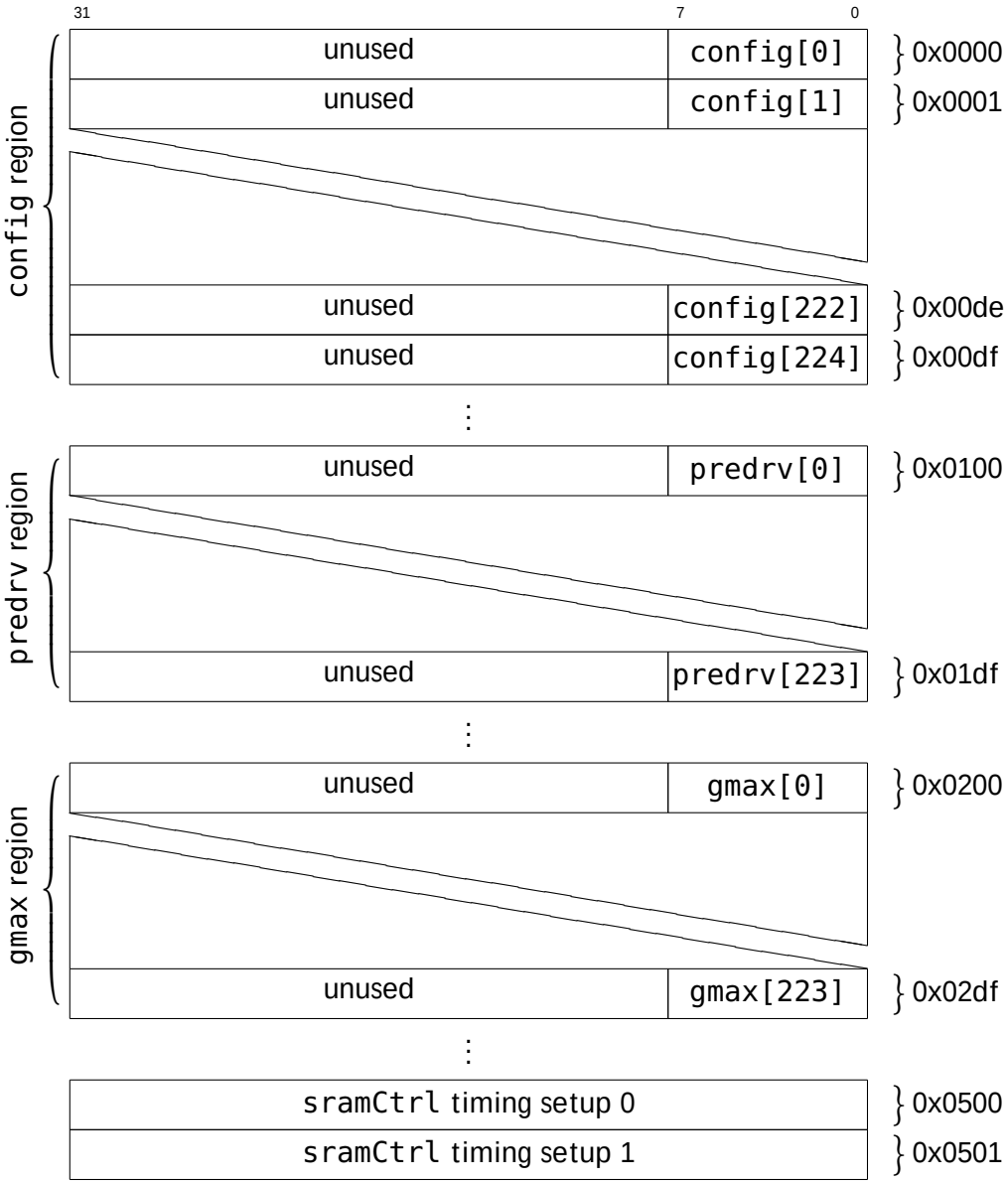


Figure 2.10.15: Memory map of the synapse driver configuration memory accessible through the `sramCtrl` module in DSC.

Bit mappings The syndriver-SRAM bits are mapped to the following syndriver control signals:

type	bit	name	function
config bottom	0	bot:senx	connect row to excitatory input of denmems
	1	bot:seni	connect row to inhibitory input of denmems
	3:2	bot:selgm	select V_{gmax} from global FGs
	4	topin	shortcut input with input from top neighbour syndriver
	5	locin	select input from L1
	6	en	enable this synapse driver
	7	enstdf	enable STP
config top	0	top:senx	connect row to excitatory input of denmems
	1	top:seni	connect row to inhibitory input of denmems
	3:2	top:selgm	select V_{gmax} from global FGs
	6:4	cap	select STDP capacitance (capacity = 8fF∗cap)
	7	dep	select STP mode: 0: facilitation, 1: depression
predrv bottom	3:0	preout0 (p0)	decoder bits [5:4] for bottom left synapses
	7:4	preout2 (p2)	decoder bits [5:4] for bottom right synapses
predrv top	3:0	preout1 (p1)	decoder bits [5:4] for top left synapses
	7:4	preout3 (p3)	decoder bits [5:4] for top right synapses
gmax bot	3:0	bot:Iout dac0	fraction of selected V_{gmax} used when row is excitatory
	7:4	bot:Iout dac1	fraction of selected V_{gmax} used when row is inhibitory
gmax top	3:0	top:Iout dac0	fraction of selected V_{gmax} used when row is excitatory
	7:4	top:Iout dac1	fraction of selected V_{gmax} used when row is inhibitory

The functionality of the short term plasticity circuit is documented in [59].

The four preout-signals are encoded from the upper two neuron address bits, na[5:4], and the static global enable bits gen[1:0]. The mapping, depending on the preout SRAM bits, preout$x$[3:0] (or short px[3:0], with $x \in \{0, \ldots, 3\}$), is as follows (only the case preout=1 is listed):

preout	condition for preout=1
0	p0[2] = na[4] & p0[0] = na[5] & p1[0] = gen[0] & p1[2] = gen[1]
1	p1[1] = na[4] & p1[3] = na[5] & p0[1] = gen[0] & p0[3] = gen[1]
2	p2[2] = na[4] & p2[0] = na[5] & p3[0] = gen[0] & p3[2] = gen[1]
3	p3[1] = na[4] & p3[3] = na[5] & p2[1] = gen[0] & p2[3] = gen[1]

This table is interpreted as follows: a preout signal to the synpase array is active if the logical condition stated in the respective line is true. The global enable bits gen[0:3] are located in Reg. 2.10.5.

The global arrangement is as follows: left syndriver use databus bits 0:7, right use 8:15. Left are on even addresses, right on odd; row zero is at the center of the chip. Due to an mapping error the bit ordering of the syndriver data bus is swapped: left syndrivers (even addresses) use bit 8:15 of the ocp data bus (but in reverse order), right syndrivers 0:7 respectively.

Use of STDP

For the realization of STDP, DSC covers three use cases:

- Reading correlation information from the STDP circuit in the synapse.
- Resetting correlation information in the synapse.
- Automatic weight update during a network experiment.

The automatic weight update state machine iterates over the array, reads weights, evaluates correlation information, computes new weights, and writes them back to the synapses. The evaluation of weights is controlled by the two 4 bit evaluation configurations provided in `CFGREG.pattern0` and `CFGREG.pattern1`. Each of these evaluations provides a single result bit C^0 and C^1. Together they are used to compute a new weight w' for current weight w using a look-up table $L_w^{C^0C^1}$:

$$w' \leftarrow L_w^{C^0C^1}. \tag{2.10.8}$$

Register 2.10.7: LUT (0x4100)

31 28	27 24		7 4	3 0	
L_0^{01}	L_1^{01}	...	L_6^{01}	L_7^{01}	0x4100
L_8^{01}	L_9^{01}	...	L_{14}^{01}	L_{15}^{01}	0x4101
L_0^{10}	L_1^{10}	...	L_6^{10}	L_7^{10}	0x4102
L_8^{10}	L_9^{10}	...	L_{14}^{10}	L_{15}^{10}	0x4103
L_0^{11}	L_1^{11}	...	L_6^{11}	L_7^{11}	0x4104
L_8^{11}	L_9^{11}	...	L_{14}^{11}	L_{15}^{11}	0x4105

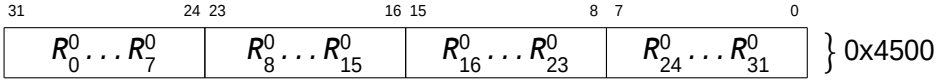
Register definitions The 192 bit register LUT (Register 2.10.7) contains three look-up tables (L^{01}, L^{10}, L^{11}) for the automatic weight update mechanism.

Register 2.10.8: SYNCORR (0x4400)

31 24	23 16	15 8	7 0	
$C_0^0 \ldots C_7^0$	$C_8^0 \ldots C_{15}^0$	$C_{16}^0 \ldots C_{23}^0$	$C_{24}^0 \ldots C_{31}^0$	0x4400
$C_0^1 \ldots C_7^1$	$C_8^1 \ldots C_{15}^1$	$C_{16}^1 \ldots C_{23}^1$	$C_{24}^1 \ldots C_{31}^1$	0x4401

The 64 bit read-only register SYNCORR (Register 2.10.8) is a data register for the correlation readout. It is written in read operations started by `CREG.cmd` = READ, for which `CREG.encr` is set to 1. Bits C_i^0 are produced by evaluation with configuration from code, while C_i^1 are produced using pattern 1.

Register 2.10.9: SYNRST (0x4500)



The 32 bit register SYNRST (Register 2.10.9) controls for which columns in a column set the analog accumulation should be reset. The polarity is active-high.

Configuring for operation Reading and writing of synaptic weights must be correctly configured (Section 2.10.4.10).

The digital configuration consists of setting registers LUT (Register 2.10.7), SYNRST (Register 2.10.9), CFGREG (Register 2.10.5), and CREG (Register 2.10.4).

Operations

Reading correlation The SYNCORR (Register 2.10.8) register is updated with correlation data for read operations (Section 2.10.4.10) where `CREG.encr` = 1, `CREG.sca` = 1, and `CREG.scc` = 1.

> **Note:**
> Analog storage times for the accumulated correlation information in the synapse, which are read out by this operation, are on the order of micro- to milliseconds. The user has to assure, that the correlation read operation is executed soon enough after the spike pairings that caused the correlation change.

Resetting correlation Correlation information in the synapse can be reset with the RST_CORR command.

1: SYNRST ← reset bits R_i
2: `CREG.adr` ← row address r
3: `CREG.sel` ← column set selection s
4: `CREG.without_reset` ← 0
5: `CREG.cmd` ← RST_CORR
6: `CREG.newcmd` ← 1
7: **repeat**
8: b ← STATUS
9: **until** $b = 0$

This will reset the following synapses in the synapse-array:

$$\text{Reset}(\text{SYNARRAY}[r][f(i,s)]) \quad \text{if} \quad R_i = 1 \text{ for } 0 \leq i < 32 \tag{2.10.9}$$

Automatic weight update for STDP The automatic weight update controller can be operated in continuous or non-continuous mode. This is controlled by `CREG.continuous`

(Register 2.10.4). In non-continuous mode (CREG.continuous = 1) the controller iterates from row CREG.adr to CREG.lastadr once. Otherwise it returns to CREG.adr after processing the last row.

Starting automatic weight updates:

1: SYNRST ← reset bits $R_i = 1\ \forall i$
2: CREG.adr ← first row address r_0
3: CREG.lastadr ← last row address r_1
4: CREG.encr ← 1
5: CREG.without_reset ← 0
6: CREG.sca ← 1
7: CREG.scc ← 1
8: CREG.sel ← column set selection s
9: CREG.continuous ← operation mode m
10: CREG.cmd ← AUTO
11: CREG.newcmd ← 1

While automatic weight updates are in progress no other access must be performed. The STATUS.auto_busy (Register 2.10.6) register is set to 1 during automatic weight updates. If continuous mode is used ($m = 1$), automatic weight updates have to be explicitly stopped:

1: CREG.continuous ← 0
2: **repeat**
3: b ← STATUS
4: **until** $b = 0$

2.10.5 JTAG Access

Both HICANN and FPGA provide backdoor JTAG access. One FPGA and eight HICANNs within the according reticle are connected in a JTAG chain. Available JTAG commands and their functionality are summarized in this section.

2.10.5.1 HICANN JTAG Access

Table 2.10.26 presents all JTAG registers in the HICANN. Registers marked with 'R' or 'W' (read or write) requires an amount of data shifts to get or set the internal registers. Registers marked with 'I' executes their function directely after IR_UPDATE (see [1]).

JTAG command	JTAG address	Width	Access	Description
READID	0x00	32 bit	R	Returns JTAG device id = 0x14849434
LVDS_PADS_EN	0x02	1 bit	W	Switch on/off LVDS pads (default:ON)
LINK_CTRL	0x03	9 bit	W	Configuration of link protocol (table 2.10.27)

JTAG command	JTAG address	Width	Access	Description
LAYER1_MODE	0x04	8 bit	W	Configuration of the direction of the Layer1 buses
SYSTEM_ENABLE	0x05	1 bit	W	Switch on/off digital communication block (default:ON)
BIAS_CTRL	0x06	6 bit	W	Control reference bias block (table 2.10.28)
SET_IBIAS	0x07	15 bit	RW	Control LVDS + SERDES bias currents
START_LINK	0x08	0 bit	I	Trigger start of communication link
STOP_LINK	0x09	0 bit	I	Trigger stop of communication link
STOP_TIME_COUNT	0x0a	1 bit	W	Reset global system time conter
READ_SYSTIME	0x0b	15 bit	R	Read current system time counter
SET_2XPLS	0x0d	1 bit	W	Configure manual transmission of double pulse event packets
PLL2G_CTRL	0x10	3 bit	W	Configure GHz PLL (table 2.10.29)
SET_TX_DATA	0x11	64 bit	W	Set packet content for transmission
GET_RX_DATA	0x12	64 bit	R	Read received packet content
SET_TEST_CTRL	0x17	4 bit	W	Configure transmission test modes (table 2.10.30)
START_CFG_PKG	0x18	0 bit	I	Trigger configuration packet start
START_PULSE_PKG	0x19	0 bit	I	Trigger pulse event packet start
READ_STATUS	0x1a	8 bit	R	Read current communication link state (table 2.10.31)
SET_RESET	0x1b	0 bit	I	Trigger setting of communication link reset
REL_RESET	0x1c	0 bit	I	Trigger release of communication link reset
SAMPLE_PRELOAD	0x1e			not implemented
INTEST	0x1f			not implemented
EXTEST	0x20			not implemented
SET_DELAY_RX _DATA	0x21	6 bit	RW	Sets/Reads delay setting in LVDS data pad

JTAG command	JTAG address	Width	Access	Description
SET_DELAY_RX _CLK	0x22	6 bit	RW	Sets/Reads delay setting in LVDS clock pad
READ_CRC_COUNT	0x27	8 bit	R	Read current CRC error counter of communication link
RESET_CRC_COUNT	0x28	0 bit	I	Reset current CRC error counter of communication link
PLL_FAR_CTRL	0x29	15 bit	W	Control Faraday PLL, see table 2.10.32
ARQ_CTRL	0x30	32 bit	W	ARQ control bits, see table 2.10.33
ARQ_TXPCKNUM	0x31	32 bit	R	Total packets transmitted
ARQ_RXPCKNUM	0x32	32 bit	R	Total packets received
ARQ_RXDROPNUM	0x33	32 bit	R	
ARQ_TXTOVAL	0x34	32 bit	W	Clock cycles without ACK until re-transmit of last packet
ARQ_RXTOVAL	0x35	32 bit	W	Clock cycles without link activity until ACK of last packet
ARQ_TXTONUM	0x36	32 bit	R	Number of timeouts in ARQ master
ARQ_RXTONUM	0x37	32 bit	R	Number of timeouts in ARQ target
BYPASS	0x3f	any	RW	JTAG standard BYPASS functionality

Table 2.10.26: Detailed list of all JTAG registers in an HICANN

Bit	Reset value	Description
[0]	1'b1	initialization automatic mode
[1]	1'b0	initialization master mode
[2]	1'b0	enable protocol handling for configuration
[3]	1'b0	enable protocol handling for pulse events
[4]	1'b0	8b/10b coding
[5]	1'b0	use 500MHz mode
[6]	1'b1	evaluate timestamps in pulse events
[7]	1'b0	debug loopback mode
[8]	1'b1	initialization automatic link speed detect

Table 2.10.27: Detailed bit of LINK_CTRL JTAG register

Bit	Reset value	Description
[4:0]	5'h10	Digital calibration value
[5]	1'b0	Debug: Enables 100uA reference output

Table 2.10.28: Detailed bit of BIAS_CTRL JTAG register

Bit	Reset value	Description
[0]	1'b1	Enable VCO
[1]	1'b1	Enable 500MHz clock
[2]	1'b1	Enable 1GHz clock

Table 2.10.29: Detailed bit of PLL2G_CTRL JTAG register

Bit	Reset value	Description
[0]	1'b0	Access to LVDS transmission towards L2
[1]	1'b0	Access from JTAG to pulse events for L1
[2]	1'b0	Access from JTAG to configuration packets for L1
[3]	1'b0	Enables packet type filter in combination with [2:1]

Table 2.10.30: Detailed bit of SET_TEST_CTRL JTAG register

Bit	Description
[0]	link ready for pulse event transmission
[1]	unitizialized
[2]	received CRC error
[3]	500 MHz mode
[4]	receiving configuration packet
[5]	receiving pulse event packet
[6]	link ready for configuration packet transmission
[7]	received pulse event packet contained two events

Table 2.10.31: Detailed bit of READ_STATUS JTAG register

Bits	Function
[0]	put PLL into test mode
[1]	select frequency range (1 high, 0 low)
[2]	low-active power down
[8:3]	*ns*: multiplier for frequency setting
[14:9]	*ms*: divider for frequency setting

Table 2.10.32: Control bits for Faraday PLL. 15 valid bits need to be shifted for this register. Frequency is input frequency $\times ms/ns$.

Bit Pos.	Function
0	high-active reset for Tag0 ARQ instance
1	loopback enable for Tag0 ARQ instance
16	high-active reset for Tag1 ARQ instance
17	loopback enable for Tag1 ARQ instance

Table 2.10.33: JTAG ARQ control bits. 32 bit need to be shifted for this register.

2.10.5.2 FPGA JTAG Access

JTAG command	JTAG address	Width	Access	Description
IDCODE	0x0	32 bit	R	Returns JTAG device id = 0x1C56C007
GET_HICANNIF_ STATE	0x1	30 bit	R	Read systime, crc counter, communication channel status (table 2.10.37)
GET_HICANNIF_ RXCFG	0x2	64 bit	R	Read last received configuration packet
GET_HICANNIF_ RXPLS	0x3	24 bit	R	Read last received pulse event packet
SET_HICANNIF_ CONFG	0x4	29 bit	W	Configuration for HICANN link (table 2.10.35)
SET_HICANNIF_ CTRLS	0x5	5 bit	W	Control of communication links mechanisms (table 2.10.36)
SET_HICANNIF_ TXDAT	0x6	64 bit	W	Data for transmission
HICANN_PACKET_ GEN	0x7	26 bit	RW	Pattern generator control and status
SET_HICANN_ CHANNEL	0x8	3bit	W	Selects active HICANN channel
BYPASS	0xf	any	RW	JTAG standard BYPASS functionality

Table 2.10.34: Detailed list of all JTAG registers in the FPGA

Bit	Reset value	Description
[0]	1'b0	start link
[1]	1'b0	debug loopback enable
[2]	1'b1	initialization auto mode
[3]	1'b1	initialization master mode
[4]	1'b0	evaluate timestamps in pulse events
[5]	1'b0	automatic releasetime for pulse events

Bit	Reset value	Description
[6]	1'b0	enable protocol handling for pulse events
[7]	1'b0	enable protocol handling for configuration
[15:8]	8'h0	L1 channel directions (heap memory mode)
[26:16]	11'h004	manual pre start time of pulse event
[27]	1'b0	8b/10b coding

Table 2.10.35: Detailed bit of HICANNIF_CONFG JTAG register

Bit	Reset value	Description
[0]	1'b0	start pulse event packet
[1]	1'b0	start configuration packet
[2]	1'b0	write routing memory data
[3]	1'b0	init channel reset mechanism
[4]	1'b0	reset CRC counter

Table 2.10.36: Detailed bit of HICANNIF_CTRLS JTAG register

Bit	Description
[7:0]	initialization automatic mode
[15:8]	initialization master mode
[29:16]	current systime counter

Table 2.10.37: Detailed bit of HICANNIF_STATE JTAG register

2.10.6 Experiment control

An experiment run is controlled from the FPGA via the different payload types defined in section 2.10.2. Especially, experiment execution is controlled via the FPGA configuration packet. A typical experiment run is performed by the following sequence:

1) Reset playback and trace memories (packet: FPGA Config)
2) Configure HICANN via ARQ (packet: HICANN Config)
3) Write playback pulses (packet: FPGA Playback)
4) Write HICANN configuration packets to playback memory, to be transmitted while experiment is running (packet: FPGA Playback)
5) Start playback and trace memories (packet: FPGA Config)
6) Stop trace memory (packet: FPGA Config)
7) Start readout of trace memory (packet: FPGA Config)
8) Receive traced pulses (packet: FPGA trace memory)

2.11 Hardware Abstraction Layer

The deep hiearchy of configurable hardware components (cf. sections 2.10.3 and 2.10.4) and the amount of controllable parameters already suggest that hardware configuration is complicated. This makes a clear, simple and robust abstraction layer one key component in the NM-PM software stack. Main elements of the Hardware Abstraction Layer (HAL) are: a robust coordinate system (cf. section 2.11.1), clear and consistent data structures representing configurable hardware entities, methods to access them (cf. section 2.11.2), and a synthesis layer which combines all these components into a single configuration metholodogy (cf. section 2.11.3).

2.11.1 User Coordinate System

This section has been written by Sebastian Jeltsch.

Conventional computers access memory and IO devices by mapping them into a single linear address space. Neuromorphic hardware devices on the other hand, implement small special purpose circuits with local memory. Using the immediate address space of the digital logic is complicated by the fact, that event the mapping of memory location to functionality varies across instances of the same analog circuit. Without random access memory, components have to communicate either via digital buses or physical coupling. The necessary connectivity is often defined by their relative orientation and position to one another. Thus, Unlike conventional computers, programming a neuromorphic hardware devices requires explicit knowledge about the topology. In fact, getting the connectivity right is one of the major obstacles when setting up experiments manually on a low-level. HALbe consistently arranges components in a hybrid-Cartesian coordinate system that reflects the component inter-connections more naturally. The term coordinate system is used rather than address space, because the Exemplarily, some coordinates are outlined in Figure 2.11.1.

Component indices are counted up from left to right and top to bottom, according to the HICANN orientation shown in Figure 2.11.1. Grid coordinates also support linear addressing by enumerating all instances in a row-first fashion. Moreover, sparse grids are supported, because not every grid point is necessarily assigned to actual hardware e.g. synapse drivers exist only every other switch row. Addressing invalid instances via x, y or enumeration yields an C++ and Python exception, respectively.

Finally, coordinates implement convenient conversion functions wherever possible to derive coordinates of connected components, e.g. a select switch row is connected to a synapse driver.

2.11.1.1 Implementation

The implementation is based on a ranged integer template library by the author. Any integer instance thereof, carries its valid value range as part of the type signature. In order to detect any violation, instances are sanitized during construction and compound assignment operations (e.g. +=). Optionally, ranged integers can be reduced to their corresponding built-in type at compile time to diminish any overhead for production builds. Many other workflow components use them to implement rigorous, concise value sanitization. The following example illustrates its use:

```
typedef integral_range<unsigned, 64 /*max*/, 4/*min*/> type;
type a = 0;  // raises exception
type b = 4;
a =>> 1;     // raises exception (a=2)

void magic_fun(integral_range<uint8_t, 7, 0> const& v);
```

Note, that the function declaration, not definition, clearly states its expected parameter range. No extra documentation is necessary.

HALbe coordinates are implemented on top of the ranged integer library as individual types. C++ inheritance simplifies implementing new coordinates. Sparse grids are implemented efficiently via CRTP [18] callback functions, eliminating virtual function overhead and enabling code inlining. Here, an actual implementation of a sanitized neuron coordinate within a 2×2 is presented to demonstrate how simple it is.

```
struct NeuronOnQuad :
    public GridCoordinate<NeuronOnQuad, XRanged<1, 0>, YRanged<1, 0> >
{
    NeuronOnQuad() = default;
    NeuronOnQuad(x_type const& x, y_type const& y) :
        self_type(x, y) {}
};
```

`x_type`, `y_type` and `self_type` are defined by the base class. The base class also provides time common grid, enumeration, serialization and hash map interfaces. The new coordinate can be used as follows.

```
// ...
NeuronOnQuad a(X(1), Y(1));
NeuronOnQuad b(Enum(3)); // references same neuron via enum
NeuronOnQuad c = NeuronOnHICANN(Enum(0)).quad();
// ...
```

The first two examples reference the same neuron in Cartesian coordinates and enumeration respectively. The third example presents a coordinate conversion, from a neuron on the HICANN into a neuron within the 2*times*2 block.

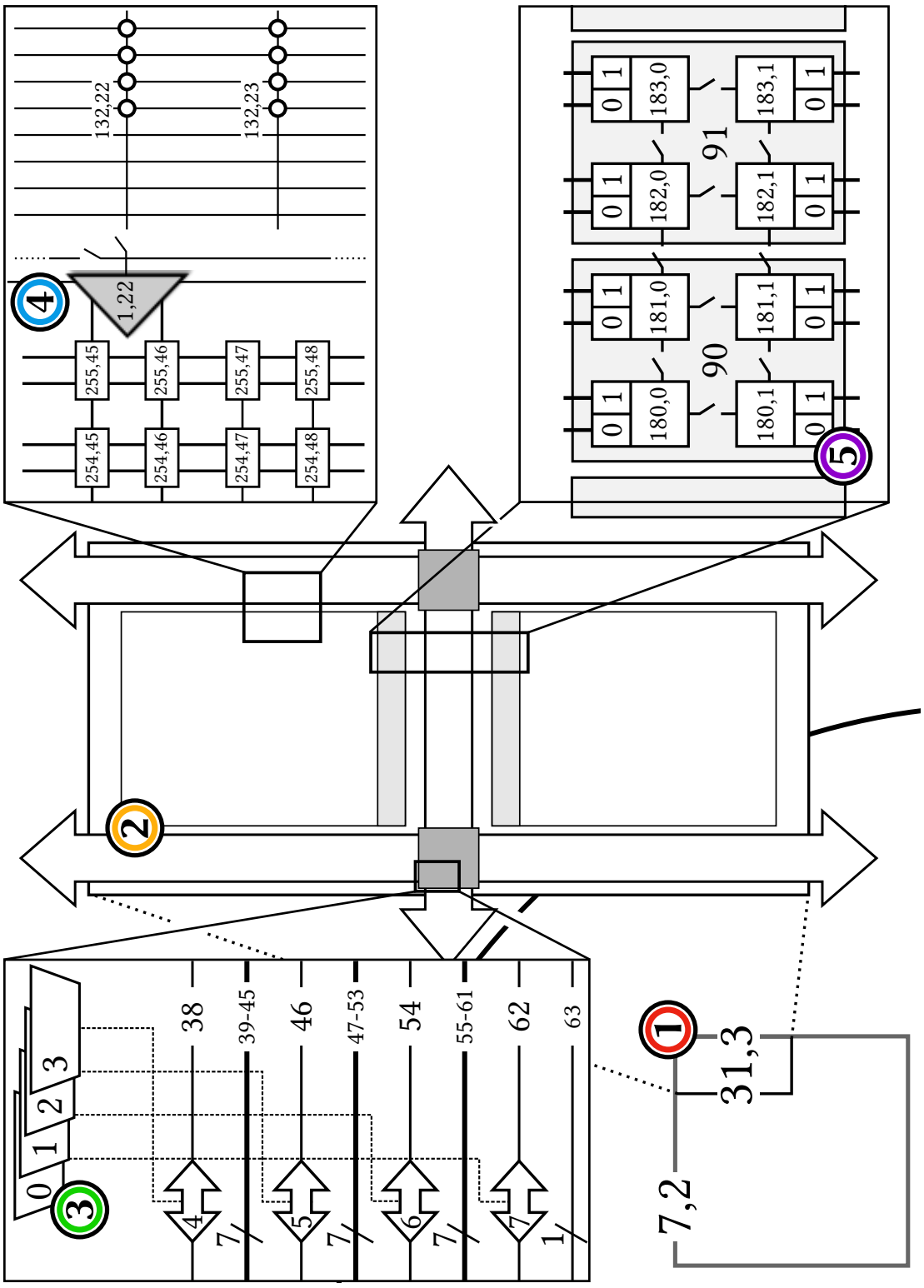


Figure 2.11.1: An illustration of HALbe coordinates for HICANN (30,3) **(2)** on reticle (7,2) **(1)**. Enumerations are given as plain indices and grids as combination of ***x, y***. Item **(3)** shows the mapping of DNC mergers to SpL1 repeaters. Note that merger *i* is merged to SpL1 repeater $7-i$. A small area including synapses, a driver and select switches is presented at item **(2)**. Lastly, the neuron inter-connection topology across and within blocks of 2×2 neurons is illustrated at item (5). Figure by Sebastian Jeltsch.

2.11.2 Stateless API

Author: Eric Müller

The common configuration interface for all the user configurable hardware components is called Hardware Abstraction Layer Backend (HALbe). This API uses the type-safe user coordinate system described in section 2.11.1 and additionally defines data containers for all user-configurable hardware entities (cf. sections 2.10.3 to 2.10.5). The interface is based on free, stateless –from the user's perspective– functions taking a handle that identifies the communication channel to the corresponding hardware unit, and coordinates to identify a unit within the hardware entity. Functions writing to the hardware additionally take a third argument that contains the data to write; functions reading from the hardware return a non-void data container.

In the following code listing examples for write access (i.e. a setter) and read access (getter) are shown:

```
// namespace HICANN
void set_crossbar_switch_row(
        Handle::HICANN& h,                    // communication channel
        Coordinate::HLineOnHICANN const& y, // coordinate 1st part
        Coordinate::Side const& s,            // coordinate 2nd part
        CrossbarRow const& switches           // data container
);

// namespace ADC
raw_data_type get_trace(
        Handle::ADC & h
);
```

The implementation of the API supports several backends: accessing the real NM-PM, the simulated ESS (cf. section 2.11.4) and multiple debugging modes (e.g., to visualize low-level configuration data, or to assess neuron behavior, cf. section 2.11.5). In case of the NM-PM backend, the main objective is the translation between user-friendly coordinates and data containers on the one hand and low-level hardware commands accessing hardware entities on the other hand.

The HALbe interface does not impose a specific configuration order. To tackle this problem, API functions are annotated with configuration states. The order of state transitions is then checked by a FSM –called State Checking and Error Identification Framework (Scheriff)– to identify illegal transitions (e.g., reading before resetting a hardware unit). For a detailed description of the reset handling see section 2.3.4.1.

2.11.2.1 Real-time Access

Executing closed-loop experiments on the NM-PM requires a close interaction between Wafer Module, FCP and Compute Node. To support this operation mode a thin software layer, called Virtual Environment for Closed-Loop Experiments (VerCL), is provided. This API provides methods to communicate spikes between FCP and Compute Node at low latency.

From the user's perspective, the software part of a experiment running in real-time requires additional precautions to eliminate unpredictable latency sources like page faults or call overhead. This makes it difficult to use convenience functions that are available for batch-style experiments; typically, the user code has to operate directly in the hardware value and time domain. During runtime, extended permissions are also needed to control real-time behavior of the operating system environment, and the custom network hardware (cf. section 2.8.1).

2.11.3 Low-level Stateful API

This section has been written by Christoph Koke.

StHAL is a C++ library that extends the functionality of HALbe. While HALbe is designed to be stateless and allows a fine granular access to every part of the NM-PM, StHAL provides access to the complete configuration and unified algorithms for common tasks. Also it handles low level runtime information as IP address, the assignment of HICANNs to FCPs, or the assignment of AnaRM (see chapter 2.7) to HICANNs. This makes it also a convenient tool to write low level test programs for the hardware and for the calibration tools (see section 2.13.1). All StHAL functions and data structures are located in the namespace `sthal`.

StHAL combines the data structures of HALbe to represent the configuration state of the NM-PM. This includes also runtime data, like spikes or analog traces. These data structure is simply a collection of the structures provided by HALbe. To simplify the usage, the interface of StHAL is centred around the HICANN. Configurations that affect actually the FCP are derived from the corresponding feature of the HICANN and so completely hidden. This ensures that components are configure consistently. Where this is not possible, e.g. for spikes, they are mapped to the corresponding HICANN. By using the HALbe coordinate system (see section 2.11.1), it gives the user intuitive access to the configuration data via the `operator[]`. The following example uses a HICANN on the first wafer and enables all background generators:

```
sthal::Wafer wafer(Coordinate::Wafer(0));
sthal::HICANN & hicann = wafer[HICANNOnWafer(Enum(123))];
for( auto bg : iter_all<BackgroundGeneratorOnHICANN>)
        hicann.layer1[bg].enable(true);
```

Using `operator[]` allows to expose references to internal data structures easily in the Python bindings. The StHAL containers also support complete serialization, which is important for large scale experiment preparation and distribution.

Since StHAL holds the complete configuration state of an experiment it can also provide more complex configuration routines. For example, the following method ensures, that only the selected neuron is connected to the AnaRM:

```
hicann.enable_aout(NeuronOnHICANN(X(0), Y(0)), AnalogOnHICANN(0));
```

StHAL manages the connection to the hardware and hides it almost completely from the user. It gets the required informations to run an experiment via an instance of the

HardwareDatabase class. It provides an interface to gain all required runtime informations, like IPs, to setup an experiment on the hardware based on HALbe coordinates. Connecting to the hardware is reduced to a single call:

```
wafer.connect(sthal::MagicHardwareDatabase());
```

The MagicHardwareDatabase is a placeholder for a real connectivity database that might come in future.

To write the complete configuration to the hardware the class HICANNConfigurator can be used. It provides a robust default configuration routine to be used by the Wafer class to write the current configuration state completely:

```
sthal::HICANNConfigurator cfg;
wafer.configure(cfg);
```

The HICANNConfigurator takes all given constrains of the HICANN configuration into account. If required, custom configuration methods can be easily achieved by overwriting the virtual methods of the HICANNConfigurator. The ExperimentRunner takes care of sending and receiving spike runtime data from the NM-PM and to start and stop the experiment. Its virtual methods can also be overloaded to allow custom experiment design. The same concept is used to run the actual experiment. The following code will run an experiment of the duration of 1ms:

```
sthal::ExperimentRunner runner{1e-3};
wafer.start(runner);
```

Futher StHAL simplifies the usage of the AnaRM software (see section 2.7.3.2). This is implemented in the class AnalogRecoder. It locks the boards, when it is constructed and frees it at destruction. It also automaticaly loads the calibration and returns traces in Volts. The class is not intended to be constructed manually, but by an HICANN instance. Traces can be recorded manually or by using the trigger of the readout board. An error will be thrown, when trace() is called, but no data has been recorded yet. The following example uses manually recording, in this case the record function will block until the recording has finished:

```
auto recorder = hicann.analogRecorder(AnalogOnHICANN(0));
recorder.record(1.0e-2)
auto trace = recorder.trace()
```

The trigger of the readout board will by activated by the FPGA, when the playback memory starts (see section 2.6.3.4). In following example neurons are stimulated and the trigger is used to record the membrane in parallel:

```
std::vector<sthal::Spike> spikes;
for (size_t ii = 0; ii < 400; ++ii)
{
        spikes.emplace_back(L1Address(0), ii* 3e-6);
}
```

```
hicann.sendSpikes(GbitLink(0), spikes);

// ... SNIP ... (L1 routing stuff)

auto recorder = hicann.analogRecorder(AnalogOnHICANN(0));
recorder.activateTrigger(1e-3);
sthal::ExperimentRunner runner{1e-3};
wafer.start(runner);
auto trace = recorder.trace()
```

StHAL has complete Python bindings. Its the API documentation can be found at table 2.10.19

2.11.4 Executable System Specification - Simulation Layer

This section has been written by Bernhard Vogginger.

The ESS is a software model of the NM-PM hardware part. It contains functional models of all relevant units of the neuromorphic circuits (chapter 2.3) and the communication modules (chapter 2.6). The ESS is fully executable and resembles how neural experiments will be run on the real system. It can be operated from HALbe or StHAL, so that any experiment for the hardware can be also executed with the ESS. Being a software model, the ESS is fully deterministic, e.g. compared to a produced wafer it does not suffer from transistor-level variations after manufacturing. It can therefore be used as a testbench for the software frontend to the system (chapter 2.13) or as a simulation backend for neural modelers, who want to explore the capabilities of the NM-PM platform or prepare their models for emulation on the real system.

2.11.4.1 Implementation

The ESS is a detailed simulation of the final NM-PM platform and has been implemented in C++/SystemC ([73, 23, 17]). It replicates its physical counterpart in all aspects regarding functionality and configuration space. Every module of the real hardware has its functional counterpart in the virtual device, where especially the interface and communication structures accurately correspond to the physical system. It implements all analog and mixed-signal modules such as Adaptive Exponential Integrate-and-Fire (AdEx) neurons and dynamic synapses (section 2.3.3.2, as well as all units responsible for on-wafer and off-wafer communication (cf. section 2.3.2, rsp. chapter 2.6). Compared to analog and Register Transfer Level (RTL) hardware simulations, this model is tuned towards simulation speed using behavioral models of all relevant functional components, e.g. the neuron circuits are just implemented by the differential equations of the AdEx model. However, it is possible to replace individual modules by more sophisticated models, all the way down to simulating single wires on the chip.

2.11.4.2 Comparison with real system

The current implementation of the ESS differs from the real hardware system in several aspects, which are listed below:

- STDP (section 2.3.4.5) is currently not implemented in the ESS
- No distance-dependent delays on the wafer: The transmission latency of pulses in the L1 routing network, which mainly depends on the number of L1 repeaters of the routing path, is not considered in the ESS for efficiency reasons. Instead, an average delay value is applied approximating this latency as well as the transmission delay in the ANNCORE.
- Access to all states: The ESS allows to trace states that are not accessible on the real system, e.g. all state variables of the neurons and synapses for many units, and not only the membrane potential of a selection of neurons.
- Logging of lost pulses: Pulses can get lost due to bandwidth limitations of the serial pulse transmission in the HICANN or the off-wafer communication modules (chapter 2.6). In the ESS these lost events are logged and counted.
- Ideal neuron and synapse models: The ESS directly implements the equations of the underlying neural models and not the equations of the physical implementation. Hence, hardware specifics like a limited dynamic range, leakage currents, crosstalk or thermal noise are not considered.
- No imperfections from manufacturing: Per default, every module on the ESS works as designed, e.g. there is no transistor-level mismatch leading to a fixed variation of neuron or synapse parameters. However, the ESS allows to artificially impose such distortions, e.g. one can specify a fixed-pattern noise for synaptic weights. Furthermore, one can load calibration data for neurons and synapse drivers.

Despite these differences the ESS remains a proper replica of the NM-PM hardware part providing equal functionality while not suffering from hardware-specific constraints like transistor-level imperfections from the manufacturing process. Offering the same configuration space and structure as its real counterpart, it is a perfect testbench for the software modules of the user interface to the system (chapter 2.13).

Furthermore, the ESS builds a useful tool for neural modelers who want to run their models on the NM-PM platform. By running ESS simulations, the modelers can analyze the effects of many hardware-specific constraints on their models, such as limited network connectivity, shared and discretized parameters, or activity dependent spike loss and spike time jitter, independently from hardware imperfections or missing calibration data.

2.11.4.3 Using the ESS

For the ESS there is an implementation of the HAL for both the stateless (HALbe) and the stateful API (StHAL). Here, we describe the use via StHAL.

The setup and execution of an ESS experiment follows the procedure for the real system (section 2.11.3). Only two modification have to be made: First, as the hardware database

use a ESSHardwareDatabase, which further requires a directory path specifying the location for temporal data of the ESS. Second, instead of the ExperimentRunner, use the ESSRunner.

Thus, a StHAL ESS experiment essentially looks as follows:

```
sthal::Wafer wafer(Coordinate::Wafer(0));
// ... SNIP ... (collect wafer configuration)
std::string ess_dir_path("./ess_dir")
wafer.connect(sthal::ESSHardwareDatabase(ess_dir_path));
// ... SNIP ... (configure ESS)
sthal::ESSRunner runner{1e-3};
wafer.start(runner);
```

2.11.5 Hardware Simulations

Author: Eric Müller

To assess hardware-specific neuron or synapse behavior detailed analog transistor-level simulations, are essential. Due to software licensing issues analog simulations are typically only available for chip developers. Furthermore, the simulation interfaces supplied by the analog circuit simulators are very generic and not optimized for neuronal network modelers.

Hence, an user-friendly interface to such an analog simulation is important. The HALbe backend for simulation of analog circuits (SimDenMem) bridges exactly this gap. Synapse driver, synapses and neuron membrane circuit (cf. sections 2.3.3.1 to 2.3.3.3) are evaluated by an analog simulator.

SimDenMem is a HALbe (see section 2.11.2) API implementation targeting an Inter-process Communication (IPC)-based simulation backend. Coordinates and data containers are appropriately converted. For example, boolean values enabling or disabling transistors have to be converted into analog voltage levels (e.g., VDD or 0 V) and a relevant subset (e.g., the neuron circuit parameters) of all analog parameters has to be extracted. A client-server-based software using IPC transfers the simulation job onto a simulation server. The simulation server uses a proprietary analog circuit simulator to obtain results and returns the data to the IPC client. In the last step result data is returned to the user and can now be visualized.

As long as user uses only one DenMem the experiments can be executed on both, the NM-PM system or the HALbe backend for simulation of analog circuits (SimDenMem) backend.

2.12 System Management Layer

Author: Eric Müller

The operation of the NM-PM system involves many resource management tasks. On front-end side, the user issues experiment jobs, on the back-end side many hardware entities have to be orchestrated. In particular, this includes assigning fractions of the NM-PM system to jobs, post-job clean-up, keeping track of hardware usage, failures, user authentication/ authorization and maintaining fairness.

2.12.1 Cluster

All network components (FCPs, Power Management Units (PMUs), Compute Nodes) are located in a single Ethernet broadcast domain. Multiple IP networks split Wafer Module/host pairs into logical communication domains. The compute nodes boot into a network-based, stripped-down Debian-based operating system – different kernel configuration options are available to focus on different operation modes (e.g., closed-loop experiments or data analysis, cf. chapter 1.2). Local I/O operations are executed within an memory-backed overlay filesystem. This allows for sharing a common file-system root between all compute nodes. Cluster resources are managed by Simple Linux Utility for Resource Management (SLURM) [45], monitoring is handled by Ganglia [46] and Nagios [12] or equivalent tools. Fairness between users is ensured using the built-in SLURM priority management capabilities. Multiple I/O nodes provide cluster storage for input and output data. This data is available on all frontend and compute nodes.

2.12.2 Hardware Resources

As a first step the hardware resources are mapped one-to-one to cluster nodes. This not only involves Ethernet-based communication channels but also analog-readout assignments (cf. chapter 2.7). The mapping between Wafer Module power control (e.g., enabling parts of the wafer, monitoring power consumption, temperature and other operating paramaters) and Compute Node is also statically mapped.

2.12.3 Users

Local users (i.e. users having access to the cluster login nodes) are allowed to submit jobs for execution on the NM-PM. Remote access is secured using X.509-based [11] certificates. For the high-level description see section 1.4.2.

In the remote case a client-server-based software infrastructure handles user authentication/authorization and submits jobs into the same job queue as local users. The user interface is based on the PyNN API (cf. sections 1.3.2 and 2.13.3).

2.13 PyNN Frontend and Translation Libraries

This section describes software methods that are necessary to provide the user with a controllable, widly homogeneous system. Calibration of the neuromorphic computing substrate is covered in section 2.13.1. Section 2.13.2 deals with the mapping between neuronal network descriptions and valid hardware configurations.

2.13.1 Calibration

Due to the analog nature of the neuron circuit, unavoidable variations in the fabrication lead to a slightly different behaviour for each DenMem. The aim of the calibration is to reduce these neuron-to-neuron variations by extracting relationships between the input parameter space and individual neuron behaviour. This section will give an overview over the calibration of the NM-PM. It will also explain the methods used to deduce neuron parameters from measured voltage traces. Finally, as an example, the calibration of the neuron resting potential will be described in more detail.

The parameterizability of the analog components of a neuron is realized by reading analog values from programmable floating gate memory cells, see section 2.3.3.5. Those floating gate memory cells provide either voltages or currents, with voltage ranges from 0 V to 1.8 V and current ranges from 0 µA to 2.5 µA. Both floating gate types are programmed via discrete 10 bit DAC values. The translation from desired voltages and currents to DAC values is one of the tasks that is done by the calibration.

Since we are not able to directly measure parameters like time constants or conductances, it is required to deduce these parameters from measured membrane voltage traces.

The main task of the calibration is therefore to provide a lookup from analog hardware values to the programmable discrete floating gate parameters. In this step, the intrinsic neuron-to-neuron variations that stem from the hardware features outlined above are mitigated. The calibration software can also be used to report if a neuron shows behavior that makes it unsuitable to be used in networks, i.e. it is marked as defective. Both the calibration and the defects are stored in a database.

The speedup of hardware time compared to biological time is taken into account by these transformations. Required corrections due to transistor size mismatches and effects described below are applied as well.

There are several factors in the HICANNv2 neuron circuit which lead to deviations that the

calibration has to correct for. The first factor affects the actual measurement of membrane voltages, not the voltages themselves. Each neuron is connected to a readout ADC line via an operational amplifier which has a natural offset varying from neuron to neuron. In simulations, this offset was calculated to be (0.957 ± 11.579) mV (see fig. 2.13.1). Hence, when measuring the same voltage for different neurons, we expect a natural standard deviation of at least 11.579 mV over all neurons. To correct for this readout shift, we make use of the fact that one V_{reset}, as a shared parameter, is applied to 128 neurons and therefor should not have any neuron-to-neuron variation among one block. By measuring V_{reset} for all 128 neurons connected to one floating gate block and taking into account that they are all connected to the same voltage, we can extract the readout shift for each individual neuron. This is done by taking the mean $\overline{V}_{\text{reset}}$ over one block as a reference and then measuring the deviation $V_{\text{shift,i}}$ of each neuron i from that mean value. Each successive voltage measurement is then shifted back by $-V_{\text{shift,i}}$.

After a membrane voltage trace has been recorded, the desired parameter needs to be extracted. To measure LIF-parameters, the following methods are used:

Membrane leakage potential $\mathbf{E_l}$: V_t is set much larger than E_l so that the neuron does not spike. Without any input, the membrane voltage trace is measured. The mean $\overline{V}_{\text{mem}}$ of this voltage trace is taken to be the neuron resting potential.

Reset voltage $\mathbf{V_{reset}}$: The membrane resting potential is set above the spike threshold so that the neuron always spikes. τ_{ref} is set to a high value in order to keep the voltage at V_{reset} for a longer time period. The, the voltage base line between spikes is measured, which is given by V_{reset}.

Spike threshold voltage $\mathbf{V_t}$: As in the V_{reset} measurement, E_l is set above V_t so that the neuron always spikes. Then, the maximum of the voltage trace is measured, which is given by V_t.

Membrane leakage conductance $\mathbf{g_l}$: V_t is set much larger than E_l to avoid spiking. A small, periodic step current is injected into the neuron which leads to periodic rise and decay of the membrane voltage. Then, an exponential function is fitted to the voltage decay to get τ_m. The conductance is given by $g_l = \frac{C}{\tau_m}$ with C being the capacitance of the neuron circuit.

Synaptic reversal potentials $\mathbf{E_{synx}}$ and $\mathbf{E_{syni}}$: All conductances except the synaptic conductance are set to zero. Strong spikes with a high frequency are sent to the neuron. These spikes cause the membrane potential to keep at the reversal potential E_{syn}. E_{syn} is then extracted by taking the mean of the membrane voltage trace.

Synaptic time constant τ_{syni} and τ_{synx}: V_t is set much larger than E_l to avoid spiking. Regulare spikes from the background generator are used to stimulate the neurons. The spikes distance is large enough, that the PSPs will not overlap. Multiple PSPs are recorded and a double exponential is fit to the mean of the recorded PSP. The time constants can be obtained from the fit.

In the following, the calibration of the membrane leakage potential is exemplified. The method described above is used for different values, called steps, of the hardware parameter E_l. Figure 2.13.2 shows an example of five different voltage traces for one specific E_l. The measurement is repeated several times per step and averaged over all repetitions to compensate floating-gate variations. Results for three different steps are shown in fig. 2.13.3. The calibration is summarized in fig. 2.13.4. This figure shows that the mean values after

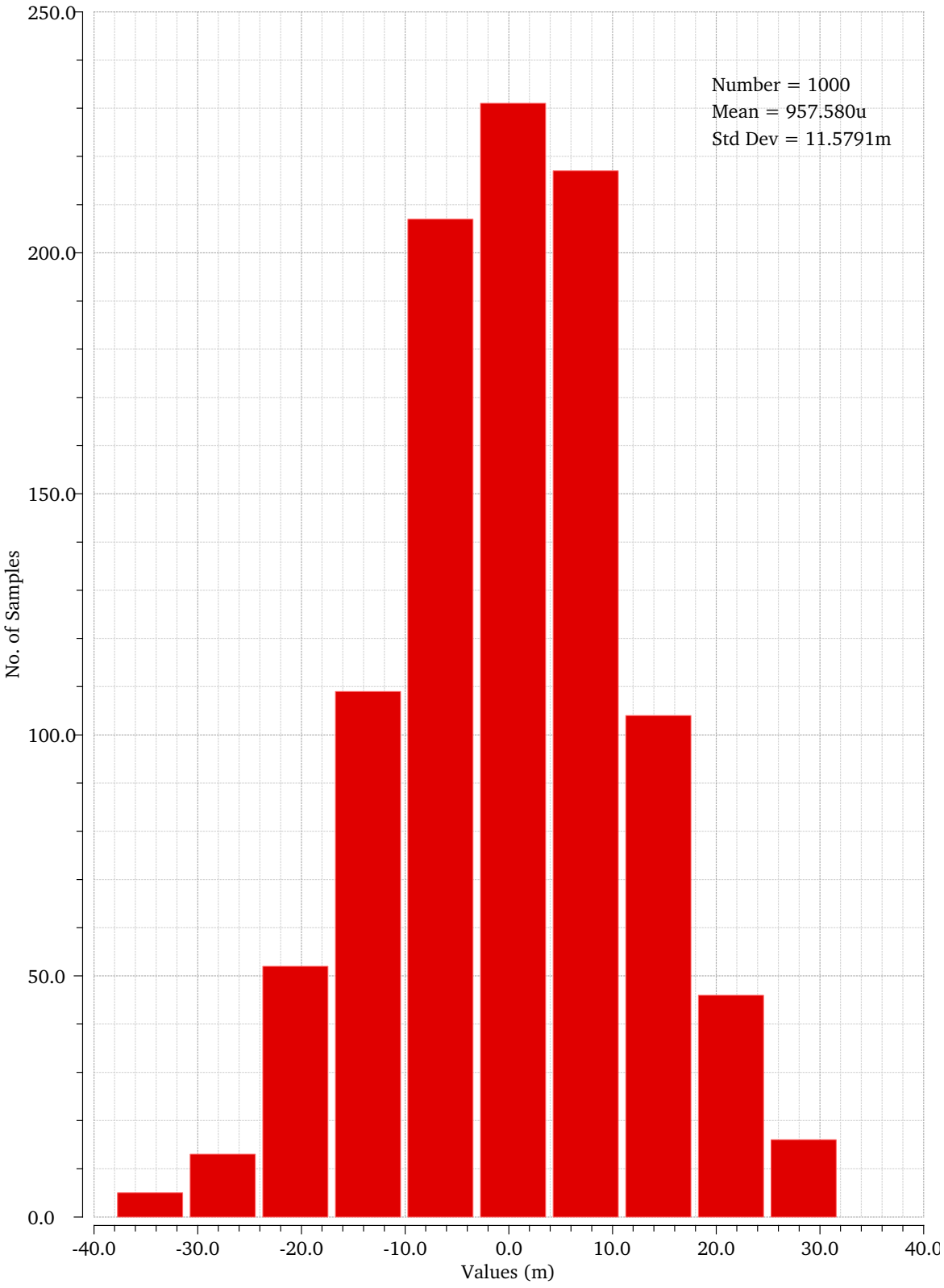


Figure 2.13.1: Result of a Monte-Carlo-Simulation of the neuron output amplifier.

calibration are closer to the target values and the neuron-to-neuron variation is reduced.

2.13.2 Automated Mapping of Neural Networks to Hardware

The original mapping has been redesigned and is now known as marocco.

This section has been written by Sebastian Jeltsch.

2.13.2.1 Neuron Placement

Input: A list of available HICANN chips; Optionally, a partial or complete user defined neuron placement.
Output: An assignment of each model neuron to a set of hardware neuron modules.
Defects: Blacklist of neurons.

The redesigned mapping process provides a simple but fast neuron placement implementation. In PyNN, network connectivity is typically established between populations. Thus, population tend to have common sources and targets. Heuristically placing neurons within the same population onto the same chip saves routing resources, because connections can be realized as part of the same L1 route. This works particularly well for feed-forward networks. However, PopulationViews and Assemblies simplify the realization of other connection patterns, rendering the heuristic less effective.

Additionally, users can place model neurons onto variably interconnected hardware neurons, allowing configurations with higher input counts. This is a major advantage over the former mapping, where all model neurons had to have the same neuron interconnection topology on hardware. Currently, the implementations allows only block shaped neuron configurations with neuron numbers that are multiples of two to simplify local routing. Consequently, events can be relayed to the neuron from either synapse arrays. In practice, this is only a minor limitation, because input counts of medium sized networks easily exceed single neuron synapse resources. In fact, a reasonable default size was found to be eight interconnected neurons.

Guidance

The implementation allows users to guide the neuron placement by explicitly specifying target chips for populations. The modeler can also control the shape of neurons individually for each population. Hence, the number of afferent synapses can be fine-tuned, such that neurons in populations receiving lots of input are realized as larger hardware neurons. The following code listing demonstrate how the neuron placement is interfaced from Python.

```
marocco = Pymarocco()
marocco.placement.setDefaultNeuronSize(4)
setup(marocco=marocco)

# ...
p = Population(113, IF_cond_exp)
```

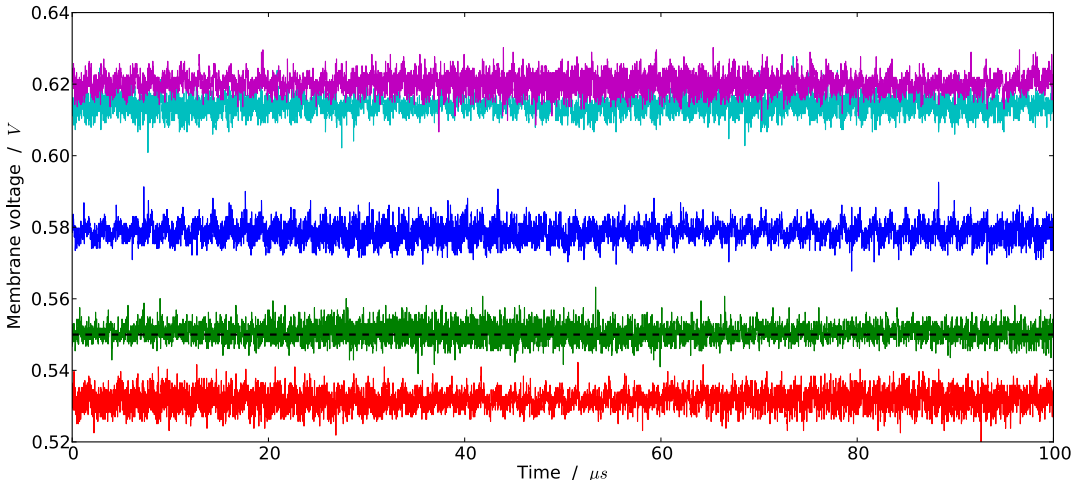


Figure 2.13.2: Membrane voltage traces for five different uncalibrated neurons. The dashed line shows the value of E_l that was set for all neurons.

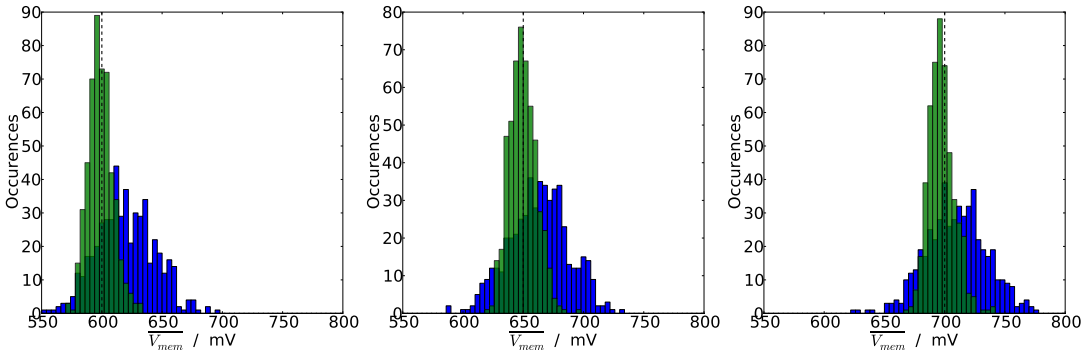


Figure 2.13.3: Distribution of measured $\overline{V}_{mem}$ before (blue) and after (green) calibration for three different values of E_l (dashed lines).

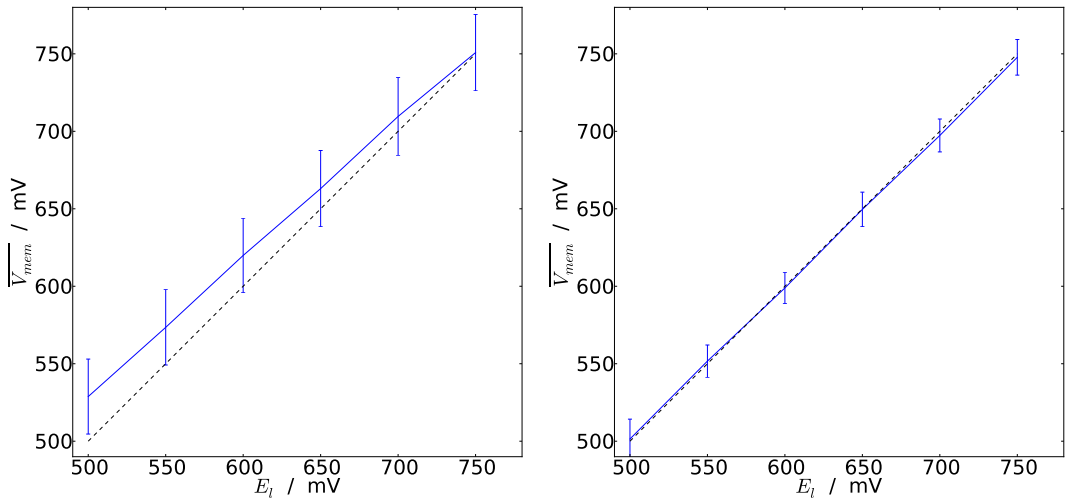


Figure 2.13.4: Summary of all neurons before (left) and after (right) calibration. The lines show mean and standard deviation over all neurons on one HICANN.

```
targets = [HICANNGlobal(X(14), Y(20)), HICANNGlobal(X(15), Y(20))]
marocco.placement.add(p, targets, 8)
```

Here, a population of 113 neurons is assigned to the chips with Cartesian coordinates $(X, Y) = (14, 20)$ and $(15, 20)$ (see Section 2.11.1). Each model neuron is placed to eight hardware neurons despite the default hardware neuron size of four.

Algorithm

The algorithm distinguishes between hardware neurons that are available for placement and those that are not. Initially, all blacklisted neurons are simply marked as not available. Then, manually placed populations are placed to their respective hardware locations. All occupied hardware neurons are also marked as no longer available for later assignments. Whenever the manual placement runs out of resources because either not enough chips have been specified for a population or too many neurons have been blacklisted, the assignment fails and the user is informed.

In the next step, all remaining populations have to be placed. Firstly, populations are sorted in descending order based on their input degree in the population graph. Secondly, all available chips are sorted in ascending order base on their location (X, Y). For any two chips at (X_0, Y_0) and (X_1, Y_1), chip 0 appears prior to chip 1 if it is closer to the center of the wafer. If equally far apart, chip 0 appears still before chip 1 if α_0 is smaller than α_1 for $\alpha_i = \text{atan2}(X_i, Y_i)$. The resulting ordering lists available chips spiraling out from the center. Starting with chips in the center helps to avoid boundary effects like reduced L1 resource densities. Furthermore, the resulting placement has a convex shape, which can be beneficial for some wafer routing implementations, see Section 2.13.2.4.

Now, populations and chips are iteratively popped from the beginning of both lists. For each pair, as many as possible model neurons are assigned to hardware neurons. Again, these neurons are marked as no longer available. Unassigned neurons are reinserted at the beginning of the population list in order to be assigned in the next iteration. How many hardware neurons are required depends on the number model neurons, the defect distribution and the neuron interconnection size. The latter is given by a configurable default size. If a chip provides more neurons than necessary, it is reinserted at the beginning of the chip list and used for the next population. Even though larger populations are fragmented across multiple chips they are guaranteed to end up nearby. This reduces the necessary routing resources for connections targeting this population.

Moreover, the implementation can optionally limit the number of available neurons per chip. This is particular useful for hardware neuron sizes of 4 and 8, where otherwise inefficient L1 usage would be the consequence as explained in Section 2.13.2.2. Less SpL1 repeaters are required in the subsequent merger routing stage to route all neurons off chip if the number of placed model neurons is reduced to 118 and 59 respectively. Experiments have shown that forgoing some hardware neurons is generally beneficial for the overall loss of synapses.

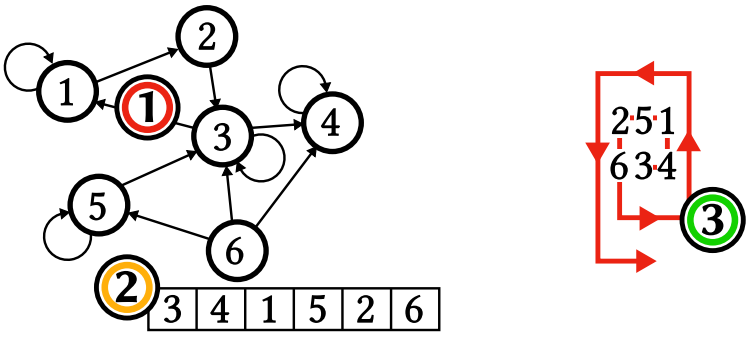


Figure 2.13.5: The PyHMF population graph **(1)** is sorted by in-degree in descending order **(2)**. Populations from the list are placed iteratively to hardware neurons on chips spiraling outwards from the center of the wafer **(3)**. This simple heuristic keeps neurons within a population topologically close together. Moreover, populations with lots of input are likely to end up closer to the center, which reduces the routing resource requirements for uniformly distributed sources.

Runtime

Initially, two lists are sorted. Sorting is done in $O(n) = n \lg n$, where n denotes the number of elements in the list. Then, model neurons are iteratively assigned to hardware neurons. In the worst case a single neuron is assigned per iteration. This contributes a worst case limit of $O(n) = n \cdot N_{\max}$, with the number of populations n and the size $N_{\max}$ of the largest population. All contributions sum up to

$$O(n, m) = m \lg m + n \lg n + n \cdot N_{\max} \quad , \tag{2.13.1}$$

where n is the number of populations and m number of chips. Which linearithmic term dominates depends on the input size. The implementation is efficient enough to place large numbers of model neurons onto extensive hardware setups in all cases.

2.13.2.2 Merger Routing

Input: Neuron placement.
Output: A merger tree configuration and L1 address assignment.
Defects: Blacklist of mergers and SpL1 repeaters.

After the model neurons have been placed, the first step towards connecting neurons, is to map them onto SpL1 repeaters. The outputs of up to 59 neurons from different blocks can be merged in order to save L1 resources. An illustration of the merger tree is found in Figure 2.3.15. The configuration requires to set each merger to forward either its left, its right, both or none of its inputs.

Only 59 of the theoretically 64 addresses are available. Address 0 is reserved for link synchronization and four more addresses are required to realize absent synapses. The implementation reserves all addresses of format $XX0001_2$ for this purpose.

Unlike its name, the merger tree is actually not a tree. Its an overlay of 8 trees with the SpL1 repeaters as root nodes. We can model the complete structure more easily as a directed acyclic graph. Edges connect mergers from lower levels to mergers in higher levels. Modeling the structure as a graph enables us to keep the algorithm generic and to cope with defect mergers efficiently, rather than implementing the fixed topology as part of the algorithm.

Looking at the merger tree, one of the first things to notice is, that mergers are not equally capable of merging events from different blocks. For example, the bottom most merger connected to DNC merger 3 is the only merger capable of collecting events from all neurons. The algorithm considers mergers in a fixed order from more to less capable in order to find configurations which require the smallest possible number of SpL1 repeaters.

In a first trial, merger 3 is considered. A breadth-first search [68] is used to discover reachable neuron blocks. If less than 59 neurons are placed to the chip and no defects are present this already yields a complete, efficient and valid configuration and the algorithm terminates. Otherwise, the trial is discarded.

In a second trial DNC mergers are considered for merging in the following order 5, 3, 1, 6, 4, 2, 7, 0. Unlike the first trial, assignments are established iteratively and configurations are kept even if a merger cannot merge all accessible blocks, because the number of neurons exceeds 59. For each DNC merger i, the accessibility of yet unassigned blocks is determined by means of a breadth-first graph search. Starting from the top level merger i, as many accessible neighboring blocks to the left and right are merged as possible. Neurons within these blocks are therefore assigned to the corresponding SpL1 repeater and require no further treatment. DNC mergers which cannot reach any mapped and yet unassigned neuron block remain available for the assignment of external spike sources in the next pipeline stage. The algorithm terminates if either no pending blocks are left or all DNC mergers have been processed.

The implementation results in efficient configuration in terms of L1 resource utilization. Any neuron placement is guaranteed to be be routed as long as no mergers are blacklisted. Otherwise, some of the neurons might be unroutable. The initial trial with merger 3 is important in order to find optimal configurations when all neurons can be mapped onto a single SpL1 repeater. However, starting the second trial with merger 3 may yield an unresolvable edge case, when the neurons from blocks 3 and 4 can be merged but not 5.

Finally, 6 bit L1 addresses are mapped to neurons. The assignment is in principle arbitrary, however, using consecutive addresses to minimize the number of different MSB improves the synapse driver assignment later on. Moreover, addresses are assigned from the highest to the lowest address. The lower address range $(0, 16]$ contains only 14 compared to 15 programmable addresses otherwise. Therefore, less synapse drivers are required on the receiver side for configurations realizing 15, 31, or 47 sources per L1 connection.

Runtime

The worst case occurs when all mergers have to be considered in the second trial, which requires constant time. Thus, the local merger routing for N chips can be established in in linear time $O(N) = N$.

Furthermore, merger tree configurations for multiple chips are derived efficiently in parallel. No means of synchronization are required, as local resource are assigned only.

2.13.2.3 Input Placement

Input: Neuron placement and List of remaining SpL1 repeaters.
Output: Mapping of spike inputs to SpL1 repeaters and L1 address assignment.
Defects: Blacklist for SpL1 repeaters.

The input placement is responsible for mapping external spike sources onto SpL1 repeaters to inject them into the L1 network. The actual spike trains are prepared as part of the parameter transformation (see Section 2.13.2.7), because their timing depends on the speedup of the system, which is subject to analog calibration. Note that the injecting FPGAs and DNCs are implicitly defined by the NM-PM1 setup.

The algorithm minimizes the required L1 resources by injecting inputs close to the geometric center of their target chips. Key to a fast implementation is to find suitable insertion points efficiently. Therefore, available SpL1 repeaters are organized in a KD-tree data structure. The `nanoflann` C++ library provides the necessary nearest neighbor algorithms [14].

Initially, spike input populations with local targets are collected in a list and sorted in descending order by their number of target chips. Afterwards, the ideal insertion point is determined for each entry as the geometric center of the target chips. All SpL1 repeaters which are not blacklisted and have not yet been designated for relaying events from neurons are available for external inputs. A KD-tree data structure is used to organize these repeaters and provide efficient geometric queries.

Iteratively, input populations are popped from the front of the source list. Each time, the SpL1 repeater closest to the ideal insertion point is queried in the KD-tree. Again, up to 59 sources can be mapped to a single SpL1 repeater. If the number of source exceeds the remaining capacity of the SpL1 repeater, as many as possible sources are assigned and the remaining ones are reinserted at the front of the source list. In case the repeater has been fully assigned, it is removed from the tree. The algorithm terminates, when either all inputs have been assigned or the system runs out of available SpL1 repeaters. In the latter case, the user is informed.

Runtime

The worst case runtime for M inputs and N chips is given by

$$O(N, M) = N^2 + M \cdot N \quad . \tag{2.13.2}$$

The first N^2 term is contributed by the initial KD-tree construction, where the second $M \cdot N$ denotes the repeated queries and deletions from the tree for all M inputs. On average, the KD-tree access complexity is reduced to $N \lg N$, resulting in

$$O(N, M) = (N + M) \cdot \lg N \quad . \tag{2.13.3}$$

Thus, the implementation perform reasonably for large networks and provides linearithmic scaling in the average case.

2.13.2.4 Wafer Routing

Input: Neuron Placement; Assignment of neurons and spike sources to SpL1 repeaters.
Output: A complete configuration of crossbar switches and repeaters; A mapping of pre-synaptic neurons to vertical buses on chips containing post-synaptic neurons.
Defects: Blacklists for buses, repeaters and crossbar switches.

The wafer routing configures repeaters and crossbar switches to establish long-range connectivity via the L1 network. Here, two implementations are presented, a simple one using Dijkstra's algorithm to discover shortest paths iteratively and an optimized L1 graph search that minimizes horizontal bus allocation. Both algorithms efficiently realize detours to cope with non-rectangular routing grids resulting from wafer boundaries, defect elements or local congestion. In this context, a detour is simply an alernative connection, that has been routed around obstacles along the way.

Modeling the Network Topology as a Graph

Graphs provide a natural representation of the L1 network. Topology, defects and current resource utilization can be represented alike. The major advantage of this approach is that algorithms can be developed more generically. They recursively discover possible routes by searching the graph. Whenever a route is established, participating components are removed from the graph to keep an up-to-date map of currently available resources. Decoupling the algorithm from the network topology renders the implementations more flexible and allows for topology explorations of future designs.

The L1 topology is modeled as a undirected graph, because buses have no preferred direction. Events can be transmitted from left to right and right to left alike. Buses are modeled as vertices. Crossbar switches and repeaters connect buses, hence they are modeled as edges. Figure 2.13.6 depicts the mapping between the L1 fabric and graph components.

The corresponding dual graph, where buses are modeled by edges and vertices represent switches and repeaters, is less suitable. This representation splits bus lanes, such that routes can occupy only a fraction of a bus which is physically impossible.

The implementation uses the `boost::graph` template library [67], a popular and efficient C++ template library shipping with many algorithms.

Common Task

Both wafer-routing implementations realize routes iteratively, but use different strategies to discover candidates in the graph. Their commonalities are discussed in the following to focus on their differences later on.

Route Representation A route though the L1 network is defined by its starting point at a single SpL1 repeater, a list of interconnected buses and the vertical target buses on chips with post-synaptic neurons. Note that the injection point on the sending side is-well defined down to the single bus. On the receiving end, any vertical bus running alongside a target chip can in principle be used to relay events into the synapse array. Pending routes, which

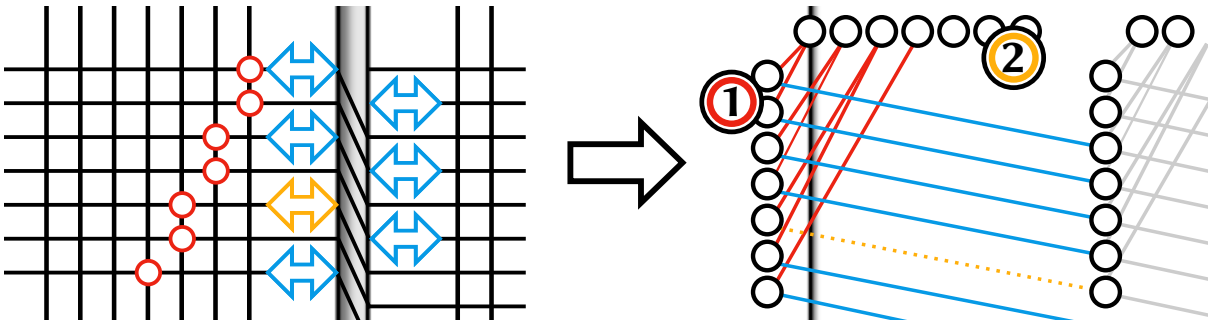


Figure 2.13.6: The L1 routing topology is implemented as a undirected graph. Horizontal **(1)** and vertical **(2)** buses are represented as vertices, while repeaters and switches connecting buses are realized as edges. Defect hardware components, like the yellow repeater, can be left out and are therefore omitted during the wafer routing.

have not yet been routed, are defined as a tuple of a source SpL1 repeater and a list of target chips.

These pending routes are constructed for each source chip, by following all outgoing projections of local neurons to the target populations. Efferent neurons therein are translated back into chips by looking up the neuron placement.

Outgoing connections from all neurons mapped to the same SpL1 repeater have to be routed at once, as L1 resources are taken out of the graph afterwards.

Route Priority Routes are established iteratively. Pending connections that are routed earlier have access to more resources, making their realization more likely. Thus, routing priorities are implemented by scheduling the allocation. By default all pending routes have the same priority. Routes with the same priority are sorted in ascending order by the distance of the source from the center of the wafer. Otherwise, realizing secluded routes with targets on the other end of the wafer first may congest central areas. Typically, this results in increased synaptic losses, since neurons with high out-degree have been placed towards the center.

Modelers can guide the wafer routing by specifying custom priorities for any projections in PyHMF. L1 connections may realize multiple projections. The effective priority is given by the maximum priority of any participating projection.

Parallelization Multiple pending routes compete for the same shared L1 resources. The wafer routing therefore lacks a natural partitioning into independent subproblems, which makes it notoriously hard to parallelize. Even if the access to individual buses is manged by means of process synchronization, such that only a single thread can allocate a bus at a time, competing threads can still steal buses from one-another and thus trap other threads in unresolvable situations. Real world approaches exist for shared-memory parallelization, but at high synchronization costs [32]. For reasons of simplicity, wafer routing is currently carried out purely sequential.

Adjacent Synapse Drivers Events on a vertical bus can be relayed to local synapse drivers and drivers on the adjacent HICANN. In order to simplify the parallelization of the synapse driver routing in the next pipeline step, both implementations avoid configurations where a single vertical bus is used to inject events into both chips at the same time. Otherwise, synchronization is required to ensure that the cumulative capacitive load on the bus does not exceed the total limit. Dropping this confinement yields topologically valid configurations, however capacitive limits are no longer enforced.

Greedy Iterative vs. Global Optimization Finding globally optimal solutions that minimize the over all synaptic loss are unlikely to be found with greedy iterative approaches. However, no efficient algorithms exist to construct the optimal solution as discussed above. Nevertheless, other approaches exist for global multi-objective optimizations, like simulated annealing [44], genetic [31, 50] or swarm [53] algorithms. Generally, the performance of Monte Carlo [51] approaches depends on the efficient generation of candidate solutions and evaluation thereof. Both conditions are not met for the wafer routing, e.g., evaluating the synaptic loss as optimization target requires to carry trough the subsequent two pipeline stages for each candidate solution.

The iterative algorithms implement corrections to attenuate their greediness and leave important resources in foresight to connections routed later in the process, e.g., horizontal buses with pending SpL1 repeaters are avoided. Allocating these buses makes it otherwise impossible to insert their events into the network, rendering them inevitably lost. Further corrections are outlined alongside the implementations.

Iterative Shortest Path Routing

Intuitively, the routing problem can be approached using available and efficient shortest path algorithms after modelling the L1 topology as a graph. A good starting point is Dijkstra's algorithm [20]. Modern implementations find the shortest path between one bus and any other L1 bus in $O(E, V) = |E| + |V| \log |V|$ [28], where E is the set of switches and repeaters and V the set of buses. Here, the stock implementation provided by the `boost::graph` library is used. Note that the problem size decreases over time, as resources are taken out of the graph from iteration to iteration.

Moreover, Dijkstra's algorithm is generic enough to establish simple multi-wafer connections in the future. At the time of thesis submission, a ready hardware implementation of the multi-wafer network has not yet been available.

Graph searches are invoked iteratively for all pending routes according to Section 2.13.2.4. In each iteration, all outgoing connections from a single SpL1 source are routed. Dijkstra's algorithm discovers the shortest paths to any other bus in the graph. The distance between two buses is given by the sum over all weighted edges along the path, thus crossbars and repeaters. This can be used to flexibly model hardware characteristics as explained further below. Shorter paths are discovered earlier in the process due to the greediness of the algorithm. Consequently, the first reachable vertical bus on one of the target chips yields the shortest possible connection to this chip.

Whenever, a connection to one of the pending targets has been found, the actual path is checked to make sure it meets the L1 capacitive constraints. Possible violations are multiple

switches set per bus on the same HICANN to establish chip local detours to e.g., go from one horizontal bus to another on the same chip. The capacitive limit cannot modeled by weighted edges. Instead, backtracking is used to detect and omit these configurations. The candidate is traced from the target to the source bus. Whenever a crossbar switch is encountered, it is checked whether other switches in this crossbar have been used already. If so, the candidate is discarded and the current graph search continued. Otherwise, when the source is reached successfully, all utilized crossbars and buses are both, memorized and allocated. Furthermore, the target chip is removed from the list of pending targets and Dijkstra's algorithm is continued.

The search ends either if all target chips have been successfully routed, no further buses can be reached or the current path exceeds a configurable length limit. The latter avoids routes of last resort, whenever detours across the whole wafer are the only possible option left. When realized, these connections would consume large amounts of L1 resources, which might better be used to realize other routes. The maximum length can be specified as multiples of the L_1 distance between source and the furthest target chip.

The algorithm ultimately terminates when all pending routes have been processed.

Figure 2.13.7 exemplarily illustrates a routing problem that has been solved via the iterative shortest path routing. Notably, the algorithm can solve concave routing problems and therefore reach the chip at item (4).

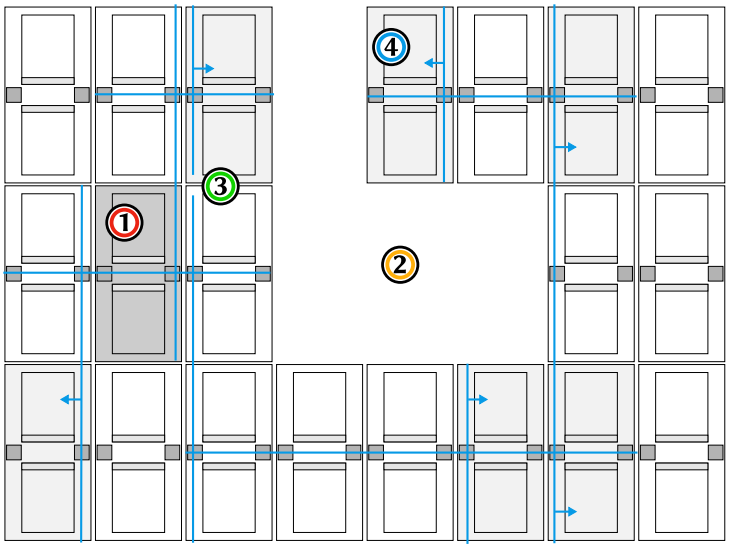


Figure 2.13.7: Routing problem solved via iterative shortest path routing. The source chip **(1)** of the route is highlighted in dark and its targets light grey. Defect chips **(2)** are left out. Following Dijkstra's algorithm the routing searches for shortest path to each target. Depending on edge weights and current allocations nearby chips might be discovered first on different paths, resulting in non-optimal solutions **(3)**. However, targets are reached whenever possible, including the chip at **(4)**, which has been missed by the horizontal growth algorithm in Figure 2.13.8.

Modeling Hardware Characteristics The length of any path established by Dijkstra's algorithm is measured as the sum over all weighted edges along the way, thus it can be used to model hardware characteristics.

These weights are configurable by the modeler to customize the routing. Currently, three different types of edges are distinguished for connections to vertical buses, to horizontal buses and to buses with mapped SpL1 repeaters. By default, the former two are equally set to one. The latter are expensive and therefore set to 10^4, such that Dijkstra's algorithm avoids these buses if other options exist. They become a matter of last resort and are skipped completely if the effective length exceeds the limit. Weights are calculated on-the-fly to emulate directed edges on the undirected graph, safe memory and implement the congestion control as explained below.

Congestion Control Always, picking the shortest possible path to construct routes yields a greedy approach. In fact, Dijkstra's algorithm itself is greedy, it always continues with the shortest unvisited edge and produces optimal results. This is different for resource allocations, where instances are exclusive and iteratively taken away. An optimal routing considers all routes at once and produces a configuration that minimize the overall synaptic loss.

Here, greediness is attenuated by scaling edge weights according to the current bus utilization. Thus, paths through congested areas appear effectively longer. The total weight $w_{\text{total}}(h, i)$ for any edge connecting to a horizontal bus ($i = 0$) or vertical bus ($i = 1$) on chip h is given by

$$w_{\text{total}}(h, i) = w_{\text{static}}(i) + \alpha \cdot \nu(h, i) \quad . \tag{2.13.4}$$

Where $w_{\text{static}}(i)$ models the static characteristics of the hardware, as explained above, and $\nu(h, i)$ denotes the number of allocated buses for chip h and orientation i. The constant α is a scaling factor to control the relative amplitude of the static and dynamic weights. The default is $\alpha = \frac{1}{10}$.

Shortcomings The iterative shortest path approach tends to allocate more resources than necessary. Constructing Minimum Rectilinear Steiner Trees (MRSTs) rather than combining shortest paths would minimize the number of allocated resource per iteration, however it is a Non-deterministic Polynomial-times (NPs) complete problem [43, 30]. Moreover, most MRSTs heuristics require a fixed vertex set [41], whereas pending routes only specify target chips rather than actual target buses. It has been found experimentally, that the routing is mostly limited by horizontal bus utilization. In consequence, the following iterative horizontal growth algorithm has been specifically designed to minimize the horizontal bus allocations.

Runtime In the worst case, the whole graph has to be searched from each source to find every target. Such a behaviour is expected for unstructured random networks, where any two HICANN chips have to be connected. The number of source SpL1 repeater is proportional to the number of neurons N in the model description. The algorithmic complexity is therefore bound by N times the iterative invocation of Dijkstra's algorithm, which has a runtime complexity of $O(E, V) = |V| \log |V| + |E|$ according to the `boost` documentation [66]. Where

V is the set of buses and E the set of switches and repeaters. The backtracking is typically fast and uses internally constant time lookups. It contributes a multiplicative worst case complexity of $|V|$. This results in a overall complexity of

$$O(N, E, V) = N \cdot |V| \cdot (\log |V| + |E|) \quad . \tag{2.13.5}$$

At first glance, the complexity scales linearly in the number of neurons N. However, the hardware extend which is proportional to E and V has to be chosen sufficiently large to host all neurons. For fixed hardware setups, like a single wafer, V and E are bound by the L1 network and the implementation actually scales linearly. For multi-wafer routings, a hierarchical approach can be employed to route individual wafers first and then establish inter-wafer connectivity.

This is only a worst case assessment, sparse network models are typically routed much faster.

Iterative Horizontal Growth Routing

The iterative horizontal growth algorithm is the second wafer-routing implementation. It is optimal in terms of horizontal bus utilization. Similar to the backbone algorithm described in [27] only $N+1$ horizontal buses are required if leftmost and rightmost HICANN are horizontally N chips apart. This algorithm however, does not depend on a single horizontal backbone, which renders the approach suitable for defects, congested areas and non-rectangular routing grids.

Routes grow horizontally until either the outer most chip has been reached or the horizontal continuation of the path is blocked. In the latter case, a vertical detour is established to continue the horizontal growth in another HICANN row. Furthermore, the algorithm is aware of SpL1 utilization and avoids using horizontal buses that are required for route realizations later on. This helps to find globally more optimal solutions. Figure 2.13.8 shows a routing problem solved by horizontal growth. Note that the algorithm does not change directions as the connection grows, therefore concave neuron placements cannot be routed.

Like the previous shortest path algorithm, the horizontal growth algorithm is invoked iteratively for all SpL1 sources according to Section 2.13.2.4. The algorithm itself is implemented recursively. For each pending route, first the horizontal extent $X_{\text{right}} - X_{\text{left}}$ is determined for the left and right most target chips located at X_{left} and X_{right}, respectively. Starting from the horizontal bus at the source SpL1 repeater, the route grows left and right until the outermost extends X_{left} and X_{right} are reached or no more suitable detours can be found.

In each recursion, outgoing edges from the current vertex are examined to find the subsequent horizontal bus on the adjacent chip towards the direction of growth. If the corresponding bus is found and no SpL1 inputs are assigned, the recursion continues, otherwise a vertical detour needs to be established. Several reasons can lead to this bus not being found, like prior allocation, blacklisting or it simply does not exist at the wafer boundary.

In case of a detour, all vertically reachable buses on the current chip are considered. The best possible option is determined by walking vertically for each option until the wafer boundaries on top and bottom are reached or the path is blocked. Then, individually count the number of horizontally reachable target chips in the direction of growth for every cross-

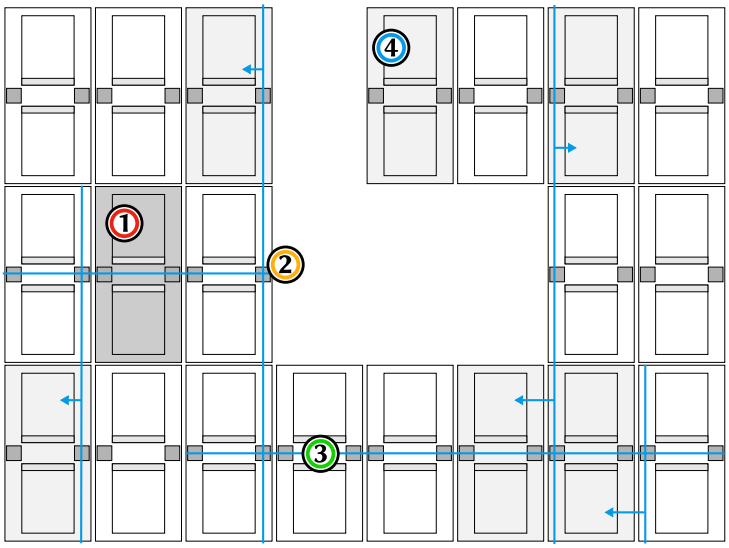


Figure 2.13.8: The same routing problem as in Figure 2.13.7 is solved by means of the iterative horizontal growth algorithm. The source chip **(1)** of the route is depicted in dark, while requested targets are colored in light grey. Defects **(2)** are vertically detoured to continue growth in another HICANN row **(3)**. The target chip at **(4)** is unreachable, because the algorithm keeps growing in a single direction.

bar switch along the way. Consistently, buses with mapped SpL1 repeaters are skipped. Furthermore, the evaluation does not take any detouring into account. The option that yields the maximum number of reachable targets, if any, is chosen to establish the detour accordingly. Subsequently, the growth continues in the HICANN row accessed by the detour.

Whenever a chip is reached, which is in the same column as any of the targets, a vertical connection is established. Again, all vertical buses reachable via local crossbar switches are considered. The best option is picked based on a score. For each option, the score is initially set to zero and a vertical walk is started, both upwards and downwards. When a target chip is encountered, the score is increment by two if less than 12 other routes yet project on the same set of synapse drivers and otherwise by one. In principle there are 16 vertical buses that target the same 14 drivers. The scoring has proven itself useful to reduce the number of competing routes.

The horizontal growth terminates when either all targets have been reached or no more viable detours can be found.

The wafer routing terminates after all pending routes have been processed.

Runtime In the worst case, connections have to be established from any chip to any other and detours have to be found for each horizontal recursion. The number of iterations is proportional to the number of SpL1 sources which in turn is proportional to the number of neurons N in the model. The horizontal growth itself scales linearly with the number of buses in the system $|V|$. Detouring requires to consider a limited number of vertical options

and to count the horizontally accessible targets. During the evaluation, none of the buses is considered twice, thus limiting its complexity by another factor of $|V|$. The total complexity can then be expressed as

$$O(N, V) = N \cdot |V|^2 \quad . \tag{2.13.6}$$

Here, V is limited by the area of actively used chips on the wafer and cannot grow beyond a single wafer. If neurons are placed all over the wafer, V is maximized and implementation scales linearly with the size of the model network.

Evaluation

In most cases, the iterative horizontal growth performs better. It has been specifically designed to cope with the shortcomings of the shortest path approach by reducing the horizontal buses allocations.

Though, the shortest path approach might still be useful for inter-wafer routing and in situations where discovery towards a single direction is not enough, like concave routing areas.

2.13.2.5 Synapse Driver Routing

Input: An assignment of pre-synaptic neurons to vertical buses for each target chip.
Output: An assignment of vertical L1 buses to synapse drivers.
Defects: Blacklist of synapse drivers

In the previous step, the wafer routing has been established. Now, events passing by chips with local targets have to be relayed into the synapse array. For each vertical bus there are 14 reachable drivers. However, 15 other vertical buses on each chip compete for the same resources. Moreover, drivers can mirror their inputs to the adjacent top and bottom drivers to map larger fractions of the address space and implement more synapses per input. Buses running on either side of the chip can only be accessed by drivers on the respective side, therefore the synapse driver assignments on both sides can be optimized individually.

For experiments with small input counts per chip, multiple suitable configurations exist that equally realize all pending connections. For larger experiments with high input counts on the other hand, drivers are the limiting resource. Finding optimized assignments is important to avoid inhomogeneous synaptic losses. Locally unrouted connections result in all afferent synapses from the source being lost and the corresponding L1 resources might have been occupied in vain.

A simpler variant of the problem, where vertical buses are connectible to arbitrary drivers, is algorithmically a Knapsack optimization problem, which is known to be NP-hard [30]. The actual problem is even more complex. Incoming routes can relay events only via a subset of synapse drivers due to the select switch sparseness. Furthermore, any synapse driver can be blacklisted or the connection to neighbors disrupted.

Finally, capacitive limits restrain the amount of inter-connectible drivers. Exceeding limits results in unreliable event delivery.

Counting Model Synapses and Required Synapse Drivers

As a starting point for the synapse driver routing optimization, the theoretical number of drivers required to realize all connection is calculated per incoming connection. This number depends on: the mapping of afferent neurons to the 6 bit address space, model projection properties and actual connectivity between pre and post-synaptic neurons. The address space can be mapped efficiently to synapse rows as well as columns with odd and even index according to the two stage decoding scheme. Different Short-term plasticity (STP) parameters on the other hand are more expensive and require dedicated synapse drivers.

The access granularity of synapses urges to count the number of necessary half synapse rows first. A half synapse row is defined as all synapses within one row receiving the same strobe signal. For any route r all incoming connections within the same connection bin $b = (t_{e/i}, 2\,\mathrm{MSB})$ can be realized via the same half synapse row. Where, b bins incoming connections from sources with the same 2 MSB address bits and the same synapse type $t_{e/i}$ (either excitatory or inhibitory). The number of necessary half synapse rows L for bin b, route r, efferent neuron j and STP parameter set P_{STP} is given by

$$L_r(j, b, P_{STP}) = \left\lceil \frac{1}{2w_j} \sum_{i \in b} N_r(i, j, P_{STP}) \right\rceil \quad . \tag{2.13.7}$$

On the right hand side, $N_r(i, j, P_{STP})$ denotes the number of synapses between afferent neuron i and efferent neuron j using the same instance of STP parameters. The number of synapses that share the same configuration is achieved by summing over all sources within b. Dividing this by two times the physical width w_j of the target hardware neuron j and rounding it to the ceiling results in the number of necessary half synapse rows $L_r(j, b, P_{STP})$.

Every synapse driver has access to four half synapse rows. Thus, the number of synapse drivers per STP parameter set is given by the sum over all bins b of the maximum number of half rows required for any of the target neurons j. Subsequently, the results has to be divided by four and rounded to the ceiling. To finally get to the number of actually necessary drivers D_r we have to sum over all STP parameter sets as they can only be implemented per driver.

$$D_r = \sum_{P_{STP}} \left\lceil \frac{1}{4} \sum_{b} \max_{j} L_r(j, b, P_{STP}) \right\rceil \quad . \tag{2.13.8}$$

If D_r drivers are allocated for r, theoretically all synapses can be realized. However, the calculation can neither include blacklisted drivers nor synapses, because they have not yet been assigned. Consequently, some synapses might still get lost in the subsequent synapse array routing. To circumvent the issue for low input counts, additional synapse drivers can be requested. Moreover, D_r can become larger than capacitive limits allow. Then, synapses are inevitably lost.

Note that using a wide variety of different STP parameters has an immediate impact on the number of drivers and ultimately the loss of model synapses. Thus, merging similar STP parameter sets for projections targeting the same neurons is recommended.

Finally, the total number of synapses, represented by route r, is determined to prioritize

those, which contribute many synapses. The synapses are counted according to

$$N_r = \sum_{j,b,P_{\text{STP}}} \sum_{i \in b} N_r(i, j, P_{\text{STP}}) \quad . \tag{2.13.9}$$

Iterative Best Fit

The iterative best fit algorithm is similar to the synapse driver routing presented in [27]. The drivers are iteratively assigned to pending connections in in a first-fit decreasing fashion. This heuristic is known to yield close to optimal results for bin packing problems [21]. In case the overall amount of necessary drivers per chip exceeds the number of available ones, the problem can no longer be approximated as bin packing. Thus, the driver requirement needs to be rescaled to fit the topology. Otherwise, allocating drivers in a first-fit fashion might not leave drivers for routes later on.

Assignments can not necessarily allocated side by side due to the select switch sparseness fragmenting the driver banks. Consequently, connections may get lost even though the total claim is equal or less than the number of drivers.

Compared to [27], the implementation works on the up-to-date hardware topology. Furthermore, only the necessary number of drivers or the capacitive allowed amount are assigned rather than all available driver to ensure reliable communication. In a final step, gaps in the synapse driver routing left by the bin packing are filled, as far possible, with connections ruled out during in the initial normalization stage.

Normalization In cases, where the overall number of required synapse drivers exceeds the number of mappable drivers per chip, the requirements have to be scaled down in order to approximate the problem as bin packing. Afterwards, we can use the first-fit decreasing heuristic to derive optimized solutions.

The normalization is carried out independently for the left and right side of the chip. Initially, the numbers of required drivers are collected for all incoming routes r on all 256 connectible vertical buses. Note that half of which are on the adjacent HICANN chip. If the sum $D = \sum_r \min(D_r, D_{\text{max}})$ exceeds 112 minus blacklisted drivers, the subsequent rescaling is used. Here, D_r refers to the number of necessary drivers for route r and D_{max} to the capacitively tolerable number of synapse drivers. The normalization is carried out according to

$$D'_r = \begin{cases} 0 & \text{if } D_r = 0 \\ max(1, min(D_r, D_{\text{max}}, \lfloor A \cdot N_r / N_{\text{total}} \rfloor)) & \text{else} \end{cases} \quad , \tag{2.13.10}$$

with the number of synapses N_r represented by route r, the total number of model synapses N_{total} to be realized on the local chip and the number of available synapse drivers on the current side A, which is 112 if no drivers have been blacklisted.

After the normalization, it is possible that $K = \sum_r D'_r$ is still larger than A due to the maximum function in Equation (2.13.10). Then D'_r are iteratively decremented in descending order of $\Delta_r = D'_r - A \cdot N_r / N_{\text{total}}$ until $K = A$, but only if $\Delta_r > 0$. Note that routes r that contribute only a few synapses might end up with $D'_r = 0$.

Alternatively, it might happen that K is less than A after the normalization, mainly for two reasons. Either D_{max} times the number of routes is less than A or when routes contribute only a few synapses but still require many drivers due to e.g., different STP parameter sets. In the latter case, the assignments are increased in the same order as before for all $\Delta_r < 0$ until either $K = A$ or all pending routes r have either $\min(D_r, D_{max})$ drivers.

Driver Assignment Initially, all routes are ordered in a list according to their normalized requirements D'_r from many to few. Then two arrays with 56 entries each are initialized to represent the synapse driver banks, where entry j represents the jth driver in the top and bottom half respectively. Each entry marks a driver as either available or taken. In the beginning, all drivers, except those in the blacklist, are marked as available.

Then the actual iterative assignment starts. In each iteration, the front most route r is popped from the list. The algorithm considers all possible assignment options in order to find the best insertion point. For each vertical bus up to 14 reachable drivers exist. Furthermore, an assignment of n drivers can be shifted around the insertion point p. The drivers are connectible as long as $p \in [x, x + n)$.

First, all 14 options are checked whether they provide D'_r available adjacent synapse drivers. If more than one candidate exists, the first in the smallest gap of available drivers is chosen in order to minimize fragmentation. Moreover, the assignment is shifted such that the distance between the current assignment and the closest assignment is minimized, leaving the smallest possible gap. Otherwise, if no location with a sufficient amount of resources exists, the location with the most remaining resources is chosen. In cases where none of the candidates has resources left anymore, the connections cannot be realized and the synapses are marked as lost. Each assigned driver is further marked as taken in the array and is no longer available. The iteration terminates, when the route list runs empty.

Afterwards, routes that have been dropped during normalization, because D'_r has been reduced to 0, are scheduled for insertion. Firstly, these routes are sorted by their original driver requirement D_r in descending order. Then, the same algorithm sketched above is used to assign synapse drivers for these remaining routes. The algorithm terminates after the assignment finishes.

Simulated Annealing

missing

2.13.2.6 Synapse Array Routing

Input: Synapse driver routing for each chip.
Output: A mapping between the address space of incoming events to synapse rows and an assignment of model to hardware synapses.
Defects: Blacklist of synapses.

The synapse array routing completes the network configuration. First, pre-synaptic neurons are assigned to half synapse rows and model synapses are assigned to hardware synapses afterwards.

Synapse Row Assignment

The first step establishes synapse driver configurations in order to assign sources to synapse rows. Synapses are selectively listening on events according to the 2 MSB decoder configuration at the synapse driver. Therefore, 16 sources can share half the synapses in a row. Moreover, synapses that implement STP have to be assigned to appropriate synapse drivers. However, support for STP is not yet implemented at this pipeline stage.

The implementation assigns half synapse rows according to the relative amount of sharable model synapses to minimize the synaptic loss. Therefore, we recall the binning $b = (t_{e/i}, 2\,\text{MSB})$ previously introduced for the synapse driver routing (see Section 2.13.2.5). A bin contains all sources with the same 2 MSB address bits and the same synapse type $t_{e/i}$, which is either excitatory or inhibitory. Synapses falling into the same bin can be realized by a shared half synapse row.

Note that the number of assignable half synapse rows per driver depends on the implementation size of the target neurons. For small configurations with no horizontally interconnected neuron circuits only two of four half synapse rows can be assigned. Furthermore, if the horizontal neuron extend is odd, the number of synapses for strobe signal A and B as well as C and D is different and depends on the offset of the hardware neurons on the chip. Even though the mapping supports variably sized neurons, it is recommended to use neuron sizes that are multiples of 4, thus ensuring even horizontal neuron extends. Otherwise, additional synapses can be lost during the following assignment.

Synapse Assignment

In a second step, model synapses are assigned to hardware paragons in the order they appear in the model description. Meaning that synapses of efferent neurons with a lower id have higher static priority and are more likely to be realized. The order does not influence the number of lost synapses, only which synapses are lost.

Initially, all synapse weights are set to zero and all decoders are programmed to 0001_2 as explained in Section 2.13.2.2. Therefore, leakage conductance onto the neuron membrane is minimized for weights that are not actually in use.

Afterwards, the connections are iterated in a target index first manner. Therefore, as soon as the system runs out of synapses for a source-target combination all synapses from sources within the same bin b can be marked as lost.

Moreover, a lookup table is used to find the next free hardware synapse for b efficiently in constant time. After each assignment the corresponding row and column entry has to be updated. First, the column index is increased by two until the column index no references a neuron circuit not part of the same model neuron. Then the row index is increased to point to the next half synapse row assigned to b and the column index is reset to the left most synapse in the new row. If no more free rows exist, the synapses as well as all other remaining synapses within b are marked as lost.

The actual translation of model conductances into digital weights is described in Section 2.13.2.7.

Runtime

The implementation scales linearly with the number of local synapses. All synapses have to be processed, either realized or not, in order to track the synaptic loss. The weight lookup ensures that each hardware synapse is handled only once and available synapses are found efficiently in constant time. Again, only chip local resources assigned, thus synapse array routing is carried out in parallel for multiple chips.

2.13.2.7 Parameter Transformation

Input: Neuron placement, synapse mapping and calibration data; Calibration data and limited parameter ranges.
Output: Digital and analog parameter configurations hardware neurons and synapses.
Defects: None

The final step in setting up the hardware is to specify all remaining analog and digital parameters. Some are set to defaults to realize a reasonable regime for the chip to work. Others have to be chosen to resemble the model as closely as possible. The actual parameter transformation is not part of the mapping framework, it has been factored out into the calibtic calibration framework. Thus, other workflow components can use the transformations and calibration and data alike, e.g., StHAL (see Section 2.11.3) uses calibtic to correct recordings of analog voltage traces.

Parameters are transformed in two steps. Firstly, they are scaled from model into hardware domain. Afterwards, a circuit specific correction is applied, which is referred to as calibration. The scaling from model into hardware domain as well as the following correction are both implemented within the calibtic framework. Here, the basic scalings of voltages, time, conductances and currents are presented, though they have mostly been described in [64]. The generation of calibration data is beyond the scope of this thesis. It is discussed in more detail in [64].

Calibration Framework

The calibtic frameworks is an extensible C++ library that provides storage agnostic access to calibration data. calibtic has been established as part of this thesis. At the time of writing, it has been used to implement the Analog-to-Digital Converter (ADC) analog readout calibration as well as a redesigned calibration for HICANN analog parameters.

calibtics biggest advantage is that it stores the transformation rule alongside the actual data. Consequently, transformations can be changed or updated as needed, whereas old data sets remain consistent and applicable without adding new code paths. Furthermore, calibtic is used for data acquisition and the application of calibrations alike. Python interfaces are provided for the former to integrate well with the StHAL and HALbe based calibration routines. Together, this is a major improvement over former approaches. Previously making a small change on the acquisition side could render existing data recording useless or application thereof inconsistent.

Moreover, PyHMF has been integrated with calibtic, allowing to conveniently transform model parameters sets into hardware floating-gate values.

Finally, a dynamic plug-in system adds flexible storage options. At the time of thesis submision, storage backends existed for `XML`, `JSON` and mongoDB. The latter two are realized by means of the generic mongoDB `boost::serialization` archive written by the author.

Scaling of Individual Parameters

The first step of parameter transformation maps the model parameter range linearly onto parameters accessible on hardware. For example, the voltage range for the Adaptive Exponential Integrate-and-Fire (AdEx) model is typically in the range of 0 mV to -80 mV. The hardware, on the other hand, works with voltages from 0 V to 1.8 V. Moreover, small intrinsic capacitances C and conductances g lead to shorter time constant $\tau = \frac{C}{g}$. The dynamics of the system therefore evolve 10^3 to 10^5 times faster than the biological prototype.

Time and Time Constants Following the definition of the speedup factor α_{acc}, times and time constants are linearly scaled according to

$$\tau_{\text{model}} = \tau_{\text{scaled}} \cdot \alpha_{\text{acc}} \quad . \tag{2.13.11}$$

Note, the effective speedup can be controlled and is subject to calibration. As mentioned before, the system is designed to allow acceleration factors within the range of 10^3 and 10^5. Currently, the default factor for scaling as well as calibration is set to 10^4.

Voltage Ranges Voltages are transformed linearly from mV in the model to Volts in the hardware domain, according to

$$v_{\text{scaled}} = \alpha_v \cdot v_{\text{model}} + v_{\text{shift}} \quad . \tag{2.13.12}$$

Here, α_v is a linear scaling factor and v_{shift} an offset, that can be used to shift voltage ranges relative to one another. Most model dynamics depend on voltage differences, thus eliminating a common shift. However, voltage parameters can be shifted in relative to the shared V_{reset} parameter. The parameters α_v and v_{shift} can be chosen freely to map a given dynamic range. By default, α_v is set to 10 and v_{shift} to 1200 to map the biological range of $[-120, 0]$mV to $[0, 1.2]$V in hardware.

The voltage transformation applies for all voltages including reversal potential E_{l}, E_{e}, E_{i}, the reset potential V_{reset}, the spike initiation voltage V_{thresh} and the spike detection voltage V_{spike}. One exception is the slope factor Δ_T which is only scaled but not shifted.

Membrane Capacitance On hardware, two configurable membrane capacitances are available, 2.16 pF and 0.16 pF, to support a wider range of possible speedup factors. Typically, the larger capacitance is chosen except for high speedup factors of around 10^5. The total membrane capacitance for interconnected neurons is the sum over all individual membrane capacitances, following

$$C_{\text{m}} = \sum_i^N \cdot C_{\text{m,i}} \quad , \tag{2.13.13}$$

where N is the number of connected neuron circuits and $C_{m,i}$ the individual capacitances. In principle, any combination of capacitances is possible, however, at the time of thesis submission, no calibration existed for interconnected neurons.

Conductances The time constant for the leakage dynamics of the neuron τ_m is controlled via the conductance g_{leak} as follows

$$\tau_m = \frac{C_m}{g_{leak}} \quad . \tag{2.13.14}$$

The effective membrane capacitance C_m has been introduced above. This equation can be rephrased according to Equation (2.13.11) as

$$g_{leak,scaled} = \frac{C_{m,scaled}}{\tau_{m,model}} \cdot \alpha_{acc} \tag{2.13.15}$$

$$= \frac{C_{m,scaled}}{C_{m,model}} \cdot \alpha_{acc} \cdot g_{leak,model} \quad . \tag{2.13.16}$$

With the total hardware and model membrane capacitance $C_{m,scaled}$ and model $C_{m,model}$, respectively. The acceleration factor α_{acc}, the model membrane time constant $\tau_{m,model}$ and model leakage conductance $g_{leak,model}$.

This transformation for leakage conductances identically applies for any other conductance, like synaptic efficacies and the adaption coupling parameter a. Generally, leaving us with

$$g_{scaled} = \frac{C_{scaled}}{C_{model}} \cdot \alpha_{acc} \cdot g_{model} \tag{2.13.17}$$

for arbitrary conductances.

Currents Currents are transformed according to Ohm's law using the previous transformations for voltages and conductances

$$I_m = g \cdot U = \frac{C_{scaled}}{C_{model}} \cdot \alpha_{acc} \cdot \alpha_v \quad . \tag{2.13.18}$$

With model current I_m, leakage g and voltage U. This transformation applies for e.g., the AdEx adaption current b and input currents.

Shared Parameter Scaling

The transformation of parameters shared by multiple circuits require special considerations.

Neurons Neurons share a common V_{reset}, which is in fact the only shared AdEx model parameter. All other analog neuron parameters be set and calibrated individually for each neuron. The other voltages are therefore shifted by v_{shift} in Equation (2.13.12) to restore the voltage difference ΔV for the other voltages and consequently preserve the model dynamics.

In principle, there are 4 different shared V_{reset} voltage sources. These values are assigned to neurons with odd and even index in the top and bottom synapse array. Typically, four or more neurons are interconnected, spanning all instances of V_{reset}. Then, only a single value can be used.

Synapses Synaptic input currents are implemented via conductance-based synapses on hardware. Conductances are scaled according to Equation (2.13.17), where the ratio of c_{scaled}/c_{model} depends on the neuron interconnection size. The effective strength of a synaptic conductance in hardware is given by

$$g_{syn}(i,j) = g_{max}(j) \cdot g_{digital}(i,j) \quad . \tag{2.13.19}$$

Where $g_{syn}(i,j)$ is the conductance of the ith synapse in the jth column. It is the product of a driver-wise conductance $g_{max}(j)$ and an individual 4 bit digital weight $g_{digital}(i,j)$. The shared parameter g_{max} can be chosen from 4 configurable values. Ideally, g_{max} maximizes the dynamic range for all synapses operated by a single driver. Experiments including Spike Timing Dependent Plasticity (STDP) might choose g_{max} such that the maximum row-wise model conductance corresponds to a digital weight of 8. Synapses can subsequently be potentiated upon learning. However, g_{max} is currently set to a fixed default value, as no calibration exists and the synaptic input circuits saturates early.

After choosing g_{max}, the digital weights are set such that the resulting conductance resembles the scaled model conductance as closely as possible. Weights have a 4 bit resolution, thus only 16 discrete conductances are accessible. In order to minimize average distortion, weights are clipped stochastically. For a target conductance g_{scaled}, the closest two accessible conductances g_a above and g_b below are picked from the list of 16 possible values via branch and bound. Then, g_b is picked with a probability of $p = (g_{scaled} - g_a)/(g_b - g_a)$ and g_a otherwise.

Spike and Current Sources

For spike sources two things have to be done. Firstly, spike trains have to be generated for actual actual spike sources. Secondly, address 0 events have to be merged into each L1 connections to ensure reliable communication.

PyNN spike sources, like SpikeSourceArray and SpikeSourcePoisson, are implemented via FPGA spike playback over the L2 network. Source addresses have been established during the previous input routing. Therefore spike trains can be generated by using the respective address and translating spike times according to Equation (2.13.11). For spike source arrays the spike times are simply provided by the model description. For poisson spike source with rate ν a corresponding train has to be constructed according to [34]. Therefore, time is discretized into small intervals $\delta t = 1\,\text{ms}$. For each interval, a uniformly distributed random number x between 0 and 1 is generated. Whenever $x < \nu \cdot \delta t$, a spike occurs in the interval.

Moreover, a small but fixed offset is added to each spike time, which controls the onset of the actual experiment. Currently, the offset is set to a few nanoseconds to allow the L1 resources to synchronize reliably.

Address 0 Events from the chip-local background generators are merged into the output of neuron sources to ensure reliable transmission over L1. Generally, the background event generators are used for link synchronization only and not for actual experiments. External inputs are augmented with additional address 0 events.

Current Sources on HICANN can implement periodic step currents of length $T = 1024 \times \delta t$ Model currents are translated according to Equation (2.13.18). However, only a single current per chip can be used, thus using the first in the model description.

Calibration

After the parameters have been scaled into the appropriate hardware range, a calibration is applied to account for individual circuit variations. calibtic allows to implement arbitrary corrections for parameters, which are typically small compared to scaling. For more details on the actual calibration and corrections see [64].

Runtime

The parameter transformation is straight forward and does not involve any complex algorithms. Most HICANN parameters are determined in constant time, rendering the implementation suitable for large networks. Furthermore, parameters are set for multiple chips in parallel.

In practice, spike generation can be rather time consuming, even though the implementations scales linearly with the duration of the train.

2.13.3 PyNN.hardware.nmpm

Author: Eric Müller

PyNN is a simulator-independent API for specifying neuronal network models. Neurons and connections can be grouped into higher-level constructs, statistical measures can be used to describe parameters. The NM-PM software stack implements this API and fits into the list of supported PyNN simulators backends.

A simple network with a Poisson spike source projecting to a pair of `IF_curr_alpha` neurons (cf. [57]):

```
from numpy import random, add
import pyNN.SIMULATOR as sim

sim.setup(timestep=0.1, min_delay=0.2, max_delay=1.0)

cell_params = {
        'tau_refrac': 2.0,
        'v_thresh': [-50.0, -48.0],
        'tau_syn_E': 2.0,
        'tau_syn_I': 2.0
```

```
}

output_pop= sim.Population(2, IF_curr_alpha(**cell_params))

tstop = 1000.0
rate = 100.0
number = int(2*tstop*rate/1000.0)
random.seed(26278342)
spike_times = add.accumulate(
        random.exponential(1000.0/rate, size=number))

input_pop = sim.Population(1,
        SpikeSourceArray(spike_times=spike_times))

projection = sim.Projection(input_pop,
                            output_pop,
                            sim.AllToAllConnector(),
                            sim.StaticSynapse(weight=1.0)
)

input_pop.record('spikes')
output_pop.record(('spikes', 'v'))

sim.run(tstop)

output_pop.write_data('output.dat',
        annotations={'script_name': __file__})

sim.end()
```

The PyNN.hardware.nmpm uses the client-server approach to split the software stack into the PyNN interface implementation and the backend-specific part. On the user side, the client implements the PyNN API and uses an IPC mechanism to trigger experiment execution and retrival of experiment data. Additionally to the PyNN functionality, the IPC layer provides user authorization and authentication.

Part 3

Neuromorphic Computing with Many-core Emulation of Brain Models

3.1 Multi-core Platform: NM-MC

The Neuromorphic Multi-Core Platforms (NM-MC1, NM-MC2) provide cheap, reliable, and readily available platforms on which to perform experiments for the Human Brain Project.

Currently it is envisaged that these experiments fall into two broad areas: those supporting the neuromorphic approach to brain modelling, *i.e.* reduced cortical circuits using point neurons, and neurorobotics experiment; and those exploring features used in the Simulation Platform, *i.e.* virtual environments, whose performance can be explored before the Simulation Platform is ready. The flexibilty of the digital approach to neuromorphics means that if other suitable experiments are required, then this is just a matter of re-programming stock microprocessors.

The Neuromorphic Multi-Core Platform will leverage prior investment by the UK Engineering and Physical Science Research Council (EPSRC) in SpiNNaker technology to provide a half million core machine suitable for brain simulation. The basis of the system is a novel 18 core chip. This component can be incorporated into larger systems because it has built-in inter-chip communications.

The full 500,000 core machine has a total memory capacity of 4Tbytes, and at most six hundred 100Mbit ethernet connections. It is envisaged that this data will not be directly loaded or written back to backing store. Instead, a description of the data will be loaded, which is then expanded on the NM-MC1 system using the full 500,000 cores to perform this expansion.

For check-pointing purposes we currently envisage writing back deltas on the original data-sets. This approach is subject to change, should alternatives present themselves.

The SpiNNaker Group at Manchester have been holding successful SpiNNaker Workshops, and this task will now continue in part funded by the HBP grant. So far there have been three Workshops with twenty attendees per workshop, and a fourth is to be held in April 2014.

3.1.1 Physical Architecture

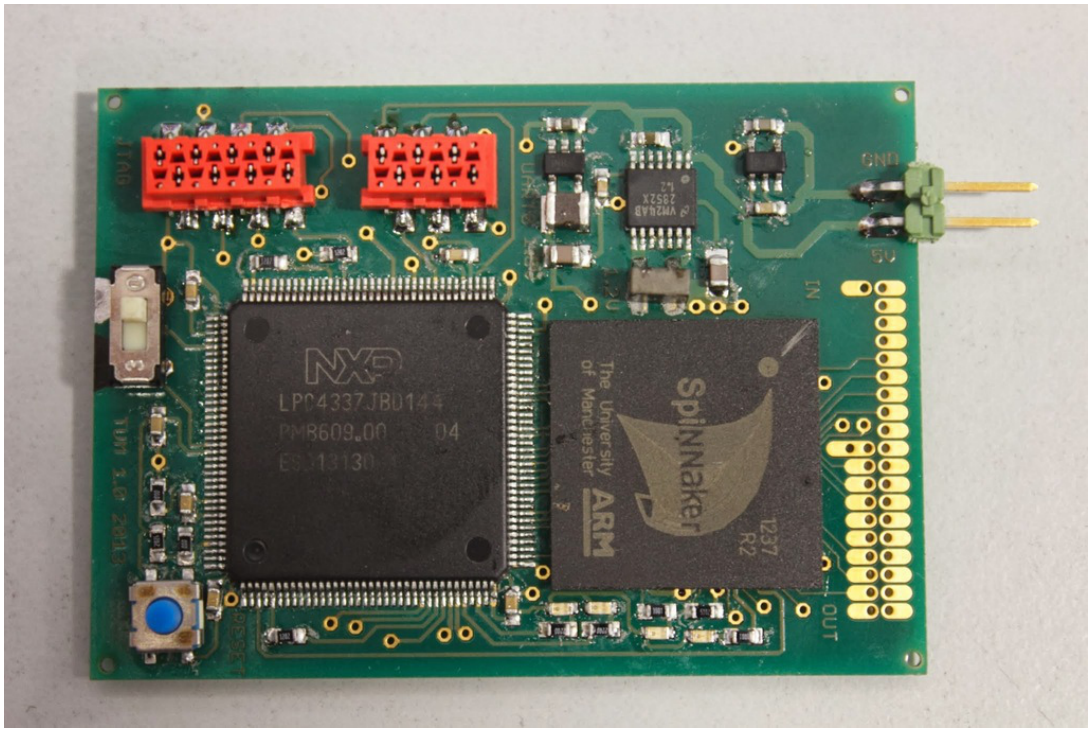
Physical machines used to deploy Platform, locations, etc.

The NM-MC1 system uses the SpiNNaker chip, which is now in production.

Various configurations are possible:

Single node board

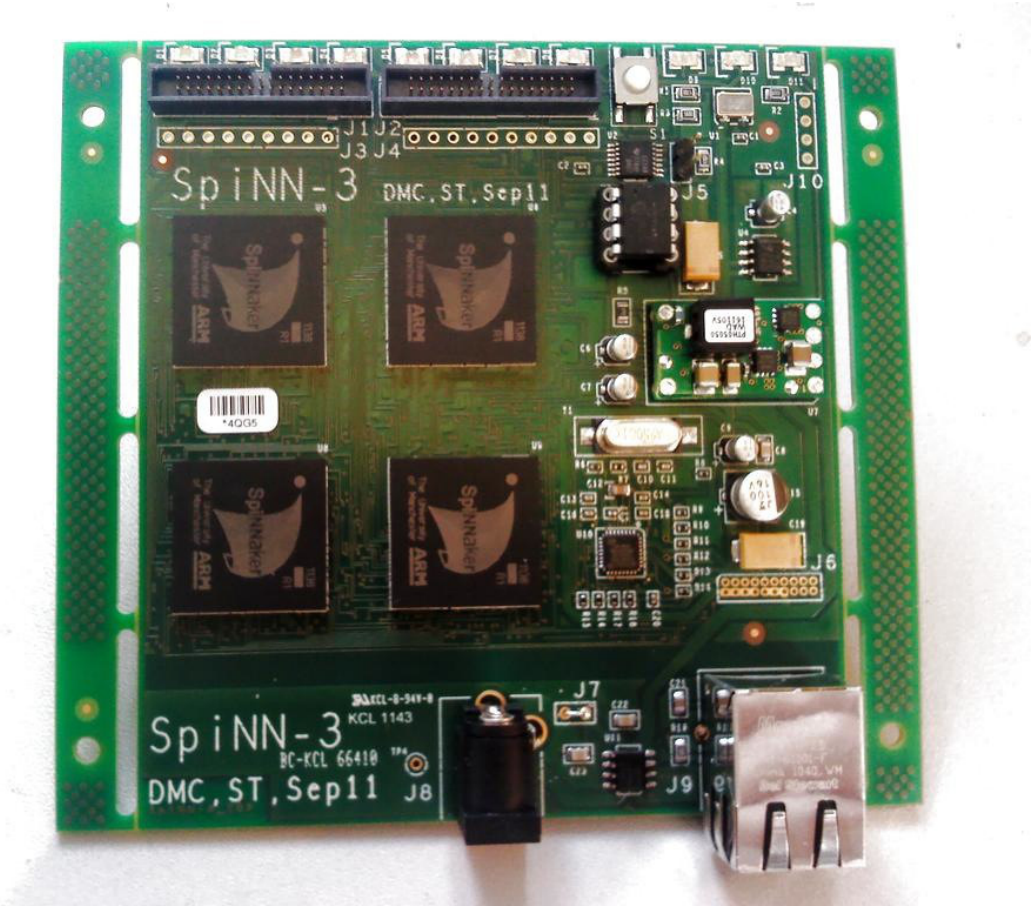
Although this has not been developed in Manchester, it is an interesting Neurorobotics platform, developed by Jörg Conradt at TU Munchen:



Four node board

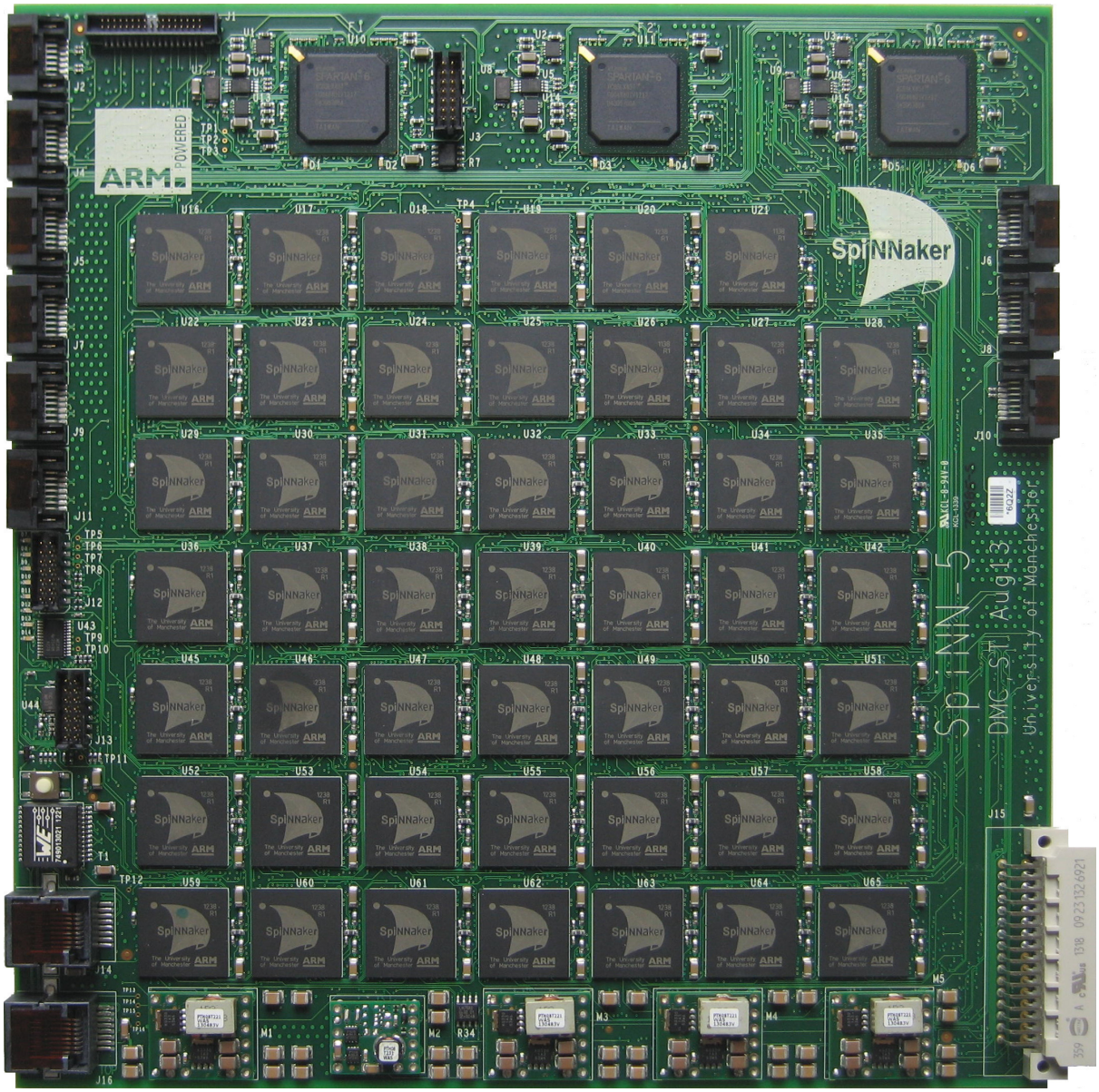
This consists of a four SpiNNaker chips and has been developed by in Manchester for use in collaboration with the Robotics group at Plymouth University. It can be used in an iCUB robot.

Over thirty of these boards are on loan to various groups around the world including many partners within the Human Brain Project.



Forty-eight node board

This is the basis of all large systems. It consumes a maximum of 90W and has FPGA links for connection to other boards. It is configured as an assymetric hexagon, and can thus be tiled easily in groups of three with full toroidal connectivity.



3.1.2 Software

The main part of the software stack consists of two parts:

Host Machine Software There are two main components required here:

PACMAN: Placing and Routing Management This consists of the software which takes a PyNN program and splits the overall task into up to 500,000 small sub-tasks each of which is to run on an individual core of the platform. The precise limit of the splitting is determined by the physical hardware available.

Data Extraction When the simulation has completed, this component is responsible for polling the target machine to find the computed data which the user has indicated is of interest.

This part of the software is written in python.

Target Machine Software This consists of a set of libraries and other components which support the simulation task.

It is split into three parts:

Loading This part of the software is conccerned with loading the model and its parameters.

Simulating This part of the task is concerned with the execution of the simulation. There is some possible overlap with the next part.

Data Extraction This component is concerned with taking the computed data off of the machine and passing it back to the **Host Machine Software**.

This part of the software is written in 'C'.

In addition there is a requirement for debugging, and system monitoring and management; both for the system admintrators and the end-users.

It is envisaged that virtual environment software will be produced by the Simulation sub-project, and that job-control software will be provided by SP9, Task 3.

Progress with the support software for NM-MC1 is not best described by features, but instead on the scale of the systems supported. This is because as the size of the system increases we expect algorithms and data structures which have proved perfectly satisfactory thus far will prove to require re-working as the system size increases.

We therefore envisage the following scales and the months on which they will be delivered.

SpiNNware-103 A system which supports any single board system. The largest board has 48 nodes or chips, and can therefore accept a system with up to 864 processor cores ($\sim 1000 = 10^3$).

Expected Delivery: 2nd quarter 2014.

SpiNNware-104 A system which supports any single subrack system. A subrack can hold up to 24 boards, and can therefore accept a system with up to 20736 processor cores ($\sim 10000 = 10^4$).

Expected Delivery: 4th quarter 2014.

SpiNNware-105 A system which supports any single cabinet system. A cabinet can hold up to 5 subracks, and can therefore accept a system with up to 103680 processor cores ($\sim 100000 = 10^5$).

Expected Delivery: 2nd quarter 2015.

SpiNNware-106 A system which supports any multi-cabinet system. In theory there might be up to ten cabinets, and can therefore accept a system with up to 1036800 processor cores ($\sim 1000000 = 10^6$).

Expected Delivery: 4th quarter 2015.

In conjunction with the software development, and indeed a prequisite to proper performance testing will be the existence of a hardware test bed.

We therefore envisage the following scales and the months on which they will be delivered:

SpiNNaker-103 A system consisting of a single 48 node board.

Delivered: 2nd quarter 2013.

SpiNNaker-104 A system consisting of a single subrack.

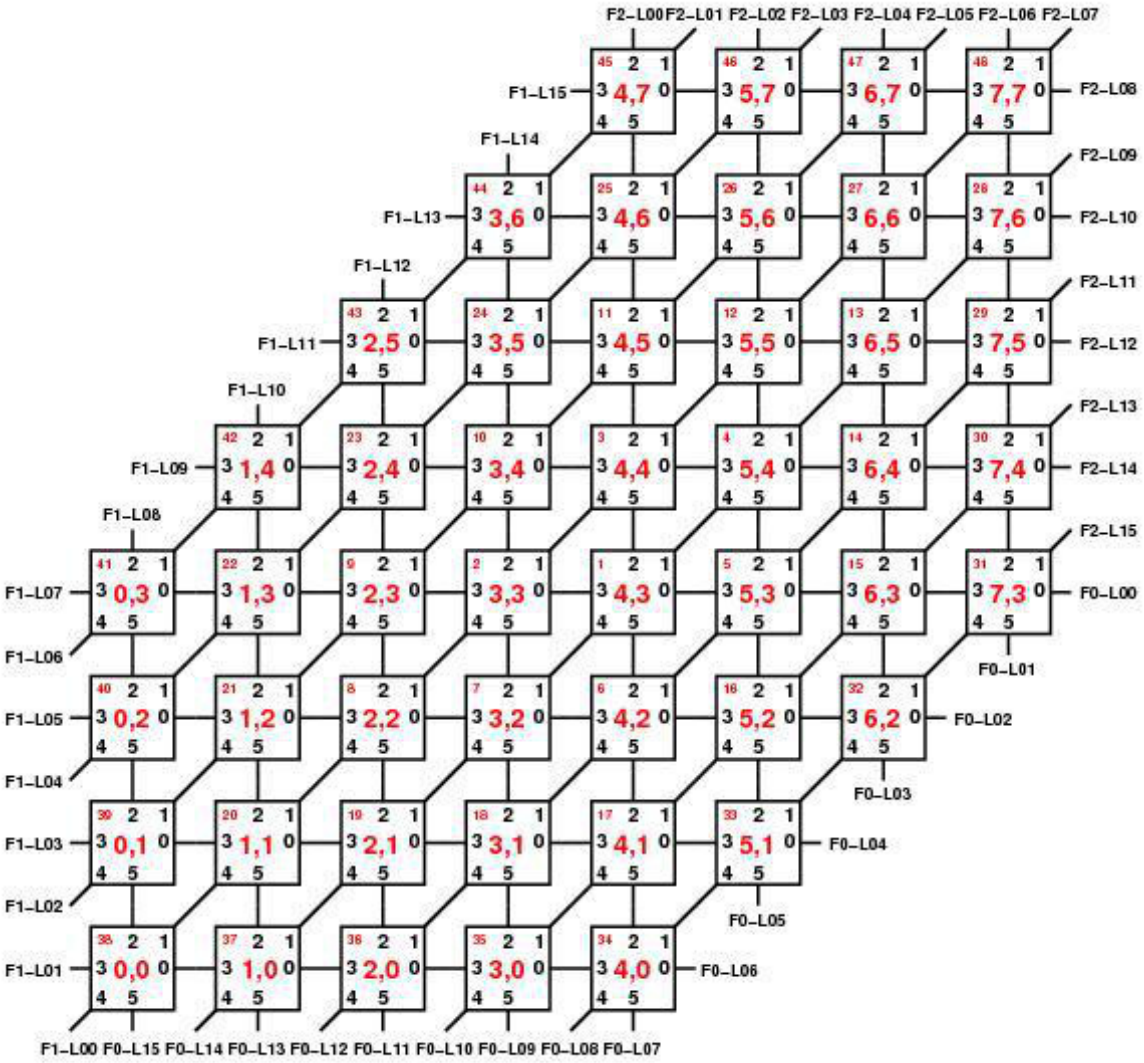
Expected Delivery: 2nd quarter 2014.

SpiNNaker-105 A system consisting of a single cabinet.

Expected Delivery: 1st quarter 2015.

SpiNNaker-106 A system consisting of up to ten cabinets.

Expected Delivery: 3rd quarter 2015.


F2-L00 F2-L01 F2-L02 F2-L03 F2-L04 F2-L05 F2-L06 F2-L07
F1-L15
F1-L14
F1-L13
F1-L12
F1-L11
F1-L10
F1-L09
F1-L08
F1-L07
F1-L06
F1-L05
F1-L04
F1-L03
F1-L02
F1-L01
F2-L08
F2-L09
F2-L10
F2-L11
F2-L12
F2-L13
F2-L14
F2-L15
F0-L00
F0-L01
F0-L02
F0-L03
F0-L04
F0-L05
F0-L06
F1-L00 F0-L15 F0-L14 F0-L13 F0-L12 F0-L11 F0-L10 F0-L09 F0-L08 F0-L07

3.2 SpiNNaker Chip Datasheet

SpiNNaker - a chip multiprocessor for neural network simulation. Datasheet

Features

- 18 ARM968 processors, each with:
  - 64 Kbytes of tightly-coupled data memory;
  - 32 Kbytes of tightly-coupled instruction memory;
  - DMA controller;
  - communications controller;
  - vectored interrupt controller;
  - low-power 'wait for interrupt' mode.
- Multicast communications router
  - 6 self-timed inter-chip bidirectional links;
  - 1,024 associative routing entries.
- Interface to 128Mbyte (nominal) Mobile DDR SDRAM
  - over 1 Gbyte/s sustained block transfer rate;
  - optionally incorporated within the same multi-chip package.
- Ethernet interface for host connection
- Fault-tolerant architecture
  - defect detection, isolation, and function migration.
- Boot, test and debug interfaces.

Introduction

SpiNNaker is a chip multiprocessor designed specifically for the real-time simulation of large-scale spiking neural networks. Each chip (along with its associated SDRAM chip) forms one node in a scalable parallel system, connected to the other nodes through self-timed links.

The processing power is provided through the multiple ARM cores on each chip. In the standard model, each ARM models multiple neurons, with each neuron being a coupled pair of differential equations modelled in continuous 'real' time. Neurons communicate through atomic 'spike' events, and these are communicated as discrete packets through the on- and inter-chip communications fabric. The packet contains a routing key that is defined at its source and is used to implement multicast routing through an associative router in each chip.

One processor on each SpiNNaker chip will perform system management functions; the communications fabric supports point-to-point packets to enable co-ordinated system management across local regions and across the entire system, and nearest-neighbour packets are used for system flood-fill boot operations and for chip debug. In addition, fixed-route

packets carry 64 bits of debug information back to particular nodes for transmission to the host computer.

Background

SpiNNaker was designed at the University of Manchester within an EPSRC-funded project in collaboration with the University of Southampton, ARM Limited and Silistix Limited. Subsequent development took place within a second EPSRC-funded project which added the universities of Cambridge and Sheffield to the collaboration. The work would not have been possible without EPSRC funding, and the support of the EPSRC and the industrial partners is gratefully acknowledged.

Intellectual Property rights

Disclaimer

The details in this datasheet are presented in good faith but no liability can be accepted for errors or inaccuracies. The design of a complex chip multiprocessor is a research activity where there are many uncertainties to be faced, and there is no guarantee that a SpiNNaker system will perform in accordance with the specifications presented here. The APT group in the School of Computer Science at the University of Manchester was responsible for all of the architectural and logic design of the SpiNNaker chip, with the exception of synthesizable components supplied by ARM Limited and interconnect components supplied by Silistix Limited. All design verification was also carried out by the APT group. As such the industrial project partners bear no responsibility for the correct functioning of the device.

Error notification and feedback

Please email details of any errors, omissions, or suggestions for improvement to: steve.furber@manchester.ac.uk.

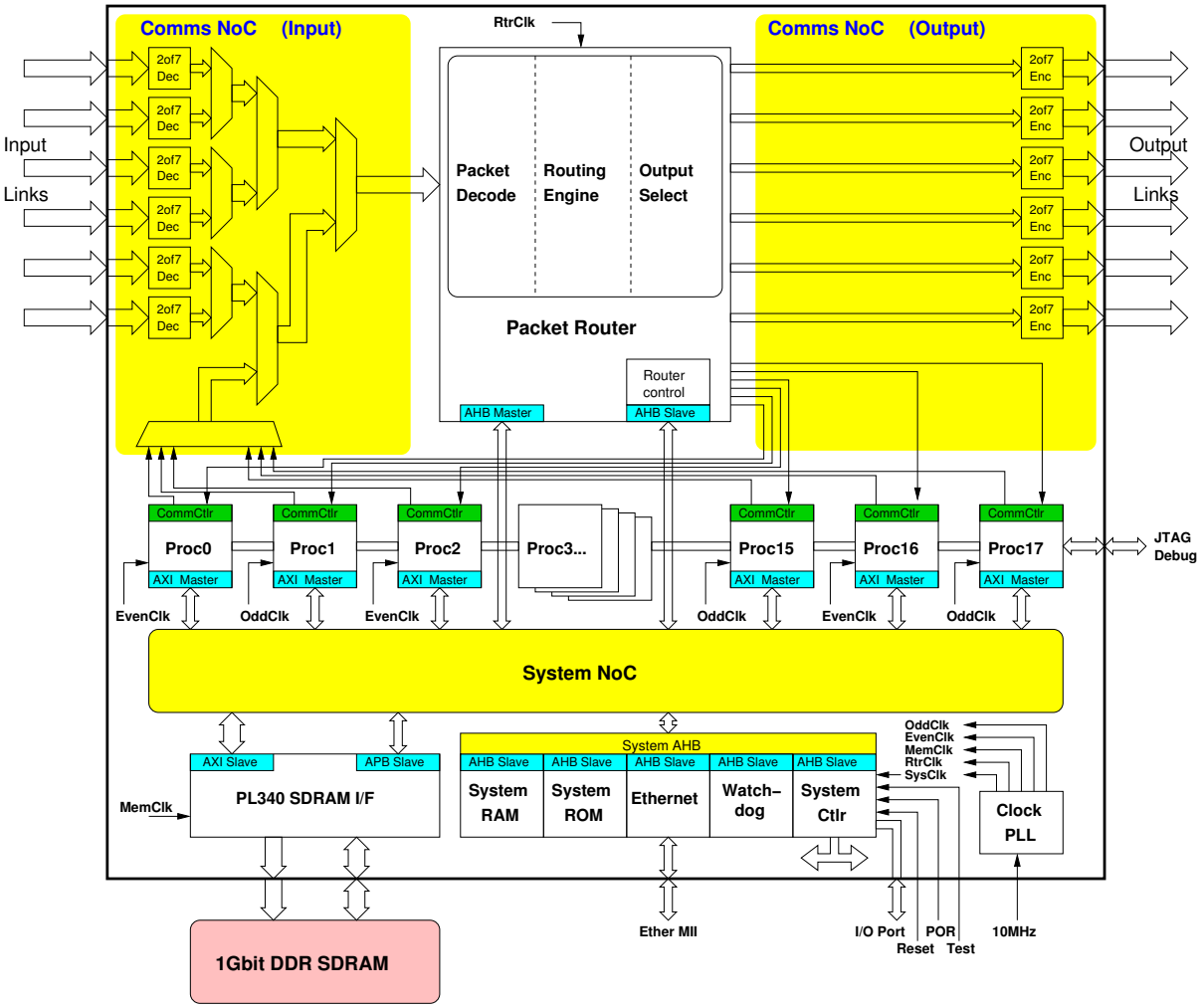
Change history

version	date	changes
2.00	21/4/10	Full SpiNNaker chip initial version
2.01	19/10/10	Change CPU clocks, add package details, minor corrections.
2.02	8/12/10	Detail corrections and enhancements

3.2.1 Chip Organization

3.2.1.1 Block Diagram

The primary functional components of SpiNNaker are illustrated in the figure below.



Each chip contains 18 identical processing subsystems. At start-up, following self-test, one of the processors is nominated as the Monitor Processor and thereafter performs system management tasks. The other processors are responsible for modelling one or more neuron fascicles - a fascicle being a group of neurons with associated inputs and outputs (although some processors may be reserved as spares for fault-tolerance purposes).

The Router is responsible for routing neural event packets both between the on-chip processors and from and to other SpiNNaker chips. The Tx and Rx interface components are used to extend the on- chip communications NoC to other SpiNNaker chips. Inputs from the various on- and off-chip sources are assembled into a single serial stream which is then passed to the Router.

Various resources are accessible from the processor systems via the System NoC. Each of the processors has access to the shared off-chip (but possibly in the same package) SDRAM, and various system components also connect through the System NoC in order that, whichever processor is Monitor Processor, it will have access to these components.

The sharing of the SDRAM is an implementation convenience rather than a functional requirement, although it may facilitate function migration in support of fault-tolerant operation.

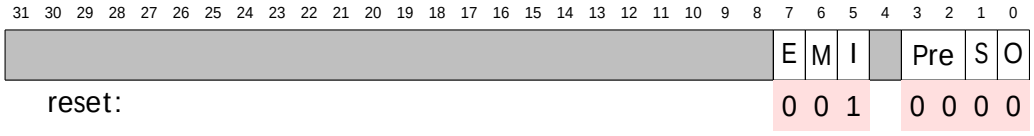
3.2.1.2 System-on-Chip hierarchy

The SpiNNaker chip is viewed as having the following structural hierarchy, which is reflected throughout the organisation of this datasheet:

- ARM968 processor subsystem
  - the ARM968, with its tightly-coupled instruction and data memories
  - Timer/counter and interrupt controller
  - DMA controller, including System NoC interface
  - Communications controller, including Communications NoC interface
- Communications NoC
  - Router, including multicast, point-to-point, nearest-neighbour, fixed-route, default and emer- gency routing functions
  - 6 bidirectional inter-chip links
  - communications NoC arbiter and fabric
- System NoC
  - SDRAM interface
  - System Controller
  - Router configuration registers
  - Ethernet MII interface
  - Boot ROM
  - System RAM
- Boot, test and debug
  - central controller for ARM968 JTAG functions
  - an off-chip serial boot ROM can be used if required

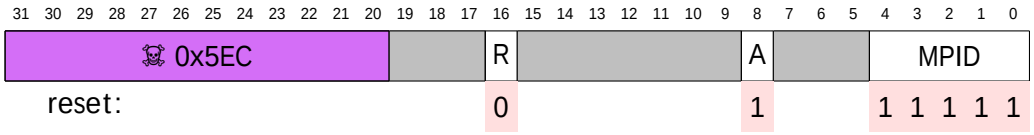
3.2.1.3 Register description convention

Registers are 32-bits (1 word) and are usually displayed in this datasheet as shown below:



- The grey-shaded areas of the register are unused. They will generally read as 0, and should be written as 0 for maximum compatibility with any future functionality extensions.
- Reset values, where defined, are shown against a red shaded background.

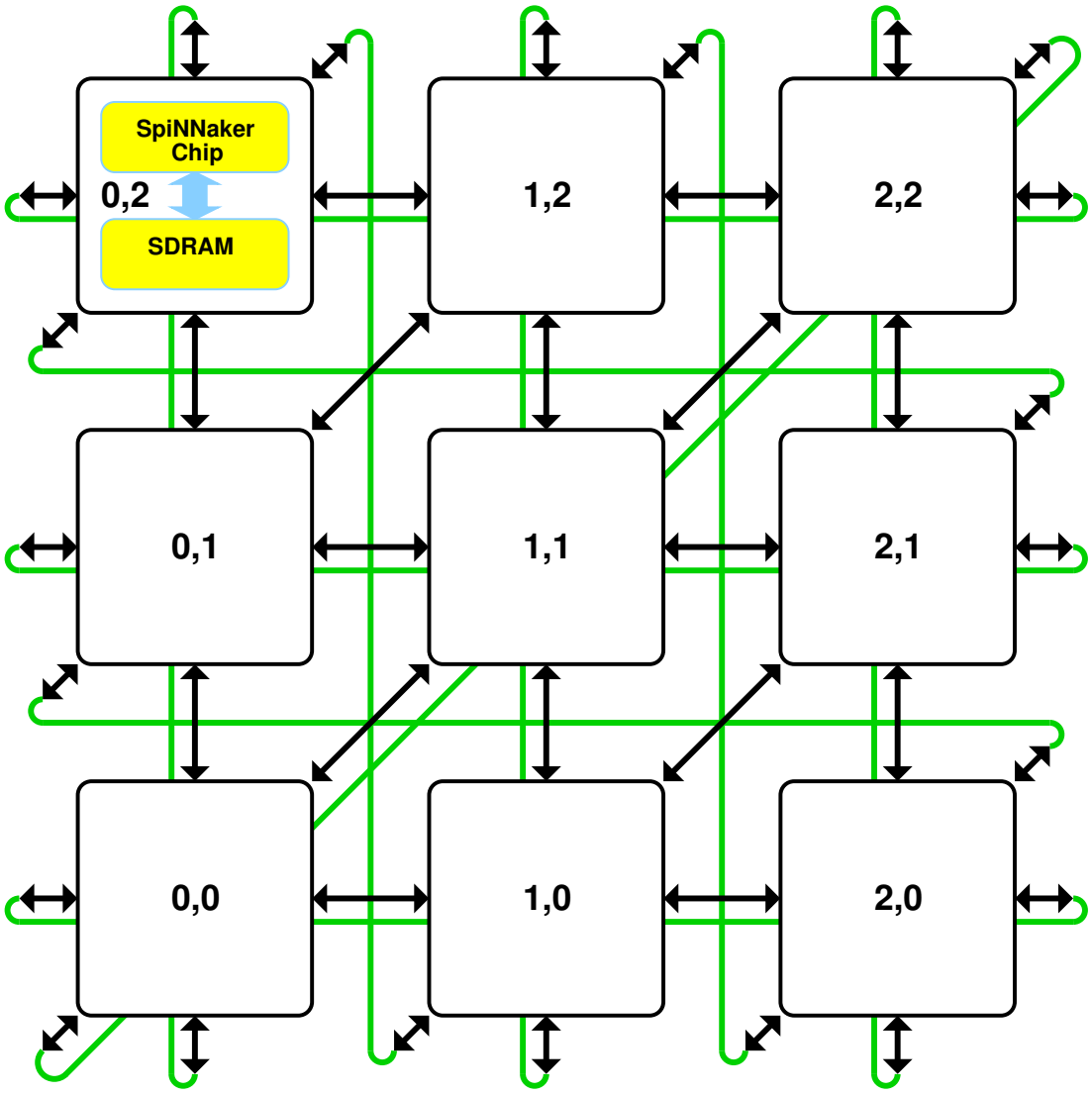
Certain registers in the System Controller have protection against corruption by errant code:



- Here any attempt to write the register must include the security code 0x5EC in the top 12 bits of the data word. If the security code is not present the write will have no effect.

3.2.2 System architecture

SpiNNaker is designed to form (with its associated SDRAM chip) a node of a massively parallel system. The system architecture is illustrated below:



3.2.2.1 Routing

The nodes are arranged in a triangular mesh with bidirectional links to 6 neighbours. The system supports multicast packets (to carry neural event information, routed by the associative Multicast Router), point-to-point packets (to carry system management and control information, routed by table look-up), nearest-neighbour packets (to support boot-time flood-fill and chip debug) and fixed-route packets (to convey application debug data back to the host computer).

Emergency routing

In the event of a link failing or congesting, traffic that would normally use that link is redirected in hardware around two adjacent links that form a triangle with the failed link. This "emergency routing" is intended to be temporary, and the operating system will identify a more permanent resolution of the problem. The local Monitor Processor is informed of uses of emergency routing.

Deadlock avoidance

The communications system has potential deadlock scenarios because of the possibility of circular dependencies between links. The policy used here to prevent deadlocks occurring is:

- **no Router can ever be prevented from issuing its output.**

The mechanisms used to ensure this are:

- outputs have sufficient buffering and capacity detection so that the Router knows whether or not an output has the capacity to accept a packet;
- emergency routing is used, where possible, to avoid overloading a blocked output;
- where emergency routing fails (because, for example, the alternative output is also blocked) the packet is 'dropped' to a Router register, and the Monitor Processor informed;

The expectation is that the communications fabric will be lightly-loaded so that blocked links are very rare. Where the operating system detects that this is not the case it will take measures to correct the problem by modifying routing tables or migrating functionality.

Errant packet trap

Packets that get mis-routed could continue in the system for ever, following cyclic paths. To prevent this all (apart from nearest-neighbour) packets are time stamped and a coarse global time phase signal is used to trap old packets. To minimize overhead the time stamp is 2 bits, cycling 00 → 01 → 11 → 10, and when the packet is two time phases old (time sent XOR time now = 0b11) it is dropped and an error flagged to the local Monitor Processor. The length of a time phase can be adapted dynamically to the state of the system; normally, timed-out packets should be very rare so the time phase can be conservatively long to minimise the risk of packets being dropped due to congestion.

3.2.2.2 Time references

A slow (nominally 32kHz) global reference clock is distributed throughout the system and is available to each processor via its DMA controller (which performs clock edge detection) and vectored interrupt controller. Software may use this to generate the local time phase information. Each processor also has a timer/counter driven from the local processor clock which can be used to support time reference signals, for example a 1ms interrupt could be used to generate the time input to the real-time neural models.

3.2.2.3 System-level address spaces

The system incorporates different levels of component that must be enumerated:

- Each Node (where a Node is a SpiNNaker chip plus SDRAM) must have a unique, fixed address which is used as the destination ID for a point-to-point packet, and the addresses must be organised logically for algorithmic routing to function efficiently.
- Processors will be addressed relative to their host Node address, but this mapping will not be fixed as an individual Processor's role can change over time. Point-to-point packets addressed to a Node will be delivered to the local Monitor Processor, whichever Processor is serving that function. Internal to a Node there is hard-wired addressing of each Processor for system diagnosis purposes, but this mapping will generally be hidden outside the Node.
- The neuron address space is purely a software issue and is discussed in 'Application notes' on page 95.

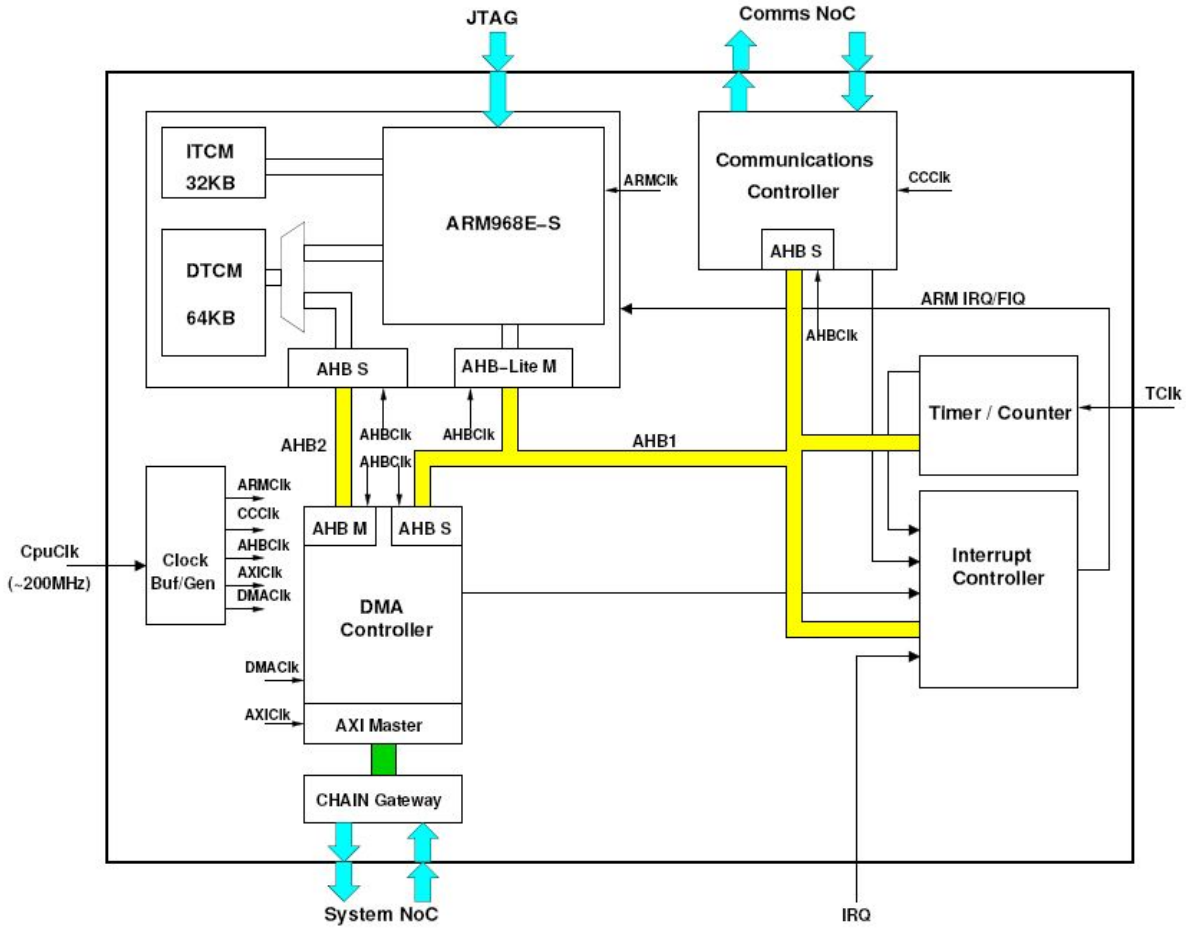
3.2.3 ARM968 processing subsystem

SpiNNaker incorporates 18 ARM968 processing subsystems which provide the computational capability of the device. Each of these subsystems is capable of generating and processing neural events communicated via the Communications NoC and, alternatively, of fulfilling the role of Monitor Processor.

3.2.3.1 Features

- a synthesized ARM968 module with:
- an ARM9TDMI processor;
- 32 Kbyte tightly-coupled instruction memory;
- 64 Kbyte tightly-coupled data memory;
- JTAG debug access.
- a local AHB with:
- communications controller connected to Communications NoC;
- DMA controller and interface to the System NoC;
- timer/counter and interrupt controller.

3.2.3.2 ARM968 subsystem organisation



3.2.3.3 Memory Map

The memory map of the ARM968 spans a number of devices and buses. The tightly-coupled memories are directly connected to the processor and accessible at the processor clock speed. All other parts of the memory map are visible via the AHB master interface, which runs at the full processor clock rate. This gives direct access to the registers of the DMA controller, communications controller and the timer/interrupt controller. In addition, a path is available through the DMA controller onto the System NoC which provides processor access to all memory resources on the System NoC. The memory map is defined as follows:

```
// ARM968 local memories
#define ITCM_START_ADDRESS          0x00000000  // instruction memory
#define DTCM_START_ADDRESS          0x00400000  // data memory

// Local peripherals - unbuffered write
#define COMM_CTL_START_ADDRESS_U    0x10000000  // Communications Controller
#define CTR_TIM_START_ADDRESS_U     0x11000000  // Counter-Timer
#define VIC_START_ADDRESS_U         0x1f000000  // vectored interrupt controller

// Local peripherals - buffered write
#define COMM_CTL_START_ADDRESS_B    0x20000000  // Communications Controller
#define CTR_TIM_START_ADDRESS_B     0x21000000  // Counter-Timer
#define VIC_START_ADDRESS_B         0x2f000000  // vectored interrupt controller

// DMA controller
#define DMA_CTL_START_ADDRESS_U     0x30000000  // DMA controller - unbuffered
#define DMA_CTL_START_ADDRESS_B     0x40000000  // DMA controller - buffered

// Unallocated; causes bus error 0x50000000 - 0x5fffffff

// SDRAM
#define SDRAM_START_ADDRESS_U       0x60000000  // SDRAM - buffered
#define SDRAM_START_ADDRESS_B       0x70000000  // SDRAM - unbuffered

// Unallocated; causes bus error 0x80000000 - 0xdfffffff

// System NoC peripherals - buffered write
#define PL340_APB_START_ADDRESS_B   0xe0000000  // PL340 APB port
#define RTR_CONFIG_START_ADDRESS_B  0xe1000000  // Router configuration
#define SYS_CTL_START_ADDRESS_B     0xe2000000  // System Controller
#define WATCHDOG_START_ADDRESS_B    0xe3000000  // Watchdog Timer
#define ETH_CTL_START_ADDRESS_B     0xe4000000  // Ethernet Controller
#define SYS_RAM_START_ADDRESS_B     0xe5000000  // System RAM
#define SYS_ROM_START_ADDRESS_B     0xe6000000  // System ROM

// Unallocated; causes bus error 0xe7000000 - 0xefffffff

// System NoC peripherals - unbuffered write
#define PL340_ APB_START_ADDRESS_U  0xf0000000  // PL340 APB port
#define RTR_CONFIG_START_ADDRESS_U  0xf1000000  // Router configuration
#define SYS_CTL_START_ADDRESS_U     0xf2000000  // System Controller
#define WATCHDOG_START_ADDRESS_U    0xf3000000  // Watchdog Timer
#define ETH_CTL_START_ADDRESS_U     0xf4000000  // Ethernet Controller
#define SYS_RAM_START_ADDRESS_U     0xf5000000  // System RAM
#define SYS_ROM_START_ADDRESS_U     0xf6000000  // System ROM

// Unallocated; causes bus error 0xf7000000 - 0xfeffffff

// Boot area and VIC
#define BOOT_START_ADDRESS          0xff000000  // Boot area

#define HI_VECTORS                  0xffff0000  // high vectors (for boot)

#define VIC_START_ADDRESS_H         0xfffff000  // vectored interrupt controller
```

The areas shown against a yellow background are accessible only by their local ARM968 processor, not by a DMA controller nor by Nearest Neighbour packets via the Router (though of course the DMA controller can see the ITCM and DTCM areas through its second port, as these are the source/destination for DMA transfers). The DMA controller and Nearest Neighbour packets see the System RAM repeated across the bottom 16Mbytes of the address space from 0x00000000 to 0x00ffffff; the remainder of the yellow areas give undefined results and should not be addressed.

The ARM968 is configured to use high vectors after reset (to use the vectors in the Boot area), but then switched to low vectors once the ITCM is enabled and initialised.

The vectored interrupt controller (VIC) has to be at 0xfffff000 to enable efficient access to its vector registers.

All other peripherals start at a base address that can be formed with a single MOV immediate instruction.

3.2.4 ARM 968

The ARM968 (with its associated tightly-coupled instruction and data memories) forms the core processing resource in SpiNNaker.

3.2.4.1 Features

- ARM9TDMI processor supporting the ARMv5TE architecture.
- 32 Kbyte tightly-coupled instruction memory (I-RAM).
- 64 Kbyte tightly-coupled data memory (D-RAM).
- AHB interface to external system.
- JTAG-controlled debug access.
- support for Thumb and signal processing instructions.
- low-power halt and wait for interrupt function.

3.2.4.2 Organization

See ARM DDI 0311C – the ARM968E-S datasheet.

3.2.4.3 Fault-tolerance

Fault insertion

- ARM9TDMI can be disabled.
- Software can corrupt I-RAM and D-RAM to model soft errors. Fault detection
- A chip-wide watchdog timer catches runaway software.
- Self-test routines, run at start-up and during normal operation, can detect faults. Fault isolation
- The ARM968 unit can be disabled from the System Controller.
- Defective locations in the I-RAM and D-RAM can be mapped out of use by software. Reconfiguration
- Software will avoid using defective I-RAM and D-RAM locations.
- Functionality will migrate to an alternative Processor in the case of permanent faults that go beyond the failure of one or two memory locations.

3.2.5 Vectored interrupt controller

Each processor node on an SpiNNaker chip has a vectored interrupt controller (VIC) that is used to enable and disable interrupts from various sources, and to wake the processor from sleep mode when required. The interrupt controller provides centralised management of IRQ and FIQ sources, and offers an efficient indication of the active sources for IRQ vectoring purposes.

The VIC is the ARM PL190, described in ARM DDI 0181E.

3.2.5.1 Features

- manages the various interrupt sources to each local processor.
- individual interrupt enables.
- routing to FIQ and/or IRQ,
- there will normally be only one FIQ source: e.g. CC Rx ready, or a specific packet-type received.
- a central interrupt status view.
- a vector to the respective IRQ handler.
- programmable IRQ priority.
- interrupt sources:
- Communication Controller flow-control interrupts;
- DMA complete/error/timeout;
- Timer 1 and 2 interrupts;
- interrupt from another processor on the chip (usually the Monitor processor), set via a register in the System Controller;
- packet-error interrupt from the Router;
- system fault interrupt;
- Ethernet controller;
- off-chip signals;
- 32kHz slow system clock;
- software interrupt, for downgrading FIQ to IRQ.

3.2.5.2 Register summary

Base address: 0x2f000000 (buffered write), 0x1f000000 (unbuffered write), 0xfffff000 (high).

User registers

The following registers allow normal user programming of the VIC:

Name	Offset	R/W	Function
r0: VICirqStatus	0x00	R	IRQ status register
r1: VICfiqStatus	0x04	R	FIQ status register
r2: VICrawInt	0x08	R	raw interrupt status register
r3: VICintSel	0x0C	R/W	interrupt select register
r4: VICintEnable	0x10	R/W	interrupt enable register
r5: VICintEnClear	0x14	W	interrupt enable clear register
r6: VICsoftInt	0x18	R/W	soft interrupt register
r7: VICsoftIntClear	0x1C	W	soft interrupt clear register
r8: VICprotection	0x20	R/W	protection register
r9: VICvectAddr	0x30	R/W	vector address register
r10: VICdefVectAddr	0x34	R/W	default vector address register
VICvectAddr[15:0]	0x100-13c	R/W	vector address registers
VICvectCtrl[15:0]	0x200-23c	R/W	vector control registers

ID registers

In addition, there are test ID registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
VICPeriphID0-3	0xFE0-C	R	Timer peripheral ID byte registers
VICPCID0-3	0xFF0-C	R	Timer Prime Cell ID byte registers

See the VIC Technical Reference Manual ARM DDI 0181E, for further details of the ID registers.

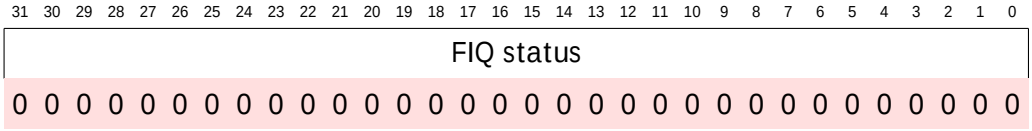
3.2.5.3 Register details

r0: IRQ status

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ status																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

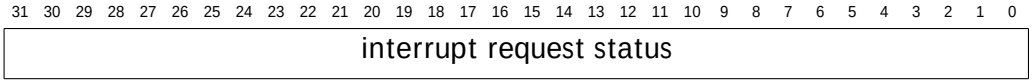
This read-only register yields the set of active IRQ requests (after masking).

r1: FIQ status



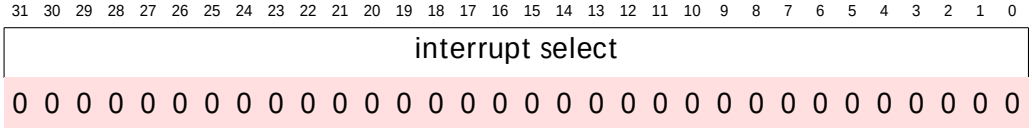
This read-only register yields the set of active FIQ requests (after masking).

r2: raw interrupt status



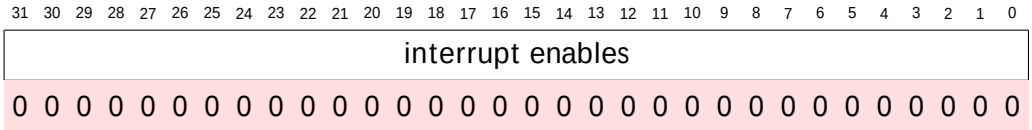
This read-only register yields the set of active input interrupt requests (before any masking).

r3: interrupt select



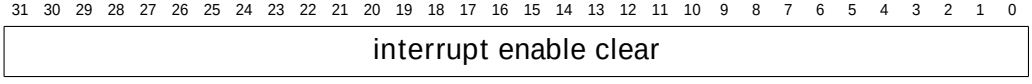
This register selects for each of the 32 interrupt inputs whether it gets sent to IRQ (0) or FIQ (1). The reset state is not specified (though is probably '0'?); all interrupts are disabled by r4 at reset.

r4: interrupt enable register



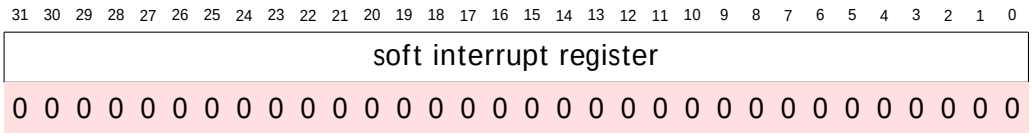
This register disables (0) or enables (1) each of the 32 interrupt inputs. Writing a '1' sets the corresponding bit in r4; writing a '0' has no effect. Interrupts are all disabled at reset.

r5: interrupt enable clear



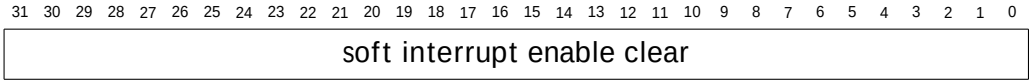
This write-only register selectively clears interrupt enable bits in r4. A '1' clears the corresponding bit in r4; a '0' has no effect.

r6: soft interrupt register



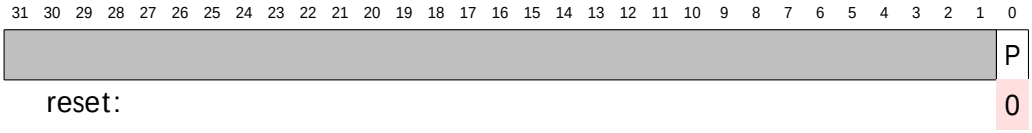
This register enables software to force interrupt inputs to appear high (before masking). A '1' written to any bit location will force the corresponding interrupt input to be active; writing a '0' has no effect. The reset state for these bits is unspecified, though probably '0'?

r7: soft interrupt register clear



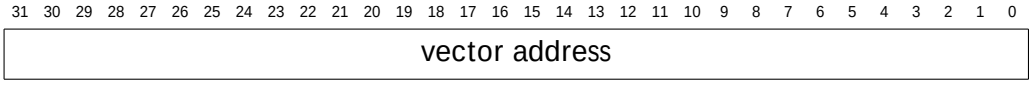
This write-only register selectively clears soft interrupt bits in r6. A '1' clears the corresponding bit in r6; a '0' has no effect.

r8: protection



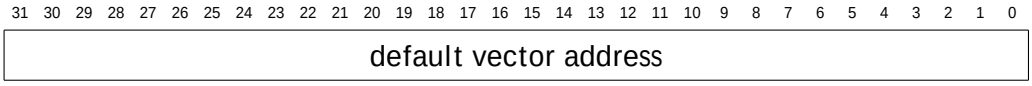
If the P bit is set VIC registers can only be accessed in a privileged mode; if it is clear then User- mode code can access the registers.

r9: vector address



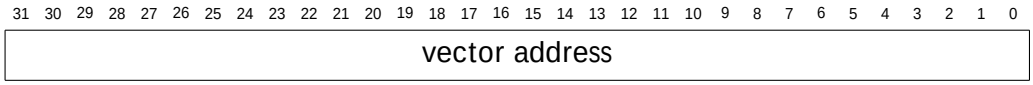
This register contains the address of the currently active interrupt service routine (ISR). It must be read at the start of the ISR, and written at the end of the ISR to signal that the priority logic should update to the next priority interrupt. Its state following reset is undefined.

r10: default vector address



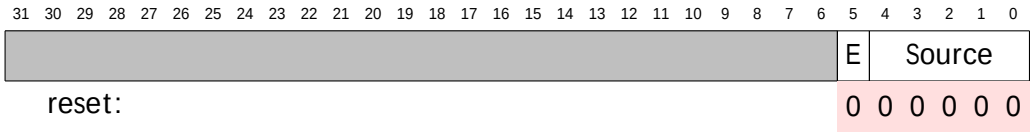
The default vector address is used by the 16 interrupts that are not vectored. Its state following reset is undefined.

vector address [15:0]



The vector address is the address of the ISR of the selected interrupt source. Their state following reset is undefined.

vector control [15:0]



The interrupt source is selected by bits[4:0], which choose one of the 32 interrupt inputs. The interrupt can be enabled (E = 1) or disabled (E = 0). It is disabled following reset. The highest priority interrupt uses vector address [0] at offset 0x100 and vector control [0] at offset 0x200, and then successively reduced priority is given to vector addresses [1], [2], . . . and vector controls [1], [2], . . . at successively higher offset addresses.

3.2.5.4 Interrupt sources

19 of the 32 interrupt sources are local to the processor (and are coloured yellow in the table below) and 13 are from chip-wide sources (which will normally be enabled only in the Monitor Processor).

#	Name	Function
0	Watchdog	Watchdog timer interrupt
1	Software int	used only for local software interrupt generation
2	Comms Rx	the debug communications receiver interrupt
3	Comms Tx	the debug communications transmitter interrupt
4	Timer 1	Local counter/timer interrupt 1
5	Timer 2	Local counter/timer interrupt 2
6	CC Rx ready	Local comms controller packet received
7	CC Rx parity error	Local comms controller received packet parity error
8	CC Rx framing error	Local comms controller received packet framing error
9	CC Tx full	Local comms controller transmit buffer full
10	CC Tx overflow	Local comms controller transmit buffer overflow
11	CC Tx empty	Local comms controller transmit buffer empty
12	DMA done	Local DMA controller transfer complete
13	DMA error	Local DMA controller error
14	DMA timeout	Local DMA controller transfer timed out
15	Router diagnostics	Router diagnostic counter event has occurred
16	Router dump	Router packet dumped - indicates failed delivery
17	Router error	Router error - packet parity, framing, or time stamp error
18	Sys Ctl int	System Controller interrupt bit set for this processor
19	Ethernet Tx	Ethernet transmit frame interrupt
20	Ethernet Rx	Ethernet receive frame interrupt
21	Ethernet PHY	Ethernet PHY/external interrupt
22	Slow Timer	System-wide slow (nominally 32 KHz) timer interrupt
23	CC Tx not full	Local comms controller can accept new Tx packet
24	CC MC Rx int	Local comms controller multicast packet received
25	CC P2P Rx int	Local comms controller point-to-point packet received
26	CC NN Rx int	Local comms controller nearest neighbour packet received
27	CC FR Rx int	Local comms controller fixed route packet received
28	Int[0]	External interrupt request 0
29	Int[1]	External interrupt request 1
30	GPIO[8]	Signal on GPIO[8]
31	GPIO[9]	Signal on GPIO[9]

3.2.5.5 Fault-tolerance

Fault insertion

It is fairly easy to mess up vector locations, and to fake interrupt sources.

Fault detection

A failed vector location effectively causes a jump to a random location; this would be messy!

Fault isolation

Failed vector locations can be removed from service.

Reconfiguration

A failed vector location can be removed from service (provided there are enough vector locations available without it). Alternatively, the entire vector system could be shut down and interrupts run by software inspection of the IRQ and FIQ status registers.

3.2.6 Counter/timer

Each processor node on a SpiNNaker chip has a counter/timer.

The counter/timers use the standard AMBA peripheral device described on page 4-24 of the AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003. The peripheral has been modified only in that the APB interface of the original has been replaced by an AHB interface for direct connection to the ARM968 AHB bus.

3.2.6.1 Features

- the counter/timer unit provides two independent counters, for example for:
  - millisecond interrupts for real-time dynamics.
- free-running and periodic counting modes:
  - automatic reload for precise periodic timing;
  - one-shot and wrapping count modes.
- the counter clock (which runs at the processor clock frequency) may be pre-scaled by dividing by 1, 16 or 256.

3.2.6.2 Register summary

Base address: 0x21000000 (buffered write), 0x11000000 (unbuffered write).

User registers

The following registers allow normal user programming of the counter/timers:

Name	Offset	R/W	Function
r0: Timer1load	0x00	R/W	Load value for Timer 1
r1: Timer1value	0x04	R	Current value of Timer 1
r2: Timer1Ctl	0x08	R/W	Timer 1 control
r3: Timer1IntClr	0x0C	W	Timer 1 interrupt clear
r4: Timer1RIS	0x10	R	Timer 1 raw interrupt status
r5: Timer1MIS	0x14	R	Timer 1 masked interrupt status
r6: Timer1BGload	0x18	R/W	Background load value for Timer 1
r8: Timer2load	0x20	R/W	Load value for Timer 2
r9: Timer2value	0x24	R	Current value of Timer 2
r10: Timer2Ctl	0x28	R/W	Timer 2control
r11: Timer2IntClr	0x2C	W	Timer 2interrupt clear
r12: Timer2RIS	0x30	R	Timer 2raw interrupt status
r13: Timer2MIS	0x34	R	Timer 2masked interrupt status
r14: Timer2BGload	0x38	R/W	Background load value for Timer 2

Test and ID registers

In addition, there are test and ID registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
TimerITCR	0xF00	R/W	Timer integration test control register
TimerITOP	0xF04	W	Timer integration test output set register
TimerPeriphID0-3	0xFE0-C	R	Timer peripheral ID byte registers
TimerPCID0-3	0xFF0-C	R	Timer Prime Cell ID byte registers

See AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003, for further details of the test and ID registers.

3.2.6.3 Register details

As both timers have the same register layout they can both be described as follows (X = 1 or 2):

r0/8: Timer X load value

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Load value for TimerX																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

When written, the 32-bit value is loaded immediately into the counter, which then counts down from the loaded value. The background load value (r6/14) is an alternative view of this register which is loaded into the counter only when the counter next reaches zero.

r1/9: Current value of Timer X

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TimerX current count																															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This read-only register yields the current count value for Timer X.

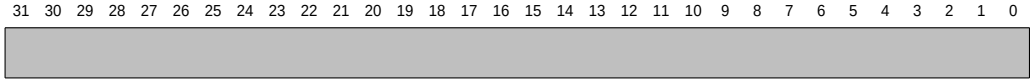
r2/10: Timer X control

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																								E	M	I		Pre		S	O
reset:																								0	0	1		0	0	0	0

The shaded fields should be written as zero and are undefined on read. The functions of the remaining fields are described in the table below:

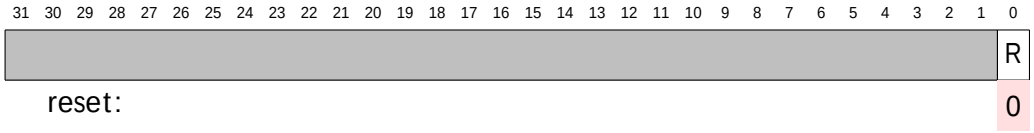
Name	bits	R/W	Function
E: Enable	7	R/W	enable counter/timer (1 = enabled)
M: Mode	6	R/W	0 = free-running; 1 = periodic
I: Int enable	5	R/W	enable interrupt (1 = enabled)
Pre: TimerPre	3:2	R/W	divide input clock by 1 (00), 16 (01), 256 (10)
S: Timer size	1	R/W	0 = 16 bit, 1 = 32 bit
O: One shot	0	R/W	0 = wrapping mode; 1 = one shot

r3/11: Timer X interrupt clear



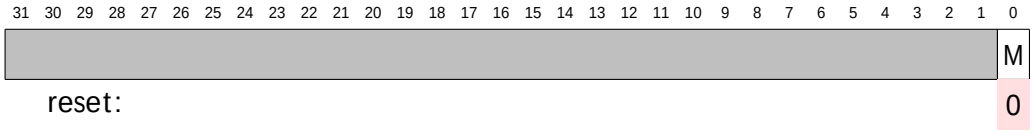
Any write to this address will clear the interrupt request.

rr4/12: Timer X raw interrupt status



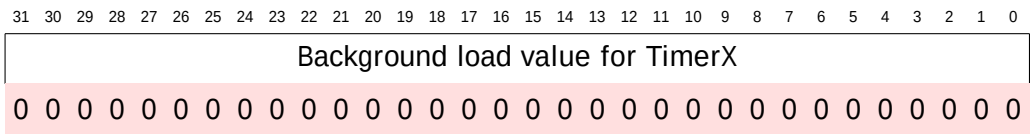
Bit zero yields the raw (unmasked) interrupt request status of this counter/timer.

r5/13: Timer X masked interrupt status



Bit zero yields the masked interrupt status of this counter/timer.

r6/14: Timer X background load value



The 32-bit value written to this register will be loaded into the counter when it next counts down to zero. Reading this register will yield the same value as reading register 0/8.

3.2.6.4 Fault-tolerance

Fault insertion

Disabling a counter (by clearing the E bit in its control register) will cause it to fail in its function.

Fault detection

Use the second counter/timer with a longer period to check the calibration of the first?

Fault isolation

Disable the counter/timer with the E bit in the control register; disable its interrupt output; disable the interrupt in the interrupt controller.

Reconfiguration

If one counter fails then a system that requires only one counter can use the other one.

3.2.7 DMA controller

Each ARM968 processing subsystem includes a DMA controller. The DMA controller is primarily used for transferring inter-neural connection data from the SDRAM in large blocks in response to an input event arriving at a fascicle processor, and for returning updated connection data during learning. In addition, the DMA controller can transfer data to/from other targets on the System NoC such as the System RAM and Boot ROM.

As a secondary function the DMA controller incorporates a 'Bridge' across which its host ARM968 has direct read and write access to System NoC devices, including the SDRAM. The ARM968 can use the Bridge whether or not DMA transfers are active.

3.2.7.1 Features

- DMA engine supporting parallel operations:
  - DMA transfers;
  - direct pass-through requests from the ARM968;
  - dual buffers supporting simultaneous direct and DMA transfers.
- Support for CRC error control in transferred blocks.
- Interrupt-driven or polled DMA completion notification:
  - DMA complete interrupt signal;
  - various DMA error interrupt signals;
  - DMA time-out interrupt signal.
- Parameterisable buffer sizes.
- Direct and DMA request queueing.

3.2.7.2 Using the DMA controller

There are 2 types of requests for DMA controller services. DMA transfers are initiated by writing to control registers in the controller, executed in the background, and signal an interrupt when complete. Bridge transfers occur when the ARM initiates a request directly to the needed device or service. The DMA controller fulfills these requests transparently, the host processor retaining full control of the transfer. Invisible to the user, the controller may buffer the data from write requests for more efficient bus management. If an error occurs on such a buffered write the DMA controller can signal an error interrupt.

The controller acts as a Bridge between the AHB bus on the ARM AHB slave interface and the AXI interface on the system NoC, performing the required address and control resequencing (stripping addresses from non-first beats of a burst), data flow management and request arbitration. The arbiter prioritises requests in the following order:

1) Bridge reads,
2) Bridge writes,

3) DMA burst requests.

No request can gain access to the AXI interface until any active burst transaction on the interface has completed. Read requests while a DMA transfer is in progress require special handling. The read must wait until any active request has completed, and therefore a Bridge read could stall the processor and AHB slave bus for many cycles. In addition, if buffered writes exist, potential data coherency conflicts exist. The recommended procedure is for the ARM processor to interrogate the WB active (A) bit in the DMA Status register (STAT) before requesting a Bridge read.

To initiate a DMA transfer, the ARM must write to the following registers in the DMA controller: System Address (ADRS), TCM Address (ADRT), and Description (DESC). The order of writing of the first two register operations is not important, but the Description write must be the last as it commits the DMA transfer. The processor may also optionally write the CRC and Global Control (GCTL) registers to set up additional parameters. The expected model, however, is that these registers are updated infrequently, perhaps only once after power-up. The processor may read from any register at any time. The processor may have a maximum of 2 submitted DMA requests of which only one will be active. When the transfer queue is empty (as indicated by the Q bit in the Status (STAT) register), the processor may queue another request.

Accesses to DMA Controller registers are restricted to programs running on the ARM968 in privileged (i.e. non-user) modes. Attempts to access these registers in user mode will result in a bus error.

An attempt to write register r1 to r3 when the queue is full will result in a bus error.

Any access (read or write) to a non-existent register will result in a bus error.

Non-word-aligned addresses and byte and half-word accesses will result in a bus error.

3.2.7.3 Register summary

Base address: 0x40000000 (buffered write), 0x30000000 (unbuffered write).

Name	bits	R/W	Function
r0: unused	0x00		
r1: ADRS	0x04	R/W	DMA address on the system interface
r2: ADRT	0x08	R/W	DMA address on the TCM interface
r3: DESC	0x0C	R/W	DMA transfer description
r4: CTRL	0x10	R/W	Control DMA transfer
r5: STAT	0x14	R	Status of DMA and other transfers
r6: GCTL	0x18	R/W	Control of the DMA device
r7: CRCC	0x1C	R	CRC value calculated by CRC block
r8: CRCR	0x20	R	CRC value in received block
r9: TMTV	0x24	R/W	Timeout value
r10: StatsCtl	0x28	R/W	Statistics counters control
r16-23: Stats0-7	0x40-5C	R	Statistics counters
r64: unused	0x100		
r65: AD2S	0x104	R	Active system address
r66: AD2T	0x108	R	Active TCM address
r67: DES2	0x10C	R	Active transfer description
r96-r127	0x180-1FC	R/W	CRC polynomial matrix

3.2.7.4 Register details

r0: unused

r1: ADRS - System Address.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
System Address																														00	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

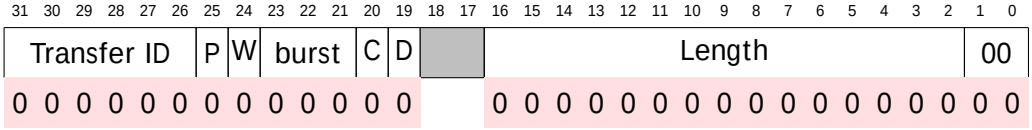
The 32-bit start byte address on the system interface. Note that a read is considered a data movement from a source on the system bus to a destination on the TCM bus. DMA transfers are word-aligned, so bits[1:0] are fixed at zero.

r2: ADRT - TCM Address.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TCM Address																														00	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The 32-bit start address on the TCM interface.

r3: DESC - DMA transfer description.



The function of these fields is described in the table below:

Name	bits	R/W	Function
Transfer ID	31:26	R/W	software defined transfer ID
P: Privilege	25	R/W	DMA transfer mode is user (0) or privileged (1)
W: Width	24	R/W	transfer width is word (0) or double-word (1)
Burst	23:21	R/W	burst length = $2^B \times$ Width, $B = 0 \ldots 4$ (i.e max 16)
C: CRC	20	R/W	check (read) or generate (write) CRC
D: Direction	19	R/W	read from (0) or write to (1) system bus
Length	16:2	R/W	length of the DMA transfer, in words

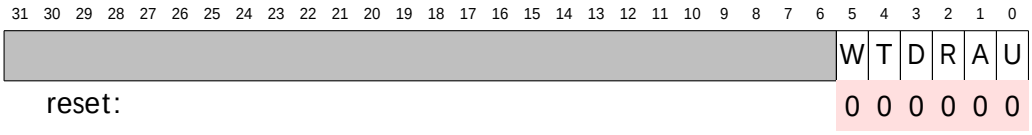
The TCM as currently implemented has a size of 64Kbytes (for the data TCM). A DMA transfer must of necessity either take as a source or a destination the TCM, justifying this restriction. DMA transfers are word-aligned, so bits[1:0] are fixed at zero.

The Burst length defines the unit of transfer (in words or double-words, depending on W) across the System NoC. Longer bursts will in general make more efficient use of the available SDRAM bandwidth.

Note that the Length excludes the 32-bit CRC word, if CRC is used.

Writing to this register automatically commits a transfer as defined by the values in r1-r3.

r4: CTRL - Control Register

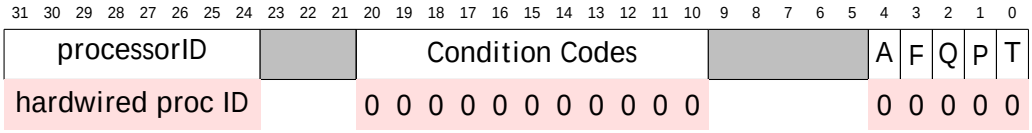


The functions of these fields are described in the table below:

Name	bits	R/W	Function
W: clear WB Int	5	R/W	clear Write Buffer interrupt request
T: clear Timeout Int	4	R/W	clear Timeout interrupt request
D: clear Done Int	3	R/W	clear Done interrupt request
R: Restart	2	R/W	resume transfer (clears DMA errors)
A: Abort	1	R/W	end current transfer and discard data
U: Uncommit	0	R/W	setting this bit uncommits a queued transfer

These bits can only be set to 1 by the user, they cannot be reset. Writing a 0 has no effect. They will clear automatically once they have taken effect, which will be at the next safe opportunity, typically between transfer bursts.

r5: STAT - Status Register.



The functions of these fields are described in the table below:

Name	bits	R/W	Function
processor ID	31:24	R	hardwired processor ID identifies CPU on chip
Condition Codes	20:10	R	DMA condition codes
A: WB active	4	R	write buffer is not empty
F: WB full	3	R	write buffer is full
Q: Queue full	2	R	DMA transfer is queued - registers are full
P: Paused	1	R	DMA transfer is PAUSED
T: Transferring	0	R	DMA transfer in progress

The condition codes are defined as follows:

Name	bits	R/W	Function
Write buff error	20	R	a buffered write transfer has failed
TBD	19:18	R	not yet allocated
Soft reset	17	R	a soft reset of the DMA controller has happened
User abort	16	R	the user has aborted the transfer (via r4)
AXI error	15	R	the AXI interface has signalled a transfer error
TCM error	14	R	the TCM AHB interface has signalled an error
CRC error	13	R	the calculated and received CRCs differ
Timeout	12	R	a burst transfer has not completed in time
2nd transfer done	11	R	2nd DMA transfer has completed without error
Transfer done	10	R	a DMA transfer has completed without error

When a DMA error occurs the corresponding condition code flag is set, the DMA engine is PAUSED (bit[1]) and the current transfer is terminated. A queued transfer remains in the queue but is not started. A new transfer can be committed if the queue is empty, but it will not start until the DMA controller is brought out of PAUSE. AD2S, AD2T and DES2 (r65-67) contain information about the failed transfer and can be used to diagnose the problem. A restart command (r4 bit[2]) is required to bring the DMA controller out of PAUSE. This will

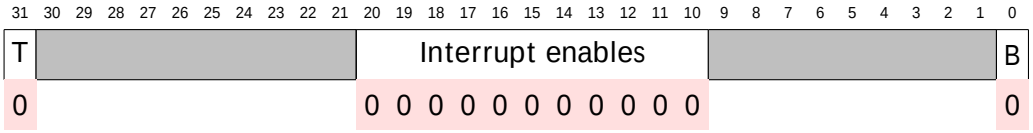
clear the error codes [16:13] and restart DMA operation. The terminated transfer must be restarted explicitly by software if this is required.

A soft reset will set bit[17], clear the transfer queue and take the DMA controller into the IDLE state. The DMA controller is not PAUSED, and new transfers can be committed and start immediately. A restart command (r4 bit[2]) is required to clear the soft reset flag [17] - starting a new transfer does NOT clear it.

Timeout [12] and Write Buffer error [20] have explicit clears in CTRL.

The two transfer done bits [11:10] count up through the sequence 00 → 01 → 11 as DMA transfers complete, and count down through the reverse sequence when a 1 is written to CTRL[3]. As a result of this coding, Transfer Done [10] can be read as indicating that at least one DMA transfer has completed, and a second completed transfer can be handled by inspecting bit[11] in software or left to be handled by a subsequent Transfer Done interrupt.

r6: GCTL - Global Control



The functions of these fields are described in the table below:

Name	bits	R/W	Function
T: Timer	31	R/W	system-wide slow timer status and clear
Interrupt enables	20:10	R/W	respective interrupt enables for the r5 conditions
B: Bridge buffer	0	R/W	enable Bridge write buffer

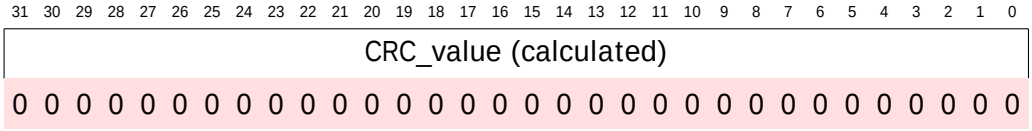
The DMA controller passes four interrupt request lines to the VIC:

- dmac_done: the logical OR of GCTL[11:10] and STAT[11:10]
- dmac_timeout: GCTL[12] and STAT[12]
- dmac_error: the logical OR of GCTL[20:13] and STAT[20:13]
- system-wide slow (nominally 32 KHz) timer interrupt

Note that write buffer errors and timeout errors do NOT stop the DMA engine nor the transfer in progress.

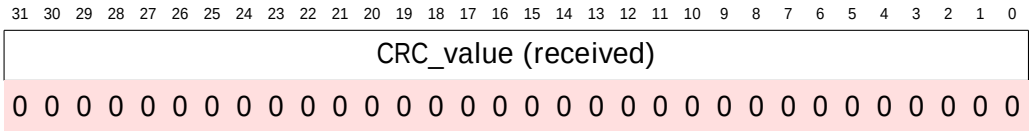
The system-wide slow timer is a clock signal that sets bit[31] on every rising edge, thereby raising an interrupt request to the VIC, and is cleared by writing a 0 to bit[31]. Writing a 1 to bit[31] has no effect.

r7: CRCC - Calculated CRC



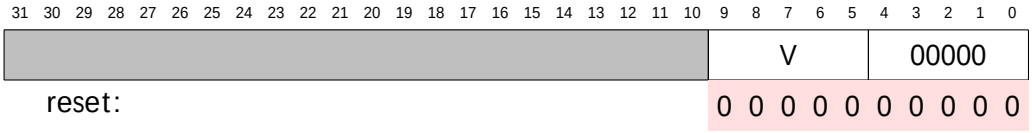
This is the 32-bit CRC value calculated by the DMA CRC unit.

r8: CRCC - Received CRC



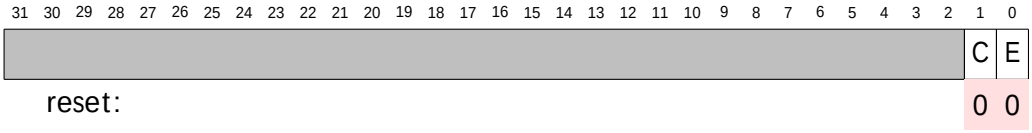
This is the 32-bit CRC value read in the block of data loaded by a DMA transfer.

r9: TMTV - Timeout value



This is a 10-bit counter value used to determine when the DMA controller should timeout on an attempted transfer burst. The count units are clock cycles. When TMTV = 0 the timeout counter is disabled. Note that a timeout will not stop the transfer.

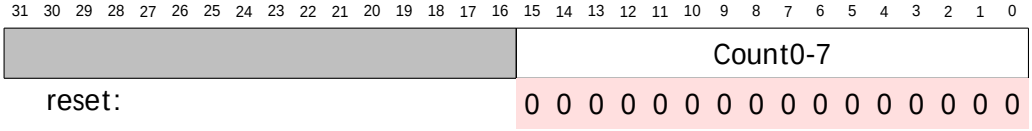
r10: StatsCtl - Statistics counters control



E, bit[0], enables the statistics counters (r16-23).

Writing '1' to C, bit[1], zeroes the statistics counters. Writing a '0' has no effect. Bit[1] always reads '0'.

r16-23: Stats0-7 - Statistics counters



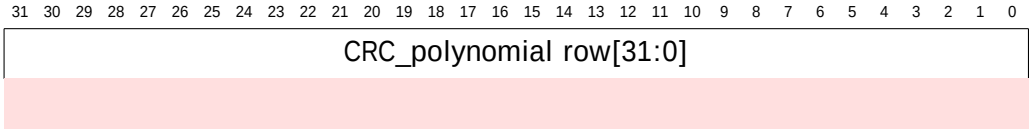
These eight 16-bit counter registers record statistics relating to the latency of DMA transations across the System NoC. Count0 records the number of transactions that complete in 0-127 clock cycles, Count1 128-255 clock cycles, and so on up to Count7 which counts transactions that complete in 896+ clock cycles.

The counters are enabled and cleared via r10.

r65-67: Active DMA transfer registers

These registers are not directly written. They reflect the state of the active DMA transfer, with AD2S and AD2T holding the respective System and TCM addresses to be used in the next burst of the transfer, and DES2 holding the description of the transfer in progress (the remaining length, ID, burst size, and direction).

r96-127: CRC polynomial matrix



The CRC hardware is highly programmable and can be used in a number of ways to detect, and possibly correct, errors in blocks of data transferred by the DMA controller between the ARM968 DTCM and the off-chip SDRAM.

For example, to use the Ethernet 32-bit CRC with polynomial 0x04C11DB7, the following 32 hexadecimal values should be programmed into r96-127:

```
FB808B20, 7DC04590, BEE022C8, 5F701164, 2FB808B2, 97DC0459, B06E890C, 58374486,
AC1BA243, AD8D5A01, AD462620, 56A31310, 2B518988, 95A8C4C4, CAD46262, 656A3131,
493593B8, 249AC9DC, 924D64EE, C926B277, 9F13D21B, B409622D, 21843A36, 90C21D1B,
33E185AD, 627049F6, 313824FB, E31C995D, 8A0EC78E, C50763C7, 19033AC3, F7011641.
```

The CRC unit is configurable to use a different 32-bit polynomial, a different polynomial length, and a different data word length. For example, it can be configured to compute CRC16 separately for each half-word of the data stream. A Matlab program can be used to determine the appropriate polynomial matrix values.

3.2.7.5 Fault-tolerance

Fault insertion

Software can introduce errors in data blocks in SDRAM which should be trapped by the CRC hardware.

Fault detection

The CRC unit can detect errors in the data transferred by the DMA controller. The DMA controller will time-out if a transaction takes too long.

Fault isolation

The DMA Controller is mission-critical to the local processing subsystem, so if it fails the subsystem should be disabled and isolated.

Reconfiguration

The local processing subsystem is shut down and its functions migrated to another subsystem on this or another chip. It should be possible to recover all of the subsystem state and to migrate it, via the SDRAM, to a functional alternative.

3.2.8 Communications controller

Each processor node on SpiNNaker includes a communications controller which is responsible for generating and receiving packets to and from the communications network.

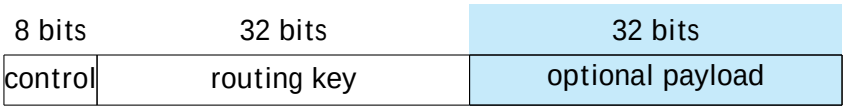
3.2.8.1 Features

- Support for 4 packet types:
  - multicast (MC) neural event packets routed by a key provided at the source;
  - point-to-point (P2P) packets routed by destination address;
  - nearest-neighbour (NN) packets routed by arrival port;
  - fixed-route (FR) packets routed by the contents of a register.
- Packets are either 40 or 72 bits long. The longer packets carry a 32-bit payload.
- 2-bit time stamp (used by Routers to trap errant packets).
- Parity (to detect some corrupt packets).

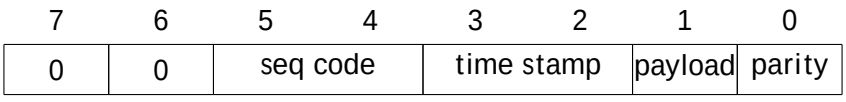
3.2.8.2 Packet formats

Neural event multicast (MC) packets (type 0)

Neural event packets include a control byte and a 32-bit routing key inserted by the source. In addition they may include an optional 32-bit payload:



The 8-bit control field includes packet type (bits[7:6] = 00 for multicast packets), emergency routing and time stamp information, a payload indicator, and error detection (parity) information:



Point-to-point (P2P) packets (type 1)

Point-to-point packets include 16-bit source and destination chip IDs, plus a control byte and an optional 32-bit payload:

8 bits	16 bits	16 bits	32 bits
control	source ID	destination ID	optional payload

Here the 8-bit control field includes packet type (bits[7 : 6] = 01 for P2P packets), a sequence code, time stamp, a payload indicator and error detection (parity) information:

7	6	5	4	3	2	1	0
0	1	seq code		time stamp		payload	parity

Nearest-neighbour (NN) packets (type 2)

Nearest-neighbour packets include a 32-bit address or operation field, plus a control byte and an optional 32-bit payload:

8 bits	32 bits	32 bits
control	address/operation	optional payload

Here the 8-bit control field includes packet type (bits[7 : 6] = 10 for NN packets), a 'peek/poke' or 'normal' type indicator (T), routing information, a payload indicator and error detection (parity) information:

7	6	5	4	3	2	1	0
1	0	T	route			payload	parity

Fixed-Route (FR) packets (type 3)

Fixed-route packets include a 32-bit payload field, plus a control byte and an optional 32-bit payload extension:

8 bits	32 bits	32 bits
control	payload	optional payload extension

Here the 8-bit control field includes packet type (bits[7 : 6] = 11 for FR packets), emergency routing and time stamp information, a payload indicator, and error detection (parity) information:

7	6	5 4	3 2	1	0
1	1	emergency routing	time stamp	payload	parity

3.2.8.3 Control byte summary

The various fields in the control bytes of the different packet types are summarised below:

Field Name	bits	Function
parity	0	parity of complete packet (including payload when used)
payload	1	data payload (1) or no data payload (0)
time stamp	3:2	phase marker indicating time packet was launched
seq code	5:4	P2P only: sequence code, software defined
emergency routing	5:4	MC & FR: used to control routing around a failed link
route	4:2	NN only: information for the Router
T: NN packet type	5	NN only: packet type - normal (0) or peek/poke (1)
packet type	7:6	= 00 for MC; = 01 for P2P; = 10 for NN; = 11 for FR

Parity

The complete packet (including the data payload where used) will have odd parity.

data

Indicates whether the packet has a 32-bit data payload (= 1) or not (= 0).

time stamp

The system has a global time phase that cycles through 00 → 01 → 11 → 10 → 00. Global synchronisation must be accurate to within one time phase (the duration of which is programmable and may be dynamically variable). A packet is launched with a time stamp equal to the current time phase, and if a Router finds a packet that is two time phases old (time now XOR time launched = 11) it will drop it to the local Monitor Processor. The time stamp is inserted by the local Router if the route field in SAR (see 'Register details' on page 33) is 111, which is the normal case, so the Communication Controller need do nothing here. If SAR holds a different value in the route field the time stamp from TCR is used.

seq code

P2P packets may use these bits (under software control) to indicate the sequence of data payloads, or for other purposes.

emergency routing

MC & FR packets use these bits to control emergency routing around a failed or congested link:

- 00 → normal packet;
- 01 → the packet has been redirected by the previous Router through an emergency route along with a normal copy of the packet. The receiving Router should treat this as a combined normal plus emergency packet.
- 10 → the packet has been redirected by the previous Router through an emergency route which would not be used for a normal packet.
- 11 → this emergency packet is reverting to its normal route.

route

These bits are set at packet launch to the values defined in the control register. They enable a packet to be directed to a particular neighbour (0 - 5), broadcast to all or a subset (as defined in the Router r33 'NN broadcast' bits - see 'r33: fixed-route packet routing' on page 49) of neighbours (6), or to the local Monitor Processor (7).

T (NN packet type)

This bit specifies whether an NN packet is 'normal', so that it is delivered to the Monitor Processor on the neighbouring chip(s), or 'peek/poke', so that performs a read or write access to the neighbouring chip's System NoC resource.

packet type

These bits indicate whether the packet is a multicast (00), point-to-point (01), nearest-neighbour (10) or fixed-route (11) packet.

3.2.8.4 Debug access to neighbouring devices

The 'peek' and 'poke' mechanism gives access to the System NoC address space on any neighbouring device without processor intervention on that chip. To read a word, include its address in a 'peek/poke' nearest neighbour packet output (i.e. with the T bit set). Only word addresses are permitted. The absence of a payload indicates that a read ('peek') is required. This would normally be done by a Monitor Processor although, in principle, any processor can output his packet.

The target device performs the appropriate access and returns a response on the corresponding link input. This is delivered to the processor designated as Monitor Processor in the local router. The response is a 'normal' NN packet which carries the requested word as payload. The address field is also returned for identification purposes with the least significant bit set to indicate a response. Bit 1 of the address will also be set if the access caused a bus error. Writing ('poke') is similar; including a payload in the outgoing packet causes that

word to be written. A payload-less response packet is returned which will indicate the error status.

3.2.8.5 Register summary

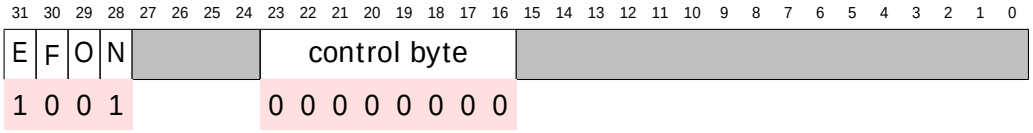
Base address: 0x20000000 (buffered write), 0x10000000 (unbuffered write).

Name	Offset	R/W	Function
r0: TCR (Tx control)	0x00	R/W	Controls packet transmission
r1: TDR (Tx data)	0x04	W	32-bit data for transmission
r2: TKR (Tx key)	0x08	W	Send MC key/P2P dest ID & seq code
r3: RSR (Rx status)	0x0C	R/W	Indicates packet reception status
r4: RDR (Rx data)	0x10	R	32-bit received data
r5: RKR (Rx key)	0x14	R	Received MC key/P2P source ID & seq code
r6: SAR (Source addr)	0x18	R/W	P2P source address
r7: TSTR (test)	0x1C	R/W	Used for test purposes

A packet will contain a data payload if r1 is written before r2; this can be performed using an ARM STM instruction.

3.2.8.6 Register details

r0: TCR - transmit control



The functions of these fields are described in the table below:

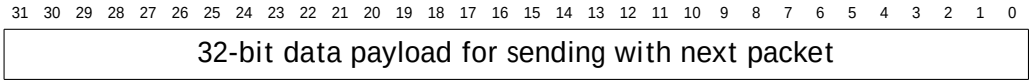
Name	bits	R/W	Function
E: empty	31	R	Tx buffer empty
F: full	30	R/W	Tx buffer full (sticky)
O: overrun	29	R/W	Tx buffer overrun (sticky)
N: not full	28	R	Tx buffer not full, so it is safe to send a packet
control byte	23:16	W	control byte of next sent packet

The parity field in the control byte will be replaced by an automatically-generated value when the packet is launched, and the sequence field will be replaced by the value in TKR. The time stamp (where applicable) will be inserted by the local Router if the route field in SAR is 111, otherwise the value here will be used.

The transmit buffer full and not full controls are expected to be used, by polling or interrupt, to prevent buffer overrun. Tx buffer full is sticky and, once set, will remain set until 0 is written to bit 30. Transmit buffer overrun indicates packet loss and will remain set until explicitly cleared by writing 0 to bit 29.

E, F, O and N reflect the levels on the Tx interrupt signals sent to the interrupt controller.

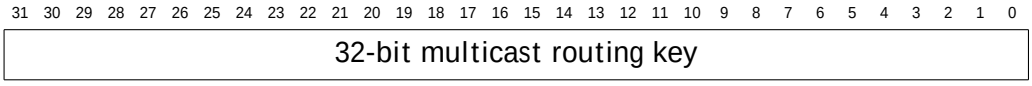
r1: TDR - transmit data payload



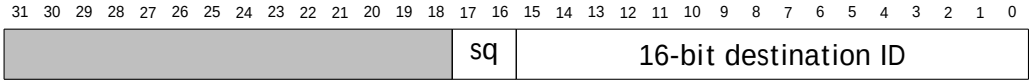
If data is written into TDR before a send key or dest ID is written into TKR, the packet initiated by writing to TKR will include the contents of TDR as its data payload. If no data is written into TDR before a send key or dest ID is written into TKR the packet will carry no data payload.

r2: TKR - send MC key or P2P dest ID & sequence code

Writing to TKR causes a packet to be issued (with a data payload if TDR was written previously). If bits[23:22] of the control register in TCR are 00 the Communication Controller is set to send multicast packets and a 32-bit routing key should be written into TKR. The 32-bit routing key is used by the associative multicast Routers to deliver the packet to the appropriate destination(s).



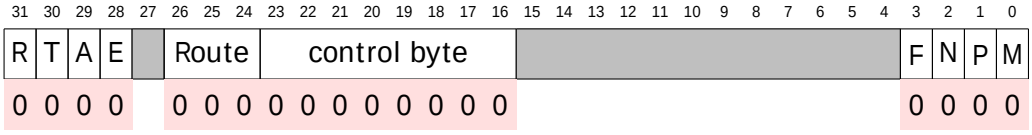
If bits[23:22] of the control register are 01 the Communication Controller is set to send point-to- point packets and the value written into TKR should include the 16-bit address of the destination chip in bits[15:0] and a sequence code in bits[17:16]. (See ‘seq code’ on page 32.)



If bits[23:22] of the control register are 10 the Communication Controller is set to send nearest neighbour packets and the 32-bit NN address/operation field should be written in TKR.

If bits[23:22] of the control register are 11 the Communications Controller is set to send fixed-route packets and the value written into TKR is a 32-bit payload, possibly augmented by a further 32 bits in TDR if this was written previously.

r3: RSR - receive status



The functions of these fields are described in the table below:

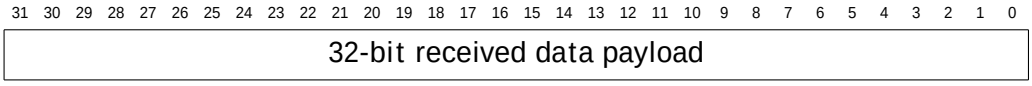
Name	bits	R/W	Function
R: received	31	R	Rx packet received
T: parity	30	R/W	Rx packet parity error (sticky)
A: framing error	29	R/W	Rx packet framing error (sticky)
E: error-free	28	R	Rx packet received without error
Route	26:24	R	Rx route field from packet
Control byte	23:16	R	Control byte of last Rx packet
F: FR packet	3	R	error-free fixed-route packet received
N: NN packet	2	R	error-free nearest-neighbour packet received
P: P2P packet	1	R	error-free point-to-point packet received
M: MC packet	0	R	error-free multicast packet received

Any packet that is received will set R, which will remain set until RKR has been read. A packet that is received with a parity and/or framing error also sets T and/or A. These bits remain set until explicitly reset by writing 0 to bit 30 or bit 29 respectively.

R, T, A, M, P, N & F reflect the levels on the Rx interrupt signals sent to the interrupt controller.

Note that these status bits will have a one-cycle latency before becoming valid so, for example, checking R one cycle after reading RKR will return 1, the old value.

r4: RDR - received data

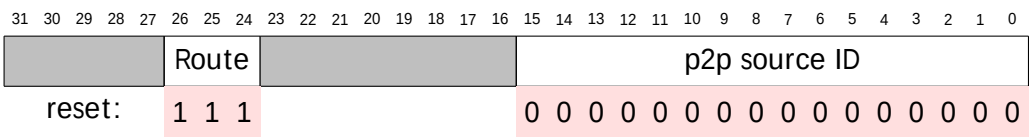


If a received packet carries a data payload the payload will be delivered here and will remain valid until r5 is read.

r5: RKR - received MC key or P2P source ID & sequence code

A received packet will deliver its MC routing key, NN address or P2P source ID and sequence code to RKR. For an MC or NN packet this will be the exact value that the sender placed into its TKR for transmission; for a P2P packet the sequence number will be that placed by the sender into its TKR, and the 16-bit source ID will be that in the sender's SAR. The register is read sensitive - once read it will change as soon as the next packet arrives.

r6: SAR - source address and route



The functions of these fields are described in the table below:

Name	bits	R/W	Function
Route	26:24	W	Set 'fake' route in packet
P2P source ID	15:0	W	16-bit chip source ID for P2P packets

The P2P source ID is expected to be configured once at start-up.

The route field allows a packet to be sent by a processor to the router which appears to have come from one of the external links. Normally this field will be set to 7 (0b111) but can be set to a link number in the range 0 to 5 to achieve this.

r7: TSTR - test

Setting bit 0 of this register makes all registers read/write for test purposes. Clearing bit 0 restricts write access to those register bits marked as read-only in this datasheet. All register bits may be read at any time. Bit 0 is cleared by reset.

3.2.8.7 Fault-tolerance

Fault insertion

Software can cause the Communications Controller to misbehave in several ways including inserting dodgy routing keys, source IDs, destination IDs.

Fault detection

Parity of received packet; received packet framing error; transmit buffer overrun.

Fault isolation

The Communications Controller is mission-critical to the local processing subsystem, so if it fails the subsystem should be disabled and isolated.

Reconfiguration

The local processing subsystem is shut down and its functions migrated to another subsystem on this or another chip. It should be possible to recover all of the subsystem state and to migrate it, via the SDRAM, to a functional alternative.

3.2.9 Communications NoC

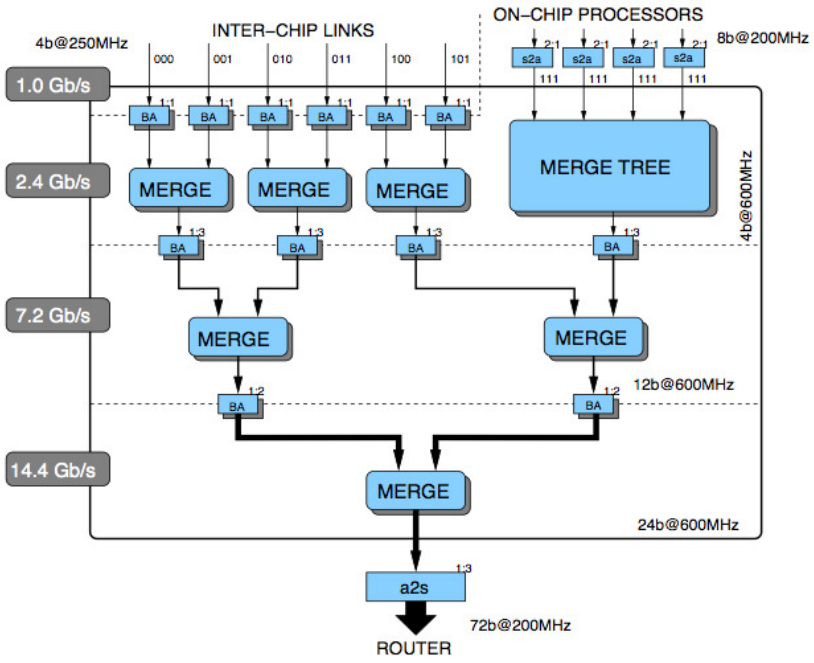
The Communications NoC carries packets between the processors on the same or different chips. It plays a central role in the system architecture. Its connectivity to the other components is shown the the chip block diagram in 'Chip organization' on page 5.

3.2.9.1 Features

- On- and inter-chip links
- Router which handles multicast, point-to-point, nearest neighbour and fixed-route packets.
- Arbiter to merge all sources into a sequential packet stream into the Router.
- Individual links can be reset to clear blockages and deadlocks.

3.2.9.2 Input structure

The input structure is a tree Arbiter which merges the various sources of packets into a single stream. Its structure is illustrated below. The numbers indicate source tagging of the packets.



3.2.9.3 Output structure

The Router produces separate outputs to all on-chip processor nodes and to the off-chip links, so the output connectivity is a set of individual self-timed links.

3.2.10 Router

The Router is responsible for routing all packets that arrive at its input to one or more of its outputs. It is responsible for routing multicast neural event packets, which it does through an associative multicast router subsystem, point-to-point packets (for which it uses a look-up table), nearest- neighbour packets (using a simple algorithmic process), fixed-route packet routing (defined in a register), default routing (when a multicast packet does not match any entry in the multicast router) and emergency routing (when an output link is blocked due to congestion or hardware failure).

Various error conditions are identified and handled by the Router, for example packet parity errors, time-out, and output link failure.

3.2.10.1 Features

- 1,024 programmable associative multicast (MC) routing entries.
  - associative routing based on source 'key';
  - with flexible 'don't care' masking;
- look-up table routing of point-to-point (P2P) packets.
- routing of nearest-neighbour (NN) and fixed-route (FR) packets.
- support for 40- and 72-bit packets.
- default routing of unmatched multicast packets.
- automatic 'emergency' re-routing around failed links.
  - programmable wait time before emergency routing and before dropping packet.
- pipelined implementation to route 1 packet per cycle (peak).
  - back-pressure flow control;
  - power-saving pipeline control.
- failure detection and handling:
  - packet parity error;
  - time-expired packet;
  - output link failure;
  - packet framing (wrong length) error.

3.2.10.2 Description

Packets arrive from other nodes via the link receiver interfaces and from internal processor nodes and are presented to the router one-at-a-time. The Arbiter is responsible for determining the order of presentation of the packets, but as each packet is handled independently

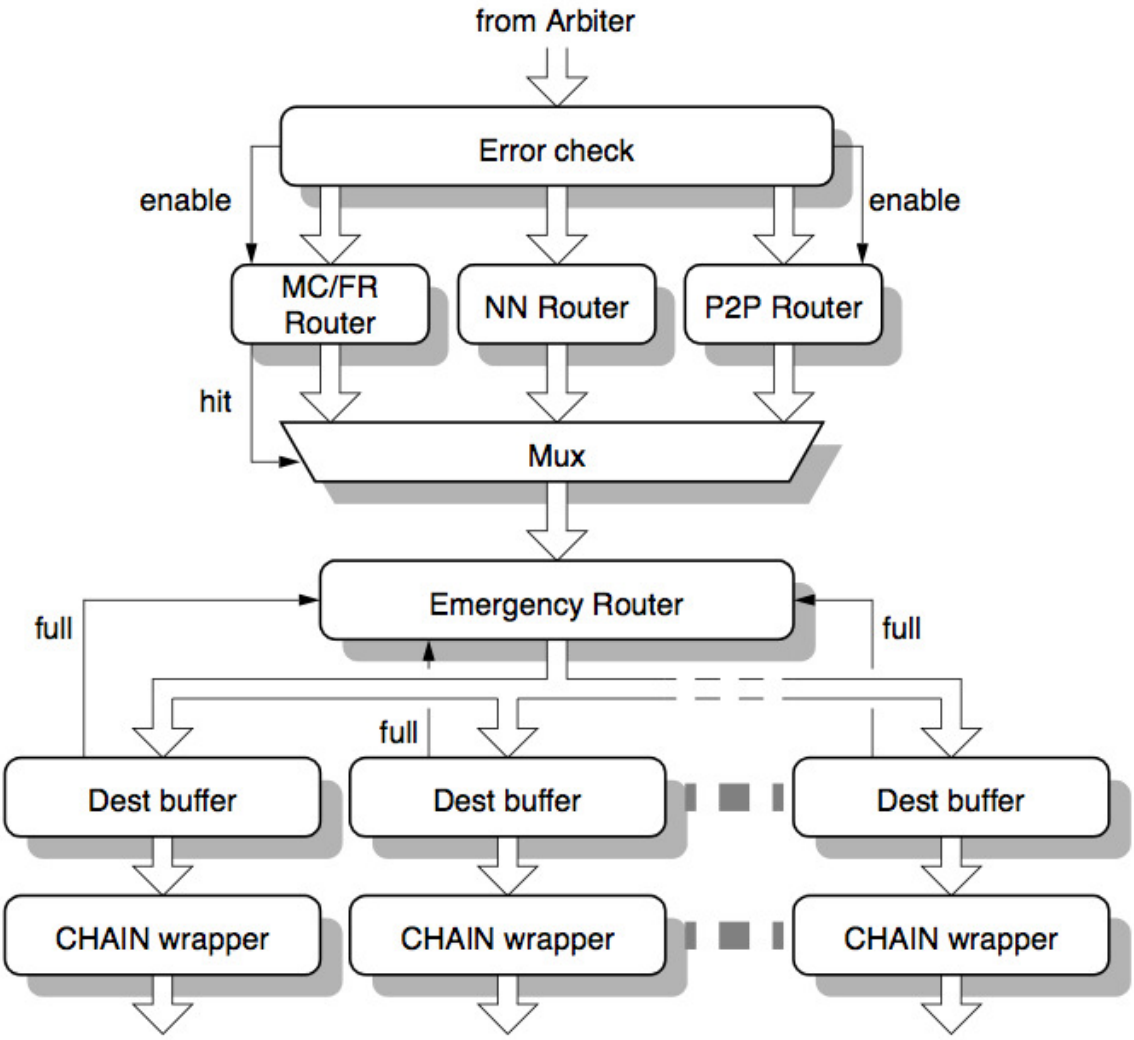
the order is unimportant (though it is desirable for packets following the same route to stay in order).

Each multicast packet contains an identifier that is used by the Router to determine which of the outputs the packet is sent to. These outputs may include any subset of the output links, where the packet may be sent via the respective link transmitter interface, and/or any subset of the internal processor nodes, where the packet is sent to the respective Communications Controller.

For the neural network application the identifier can be simply a number that uniquely identifies the source of the packet – the neuron that generated the packet by firing. This is 'source address routing'. In this case the packet need contain only this identifier, as a neural spike is an 'event' where the only information is that the neuron has fired. The Router then functions simply as a look- up table where for each identifier it looks up a routing word, where each routing word contains 1 bit for each destination (each link transmitter interface and each local processor) to indicate whether or not the packet should be passed to that destination.

3.2.10.3 Internal organization

The internal organization of the Router is illustrated in the figure below.



Packets are passed as complete 40- or 72-bit units from the Arbiter, together with the identity of the Rx interface that the packet arrived through (for nearest-neighbour, emergency and default routing). The first stage of processing here is to identify errors. The second stage passes the packet to the appropriate routing engines – the multicast (MC) router is activated only if the packet is error-free and of multicast or fixed-route type, the point-to-point (P2P) handles point-to-point packets while the NN router handles nearest-neighbour packets and also deals with default and error routing. The output of the router stage is a vector of destinations to which the packet should be relayed. The third stage is the emergency routing mechanism for handling failed or congested links, which it detects using 'full' signals fed

back from the individual destination output buffers.

3.2.10.4 Multicast (MC) router

The MC router uses the routing key in the MC packet to determine how to route the packet. The router has 1,024 look-up entries, each of which has a mask, a key value, and an output vector. The packet's routing key is compared with each entry in the MC router. For each entry it is first ANDed with the mask, then compared with the entry's key. If it matches, the entry's output vector is used to determine where the packet is sent; it can be sent to any subset (including all) of the local processors and the output links.

Thus, to programme an MC entry three writes are required: to the key, its mask and the corresponding vector. A mask of FFFFFFFF ensures all the key bits are used; if any mask bits are '0' the corresponding key bits should also be '0', otherwise the entry will not match. This can be exploited to ensure that unused entries are invalid. The effect of the various combinations of bit values in the mask[] and key[] regions is summarized in the table below:

key[]	mask[]	Function
0	0	don't care - bit matches
1	0	bit misses - entry invalidated
0	1	match 0
1	1	match 1

Thus a particular entry [i] will match only if:

- wherever a bit in the mask[i] word is 1, the corresponding bit in the MC packet routing word is the same as the corresponding bit in the key[i] word, AND
- wherever a bit in the mask[i] word is 0, the corresponding bit in the key[i] word is also 0.

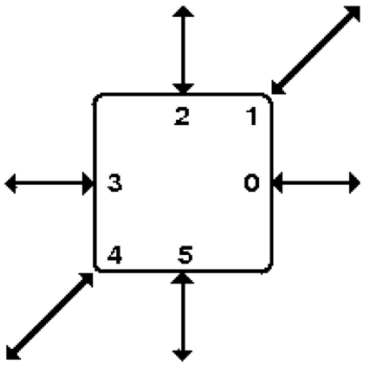
Note that the MC Router CAM is not initialised at reset. Before the Router is enabled all CAM entries must be initialised by software. Unused mask[] entries should be initialised to 0000000, and unused key[] entries should be initialised to FFFFFFFF. This invalidates every bit in the word, ensuring that the word will miss even in the presence of minor component failures.

The matching is perfomed in a parallel ternary associative memory, with a RAM used to store the output vectors. The associative memory can be set up so that more than one entry matches an incoming routing key; in this case the matching entry at the lowest address determines the output vector to be used. Multiple simultaneous matches can also be used to improve test efficiency.

If no entry matches an MC packet's routing key then default routing is employed - the packet is sent to the output link opposite the input link through which it arrived. Packets from local processors cannot be default-routed; the router table must have a valid entry for every locally-sourced packet.

The MC output vector assignment is detailed in the table below:

MC vector entry	Output port	Direction
bit[0]	Tx0	East
bit[1]	Tx1	North-East
bit[2]	Tx2	North
bit[3]	Tx3	West
bit[4]	Tx4	South-West
bit[5]	Tx5	South
bit[6]	Processor 0	Local
bit[7]	Processor 1	Local
. . .	. . .	. . .
bit[23]	Processor 17	Local



If any of the multicast packet's output links are blocked the packet is stalled for a time 'wait1' (see 'r0: Router control register' on page 44). When that time expires any blocked external outputs (i.e. links 0-5) will attempt to divert to the next lower number link, modulo 6 (see section 10.9 on page 42) and retry for a further period, 'wait2'. If two potential outputs become unblocked at the same time the original choice is preferred.

A packet which is diverted is typed as specified in 'emergency routing' on page 32. If a packet of such a type is received the router will attempt to output it as a 'reverting' packet to the output with the next lower number to the input on which it was received. If this should also be a normal packet then conventional multicast routing also takes place.

The routing tables should not be set up so that a packet paths cross each other. If the packet is programmed to do this then it is not possible to differentiate between an intended and a reverting packet; the 'reverting' designation takes priority.

A received reverting packet is routed normally if it is recognised by the router, otherwise it is 'default' routed to the link numbered two greater (mod 6) than the input link.

fixed-route (FR) packets

The FR router uses the same mechanism as the MC router although the packets do not have a key field. Instead, all packets of this type are routed to the same output vector, as specified in r33. Emergency routing is handled identically to MC packets.

This mechanism is intended to facilitate monitoring and debugging by routing data towards a point which connects with a host system.

3.2.10.5 The point-to-point (P2P) router

The P2P router uses the 16-bit destination ID in a point-to-point packet to determine which output the packet should be routed to. There is a 3-bit entry for each of the 64K destination IDs. Each 3-bit entry is decoded to determine whether the packet is delivered to the local Monitor Processor or one of the six output links, or dropped, as detailed in the table below:

P2P table entry	Output port	Direction
000	Tx0	East
001	Tx1	North-East
010	Tx2	North
011	Tx3	West
100	Tx4	South-West
101	Tx5	South
110	none (drop packet)	none
111	Monitor Processor	Local

The 3-bit entries are packed into an 8K entry x 24-bit SRAM lookup table. The 24-bit words hold entries 0, 8, 16, ... in bits [2:0], 1, 9, 17, . . . in bits [5:3], etc.

3.2.10.6 The nearest-neighbour (NN) router

Nearest-neighbour packets are used to initialise the system and to perform run-time flood-fill and debug functions. The routing function here is to send 'normal' NN packets that arrive from outside the node (i.e. via an Rx link) to the monitor processor and to send NN packets that are generated internally to the appropriate output (Tx) link(s). This is to support a flood-fill load process.

In addition, the 'peek/poke' form of NN packet can be used by neighbouring systems to access System NoC resources. Here an NN poke 'write' packet (which is a peek/poke type with a 32-bit payload) is used to write the 32-bit data defined in the payload to a 32-bit address defined in the address/operation field. An NN peek 'read' packet (which is a peek/poke type without a 32-bit payload) uses the 32-bit address defined in the address/operation field to read from the System NoC and returns the result (as a 'normal' NN packet) to the neighbour that issued the original packet using the Rx link ID to identify that source. This 'peek/poke' access to a neighbouring chip's principal resources can be used to investigate a non-functional chip, to re-assign the Monitor Processor from outside, and generally to get good visibility into a chip for test and debug purposes.

As the peek/poke NN packets convey only 32-bit data payloads the bottom 2 bits of the address should always be zero. All peek/poke NN packets return a response to the sender, with bit 0 of the address set to 1. Bit 1 will also be set to 1 if there was a bus error at the target. Peeks return a 32-bit data payload; pokes return without a payload. default and error routing In addition, the NN router performs default and error routing functions.

3.2.10.7 Time phase handling

The Router maintains a 2-bit time phase signal that is used to delete packets that are out-of date. The time phase logic operates as follows:

- locally-generated packets will have the current time phase inserted (where appropriate);
- a packet arriving from off-chip will have its time phase checked, and if it is two phases old it will be deleted (dropped, and copied to the Error registers).

3.2.10.8 Packet error handler

The packet error handler is a routing engine that simply flags the packet for dropping to the Error registers if it detects any of the following:

- a packet parity error;
- a packet that is two time phases old;
- a packet that is the wrong length.

The Monitor Processor can be interrupted to deal with packets dropped with errors.

3.2.10.9 Emergency routing

If a link fails (temporarily, due to congestion, or permanently, due to component failure) action will be taken at two levels:

- The blocked link will be detected in hardware and subsequent packets rerouted via the other two sides of a triangle of which the suspect link was an edge, being initially re-routed via the link which is rotated one link clockwise from the blocked link (so if link Tx0 fails, link Tx5 is used, etc).
- The Monitor Processor will be informed. It can track the problem using a diagnostic counter:
  - if the problem was due to transient congestion, it will note the congestion but do nothing further;
  - if the problem was due to recurring congestion, it will negotiate and establish a new route for some of the traffic using this link;
  - if the problem appears permanent, it will establish new routes for all of the traffic using this link.

  The hardware support for these processes include:
  - default routing processes in adjacent nodes that are invoked by flagging the packet as an emergency type;
  - mechanisms to inform the Monitor Processor of the problem;
  - means of inducing the various types of fault for testing purposes.

  Emergency rerouting around the triangle requires additional emergency packet types for MC and FR packets. P2P packets will find their own way to their destination following emergency routing.

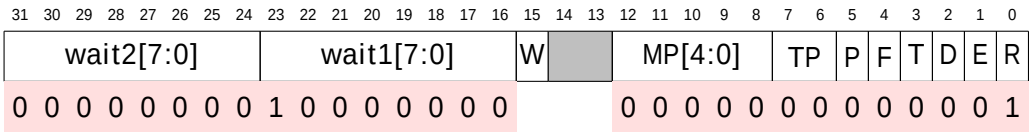
3.2.10.10 Register summary

Base address: 0xe1000000 (buffered write), 0xf1000000 (unbuffered write).

Name	Offset	R/W	Function
r0: control	0x00	R/W	Router control register
r1: status	0x04	R	Router status
r2: error header	0x08	R	error packet control byte and flags
r3: error routing	0x0C	R	error packet routing word
r4: error payload	0x10	R	error packet data payload
r5: error status	0x14	R	error packet status
r6: dump header	0x18	R	dumped packet control byte and flags
r7: dump routing	0x1C	R	dumped packet routing word
r8: dump payload	0x20	R	dumped packet data payload
r9: dump outputs	0x24	R	dumped packet intended destinations
r10: dump status	0x28	R	dumped packet status
r11: diag enables	0x2C	R/W	diagnostic counter enables
r12: timing ctr ctl	0x30	R/W	timing counter controls
r13: cycle ctr	0x34	R	counts Router clock cycles
r14: busy cyc ctr	0x38	R	counts emergency router active cycles
r15: no wt pkt ctr	0x3C	R	counts packets that do not wait to be issued
r16-31: dly hist	0x40-7C	R	packet delay histogram counters
r32: diversion	0x80	R/W	divert default packets
r33: FR route	0x84	R/W	fixed-route packet routing vector
rFN: diag filter	0x200-23C	R/W	diagnostic count filters (N = 0-15)
rCN: diag count	0x300-33C	R/W	diagnostic counters (N = 0-15)
rT1: test register	0xF00	R	hardware test register 1
rT2: test key	0xF04	R/W	hardware test register 2 - CAM input test key
route[1023:0]	0x4000	R/W	MC Router routing word values
key[1023:0]	0x8000	W	MC Router key values
mask[1023:0]	0xC000	W	MC Router mask values
P2P[8191:0]	0x10000	R/W	P2P Router routing entries (8 3-bit entries/word)

3.2.10.11 Register details

r0: Router control register



The functions of these fields are described in the table below:

Name	bits	R/W	Function
wait2[7:0]	31:24	R/W	wait time before dropping packet
wait1[7:0]	23:16	R/W	wait time before emergency routing
W	15	W	re-initialise wait counters
MP[4:0]	12:8	R/W	Monitor Processor ID number
TP	7:6	R/W	time phase (c.f. packet time stamps)
P	5	R/W	enable count of packet parity errors
F	4	R/W	enable count of packet framing errors
T	3	R/W	enable count of packet time stamp errors
D	2	R/W	enable dump packet interrupt
E	1	R/W	enable error packet interrupt
R	0	R/W	enable packet routing

The wait times (defined by wait1[] and wait2[]) are stored in a floating point format to give a wide range of values with high accuracy at low values combined with simple implementation using a binary pre-scaler and a loadable counter. Each 8-bit field is divided into a 4-bit mantissa M[3:0] = wait[3:0] and a 4-bit exponent E[3:0] = wait[7:4]. The wait time in clock cycles is then given by:

$$\text{wait} = (M + 16 - 2^{4-E}).2^{E} \text{ for } E \leq 4;$$
$$\text{wait} = (M + 16).2^{E} \text{ for } E > 4;$$

Note that wait[7:0] = 0x00 gives a wait time of zero, and the wait time increases monotonically with wait[7:0]; wait[7:0] = 0xFF is a special case and gives an infinite wait time - wait forever.

There is a small semantic difference between wait1[7:0] and wait2[7:0]:

- wait1[7:0] defines the number of cycles the Router will re-try after the first failed cycle before attempting emergency routing; wait1[] = 0 will attempt normal routing once and then try emer- gency routing.

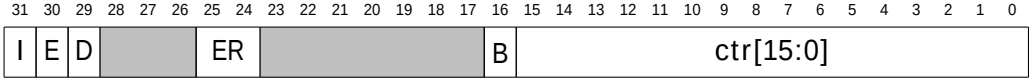
- wait2[7:0] is the number of cycles during which emergency routing will be attempted before the packet is dumped; wait2[] = 0 therefore effectively disables emergency routing.

If r0 is written when one of the wait counters is running, writing a 1 to W (bit[15]) will cause the active counter to restart from the new value written to it. This enables the Monitor Processor to clear a deadlocked 'wait forever' condition. If 0 is written to W the active counter will not restart but will use the new wait time value the next time it is invoked.

Note that the Router is enabled after reset. This is so that a neighbouring chip can peek and poke a chip that fails after reset using NN packets, to diagnose and possibly fix the cause of failure.

r1: Router status

All Router interrupt request sources are visible here, as is the current status of the emergency routing system.



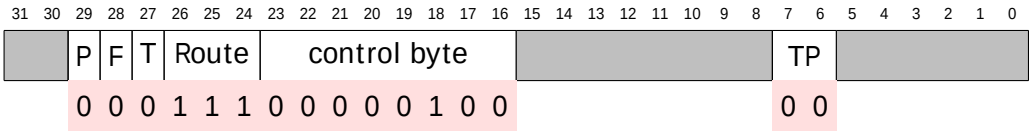
The functions of these fields are described in the table below:

Name	bits	R/W	Function
I: interrupt active	31	R	combined Router interrupt request
E: error int	30	R	error packet interrupt active
D: dump int	29	R	dump packet interrupt active
ER[1:0]	25:24	R	Router output stage status (empty, full but unblocked, blocked in wait1, blocked in wait2)
B	16	R	busy - active packet(s) in Router pipeline
ctr[15:0]	15:0	R	diagnostic counter interrupt active

The Router generates three interrupt request outputs that are handled by the VIC on each processor: diagnostic counter event interrupt, dump interrupt and error interrupt. These correspond to the OR of ctr[15:0], D and E respectively. The interrupt requests are cleared by reading their respective status registers: r5, r10 and r2N.

r2: error header

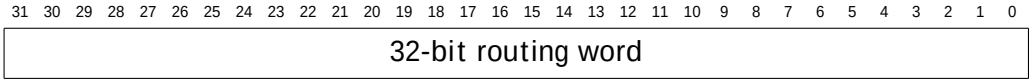
A packet which contains an error is copied to r2-5. Once a packet has been copied (indicated by bit[31] of r5 being set) any further error packet is ignored, except that it can update the sticky bits in r5 (and errors of the types specified in r0 are counted in r5).



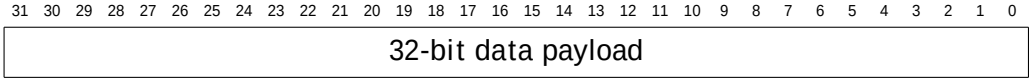
The functions of these fields are described in the table below:

Name	bits	R/W	Function
P: parity	29	R	packet parity error
F: framing error	28	R	packet framing error
T: TP error	27	R	packet time stamp error
Route	26:24	R	Rx route field of error packet
Control byte	23:16	R	control byte of error packet
TP: time phase	7:6	R	time phase when packet received

r3: error routing word

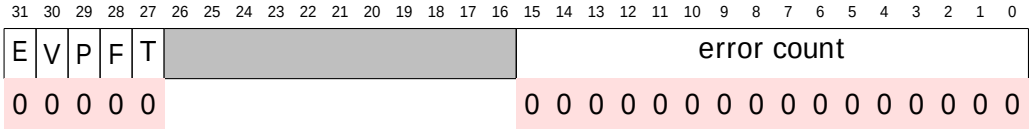


r4: error data payload



r5: error status

This register counts error packets, including time stamp, framing and parity errors as enabled by r0[5:3]. The Monitor Processor resets r5[31:27] and the error count by reading its contents.

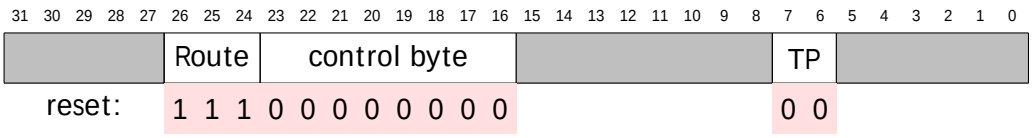


The functions of these fields are described in the table below:

Name	bits	R/W	Function
E: error	31	R	error packet detected
V: overflow	30	R	more than one error packet detected
P: parity	29	R	packet parity error (sticky)
F: framing error	28	R	packet framing error (sticky)
T: TP error	27	R	packet time stamp error (sticky)
error count	15:0	R	16-bit saturating error count

r6: dump header

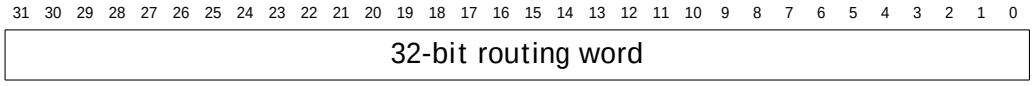
A packet which is dumped because it cannot be routed to its destination is copied to r6-10. Once a packet has been dumped (indicated by bit[31] of r10 being set) any further packet that is dumped is ignored, except that it can update the sticky bits in r10 (and can be counted by a diagnostic counter).



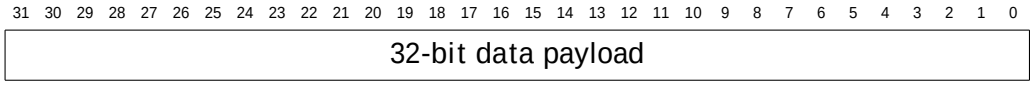
The functions of these fields are described in the table below:

Name	bits	R/W	Function
Route	26:24	R	Rx route field of dumped packet
Control byte	23:16	R	control byte of dumped packet
TP: time phase	7:6	R	time phase when packet dumped

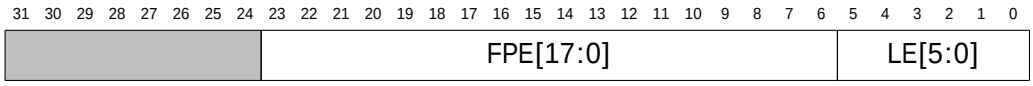
r7: dump routing word



r8: dump data payload



r9: dump outputs

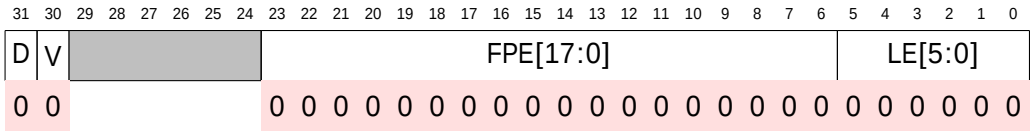


The functions of these fields are described in the table below:

Name	bits	R/W	Function
FPE[17:0]	23:6	R	Fascicle Processor link error caused dump
LE[5:0]	5:0	R	Tx link transmit error caused packet dump

r10: dump status

The Monitor Processor resets r10 by reading its contents.

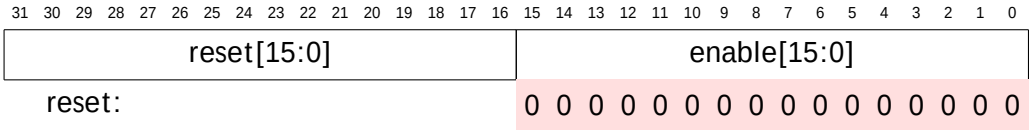


The functions of these fields are described in the table below:

Name	bits	R/W	Function
D: dumped	31	R	packet dumped
V: overflow	30	R	more than one packet dumped
FPE[17:0]	23:6	R	Fascicle Proc link error caused dump (sticky)
LE[5:0]	5:0	R	Tx link error caused dump (sticky)

r11: diagnostic counter enable/reset

This register provides a single control point for the 16 diagnostic counters, enabling them to count events over a precisely controlled time period.



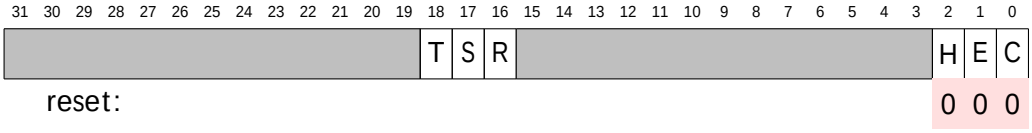
The functions of these fields are described in the table below:

Name	bits	R/W	Function
reset[31:16]	31:16	R	write a 1 to reset diagnostic counter15. . . 0
enable[15:0]	15:0	R	enable diagnostic counter 15. . . 0

Writing a 0 to reset[15:0] has no effect. Writing a 1 clears the respective counter.

r12: timing counter controls

This register controls the cycle counters in registers r13, r14 & r15, and in the delay histogram registers r16-r31.

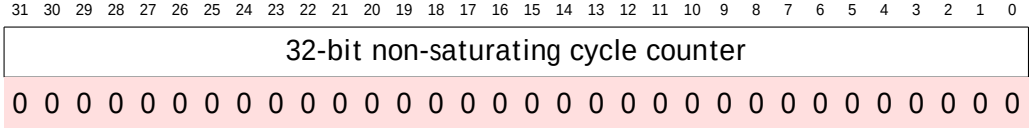


The functions of these fields are described in the table below:

Name	bits	R/W	Function
T	18	W	reset histogram
S	17	W	reset emergency router active cycle counter
R	16	W	reset cycle counter
H	2	R/W	enable histogram
E	1	R/W	enable emergency router active cycle counter
C	0	R/W	enable cycle counter

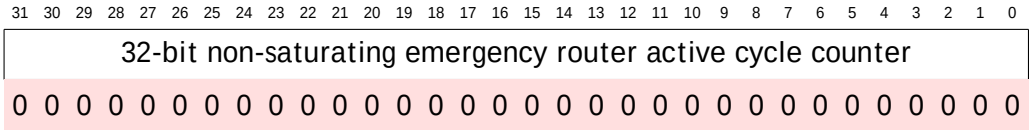
Writing a 0 to R, S or T has no effect. Writing a 1 clears the respective counter.

r13: cycle count



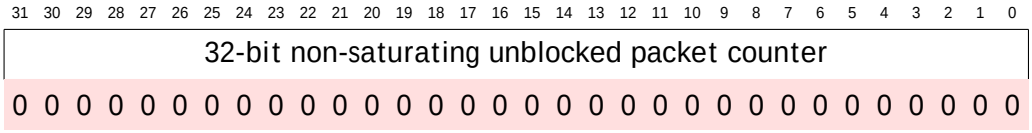
r13, when enabled by r12, simply counts the number of Router clock cycles.

r14: emergency router active cycle count



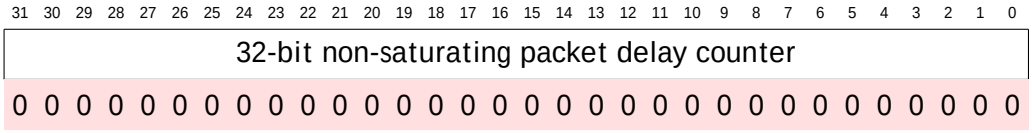
r14, when enabled by r12, counts the number of cycles for which the emergency router is actively seeking a route for a packet. This equals the number of packets plus the number of stall cycles.

r15: unblocked packet count



r15, when enabled by r12, counts the number of packets which pass through undelayed by congested output links.

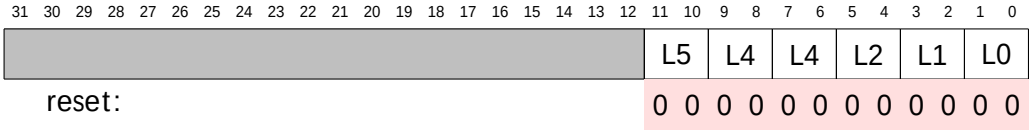
r16-31: packet delay histogram



r16-r31, when enabled by r12, count the number of times a packet is delayed due to link congestion, each register counting delays within a range of clock cycles. r15 counts the zero delay component of the histogram. These counters use the same pre-scaling as wait1 in r0, so the histogram effectively records the value in the wait mantissa at the time the congestion resolves.

r32: diversion

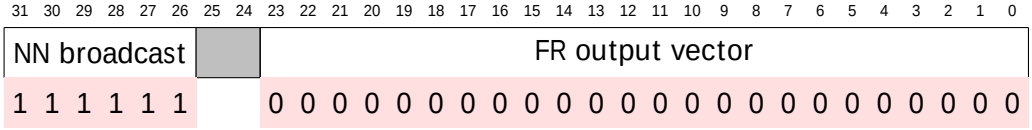
This register allows default-routed MC packets to be redirected in the case when their default path is unavailable, for example as a result of a complete node failure.



The 2-bit L0 field can be set to 00 for normal behaviour of packets default routed from link 0, to x1 to divert those packets to the local Monitor Processor, or to 10 to destroy the

packets. L1 likewise controls default routed packets that arrive through link 1, etc.

r33: fixed-route packet routing



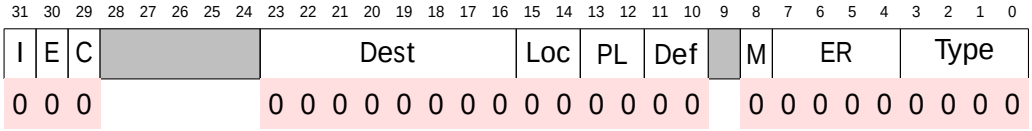
r33 routes fixed-route (type 3) packets to off-chip links and local processors in exactly the same way, with the same bit allocation, as an MC output vector as described in section 10.4 on page 39.

In addition, the 'NN broadcast' bits[31:26] define which links an NN broadcast packet is sent through. A 1 indicates an active link, and bit[26] is for link 0, bit[27] link 1, etc.

rFN: diagnostic filter control

The Router has 16 diagnostic counters (N = 0..F) each of which counts packets passing through the Router filtered on packet characteristics defined here. A packet is counted if it has characteristics that match with a '1' in each of the 6 fields. Setting all bits [24:10, 7:0] to '1' will count all packets.

A diagnostic counter may (optionally) generate an interrupt on each count. The C bit[29] is a sticky bit set when a counter event occurs and is cleared whenever this register is read.



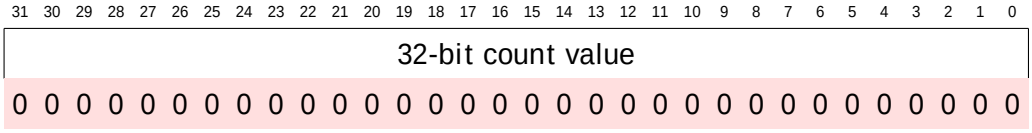
The functions of these fields are described in the table below:

Name	bits	R/W	Function
I	31	R	counter interrupt active: I = E AND C
E	30	R/W	enable interrupt on counter event
C	29	R	counter event has occurred (sticky)
Dest	24:16	R/W	packet dest (Tx link[5:0], MP, local ¬MP, dump)
Loc	15:14	R/W	local [x1]/non-local[1x] packet source
PL	13:12	R/W	packets with [x1]/without [1x] payload
Def	11:10	R/W	default [x1]/non-default [1x] routed packets
M	8	R/W	Emergency Routing mode
ER	7:4	R/W	Emergency Routing field = 3, 2, 1 or 0
Type	3:0	R/W	packet type: fr, nn, p2p, mc

If M (bit[8]) = 0 the Emergency Routing field matches that of the incoming packet, before any local Emergency Routing, so this can be used to count packets that have been Emergency Routed by a previous Router but not those that are Emergency Routed here.

If M = 1 the Emergency Routing field is matched against outgoing packets to destinations selected in the Dest field. If any outgoing packet to a selected destination matches the ER field the diagnostic count will be incremented. (Note that packets to internal destinations cannot be emergency routed and so have ER = 0.)

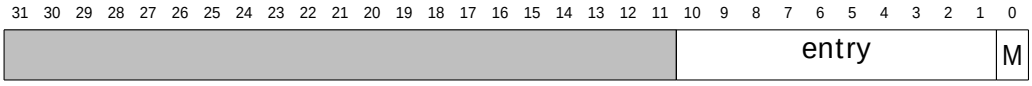
rCN: diagnostic counters



Each of these counters can be used to count selected types of packets under the control of the corresponding rFN. The counter can have any value written to it, and will increment from that value when respective events occur. If an event occurs as the counter is being written it will not be counted. To avoid missing an event it is better to avoid writing the counter; instead, read it at the start of a time period and subtract this value from the value read at the end of the period to get a count of the number of events during the period.

rT1: hardware test register 1

This register is used only for hardware test purposes, and has no useful functions for the application programmer.

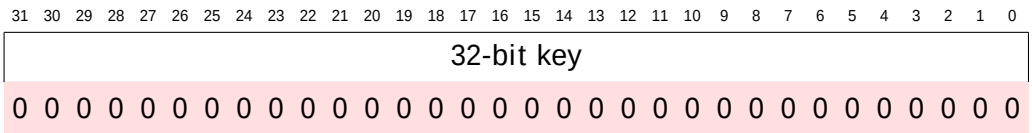


The functions of these fields are described in the table below:

Name	bits	R/W	Function
M	0	R	MC router associative look-up ‘miss’ output
entry	10:1	R	MC router associative look-up entry address

The input key used for the associative look-up whenever this register is read is in register T2.

rT2: hardware test register 2



This register holds the key presented to the association input of the multicast router when register T1 is read.

3.2.10.12 Fault-tolerance

The Communications Router has some internal fault-tolerance capacity, in particular it is possible to map out a failed multicast router entry. This is a useful mechanism as the multicast router dominates the silicon area of the Communications Router.

There is also capacity to cope with external failures. Emergency routing will attempt to bypass a faulty or blocked link. In the event of a node (or larger) failure this will not be sufficient. In order to tolerate a chip failure several expedients can be employed on a local basis:

- P2P packets can be routed around the obstruction;
- MC packets with a router entry can be redirected appropriately.

In most cases, default MC packets cannot sensibly be trapped by adding table entries due to their (almost) infinite variety. To allow rerouting, these packets can be dropped to the Monitor Processor on a link-by-link basis using the diversion register. In principle they can then be routed around the obstruction as P2P payloads before being resurrected at the opposite side.

Should the Monitor Processor become overwhelmed, it is also possible to use the diversion register to eliminate these packets in the Router; this prevents them blocking the Router pipeline whilst waiting for a timeout and thus delaying viable traffic.

Fault detection

- packet parity errors.
- packet time-phase errors.
- packet unroutable errors (e.g. a locally-sourced multicast packet which doesn't match any entry in the multicast router).
- wrong packet length.

Fault isolation

- a multicast router entry can be disabled if it fails - see initialisation guidance above.

Reconfiguration

- since all multicast router entries are identical the function of any entry can be relocated to a spare entry.
- if a router becomes full a global reallocation of resources can move functionality to a different router.

3.2.10.13 Test

Production test

The ternary CAM used in the multicast router has access for parallel testing, so a processor can write a value to all locations and see if an input with 1 bit flipped results in a hit or a miss. The CAM is not directly readable - attempts to read this space will result in bus errors - and must be tested by association. To do this a key must first be written into register rT2. A subsequent read of register rT1 will then indicate if that key has associated with any CAM entries. If it has not then rT1⟨0⟩ will be set and the other bits of this register will be undefined; if one or more of the entries are matched then the one at the lowest address in the CAM will be indicated in the 'entry' field.

All RAMs have read-write access for test purposes.

3.2.11 Inter-chip transmit and receive interfaces

Inter-chip communication is implemented by extending the Communications NoC from chip to chip. In order to sustain throughput, there is a protocol conversion at each chip boundary from standard CHAIN 3-of-6 return-to-zero to 2-of-7 non-return-to-zero. The interfaces include logic to minimise the risk of a protocol deadlock caused by glitches on the inter-chip wires.

3.2.11.1 Features

- transmit (Tx) interface:
  - converts on-chip 3-of-6 RTZ symbol into off-chip 2-of-7 NRZ symbol;
  - disable control input;
  - reset input.
- receive (Rx) interface:
  - converts off-chip 2-of-7 NRZ symbol into on-chip 3-of-6 RTZ symbol;
  - disable control input;
  - reset input.

3.2.11.2 Programmer view

The only programmer-accessible features implemented in these interfaces are software reset and a disable control, both accessed via the System Controller. In normal operation these interfaces provide transparent connectivity between the routing network on one chip and those on its neighbours.

3.2.11.3 Fault-tolerance

The fault inducing, detecting and resetting functions are controlled from the System Controller (see 'System Controller' on page 66). The interfaces are 'glitch hardened' to greatly reduce the probability of a link deadlock arising as a result of a glitch on one of the inter-chip wires. Such a glitch may introduce packet errors, which will be detected and handled elsewhere, but it is very unlikely to cause deadlock. It is expected that the link reset function will not be required often.

Fault insertion

- an input controlled by the System Controller causes the interface to deadlock (by disabling it).

Fault detection

- Monitor Processors should regularly test link functionality.

Fault isolation

- the interface can be disabled to isolate the chip-to-chip link. This input from the System Controller is also used to create a fault.

Reconfiguration

- the link interface can be reset by the System Controller to attempt recovery from a fault.
- the link interface can be isolated and an alternative route used.

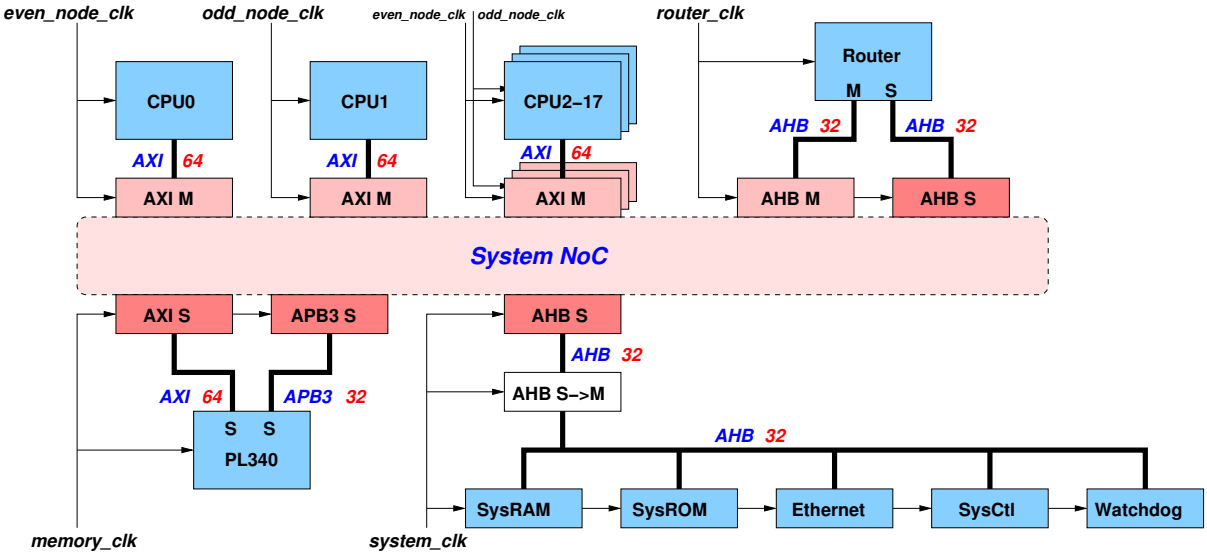
3.2.12 System NoC

The System NoC has a primary function of connecting the ARM968 processors to the SDRAM interface. It is also used to connect the processors to system control and test functions, and for a variety of other purposes.

3.2.12.1 Features

- supports full bandwidth block transfers between the SDRAM and the ARM968 processors.
- the Router is an additional initiator for system debug purposes.
- can be reset (in subsections) to clear deadlocks.
- multiple targets:
  - SDRAM interface - ARM PL340
  - System RAM
  - System ROM
  - Ethernet interface
  - System Controller
  - Watchdog Timer.
  - Router configuration registers

3.2.12.2 Organisation



3.2.13 SDRAM interface

The SDRAM interface connects the System NoC to an off-chip SDRAM device. It is the ARM PL340, described in ARM document DDI 0331D.

3.2.13.1 Features

- control for external Mobile DDR SDRAM memory device
- memory request queue
- out of order request sequencing to maximise memory throughput
- AXI interface to System NoC
- delay-locked loop (DLL) to realign SDRAM data strobes with the input data streams

3.2.13.2 Register summary

Base address: 0xe0000000 (buffered write), 0xf0000000 (unbuffered write).

User registers

The following registers allow normal user programming of the PL340 SDRAM interface:

Name	Offset	R/W	Function
r0: status	0x00	R	memory controller status
r1: command	0x04	W	PL340 command
r2: direct	0x08	W	direct command
r3: mem_cfg	0x0C	R/W	memory configuration
r4: refresh_prd	0x10	R/W	refresh period
r5: CAS_latency	0x14	R/W	CAS latency
r6: t_dqss	0x18	R/W	write to DQS time
r7: t_mrd	0x1C	R/W	mode register command time
r8: t_ras	0x20	R/W	RAS to precharge delay
r9: t_rc	0x24	R/W	active bank x to active bank x delay
r10: t_rcd	0x28	R/W	RAS to CAS minimum delay
r11: t_rfc	0x2C	R/W	auto-refresh command time
r12: t_rp	0x30	R/W	precharge to RAS delay
r13: t_rrd	0x34	R/W	active bank x to active bank y delay
r14: t_wr	0x38	R/W	write to precharge delay
r15: t_wtr	0x3C	R/W	write to read delay
r16: t_xp	0x40	R/W	exit power-down command time
r17: t_xsr	0x44	R/W	exit self-refresh command time
r18: t_esr	0x48	R/W	self-refresh command time
id_n_cfg	0x100	R/W	QoS settings
chip_n_cfg	0x200	R/W	external memory device configuration
user_status	0x300	R	DLL test and status inputs
user_config0	0x304	W	DLL test and control outputs
user_config1	0x308	W	DLL fine-tune control

Test and ID registers

In addition, there are test and ID registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
int_cfg	0xE00	R/W	integration configuration register
int_inputs	0xE04	R	integration inputs register
int_outputs	0xE08	W	integration outputs register
periph_id_n	0xFE0-C	R	PL340 peripheral ID byte registers
pcell_id_n	0xFF0-C	R	PL340 Prime Cell ID byte registers

See ARM document DDI 0331D for further details of the test registers.

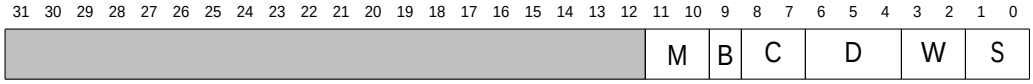
Restrictions on when registers may be modified

Normally the PL340 registers will be initialised during system start-up and then left alone. Restrictions on when the registers may be safely modified are detailed in the PL340 datasheet, ARM doccument DDI 0331D.

The DLL test and control outputs and the DLL fine-tune control registers should only be written to when the PL340 is quiescent and no processor is issuing an SDRAM access or has one pending.

3.2.13.3 Register details

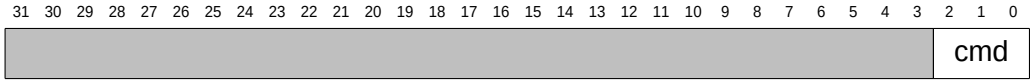
r0: memory controller status



The functions of these fields are described in the table below:

Name	bits	R/W	Function
M: monitors	11:10	R	Number of exclusive access monitors (0, 1, 2, 4)
B: banks	9	R	Fixed at 1'b01 = 4 banks on a chip
C: chips	8:7	R	Number of different chip selects (1, 2, 3, 4)
D: DDR	6:4	R	DDR type: 3b'011 = Mobile DDR
W: width	3:2	R	Width of external memory: 2'b01 = 32 bits
S: status	1:0	R	Config, ready, paused, low-power

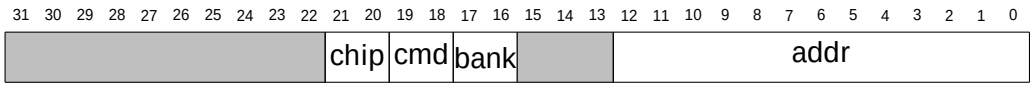
r1: memory controller command



The function of this field is described in the table below:

Name	bits	R/W	Function
cmd: command	2:0	W	Go, sleep, wake-up, pause, config, active_pause

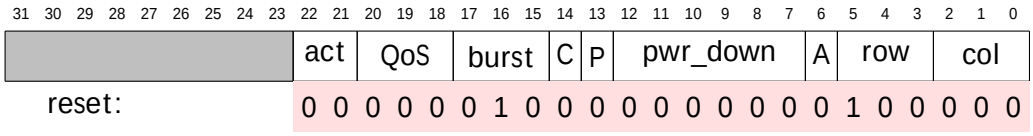
r2: direct command



This register is used to pass a command directly to a memory device attached to the PL340. The functions of these fields are described in the table below:

Name	bits	R/W	Function
chip	21:20	W	chip number
cmd	19:18	W	command passed to memory device
bank	17:16	W	bank passed to memory device
addr[13:0]	13:0	W	address passed to memory device

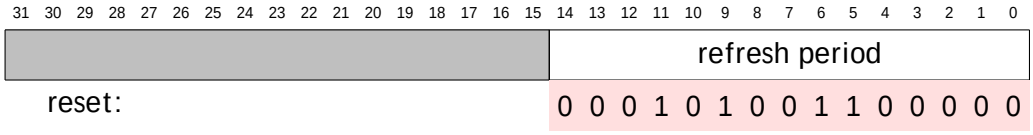
r3: memory configuration



The function of this field is described in the table below:

Name	bits	R/W	Function
act	22:21	R/W	active chips: number for refresh generation
QoS	20:18	R/W	selects the 4-bit QoS field from the AXI ARID
burst	17:15	R/W	burst length (1, 2, 4, 8, 16)
C	14	R/W	stop memory clock when no access
P	13	R/W	auto-power-down memory when inactive
pwr_down	12:7	R/W	# memory cycles before auto-power-down
A	6	R/W	position of auto-pre-charge bit (10/8)
row	5:3	R/W	number of row address bits (11-16)
col	2:0	R/W	number of column address bits (8-12)

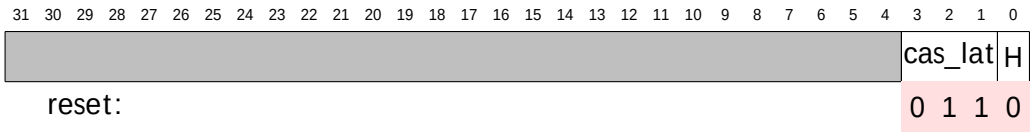
r4: refresh period



The function of this field is described in the table below:

Name	bits	R/W	Function
refresh period	14:0	R/W	memory refresh period in memory clock cycles

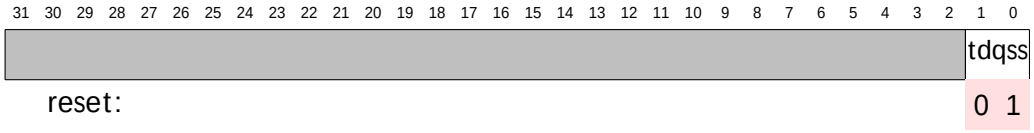
r5: CAS latency



The functions of these fields are described in the table below:

Name	bits	R/W	Function
cas_lat	3:1	R/W	CAS latency in memory clock cycles
H	0	R/W	CAS half cycle - must be set to 1'b0

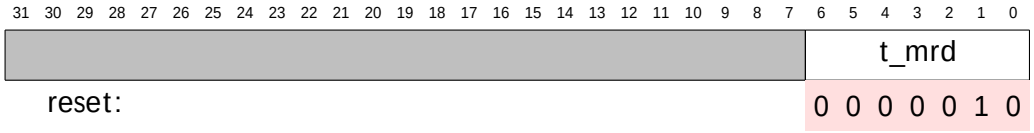
r6: t_dqss



The function of this field is described in the table below:

Name	bits	R/W	Function
tdqss	1:0	R/W	write to DQS in memory clock cycles

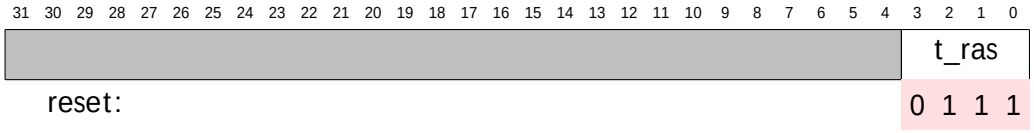
r7: t_mrd



The function of this field is described in the table below:

Name	bits	R/W	Function
t_mrd	6:0	R/W	mode reg cmnd time in memory clock cycles

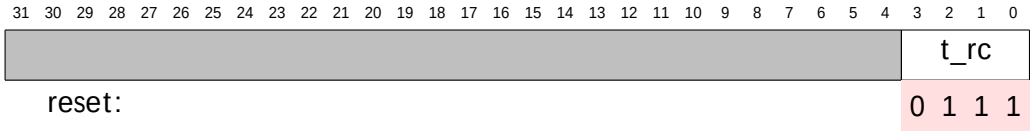
r8: t_ras



The function of this field is described in the table below:

Name	bits	R/W	Function
t_ras	3:0	R/W	RAS to precharge time in memory clock cycles

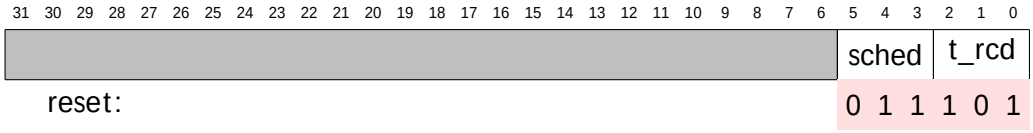
r9: t_rc



The function of this field is described in the table below:

Name	bits	R/W	Function
t_rc	3:0	R/W	Bank x to bank x delay in memory clock cycles

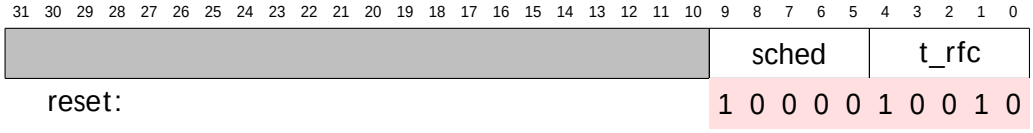
r10: t_rcd



The functions of these fields are described in the table below:

Name	bits	R/W	Function
t_rcd	2:0	R/W	RAS to CAS min delay in memory clock cycles
sched	5:3	R/W	RAS to CAS min delay in aclk cycles −3

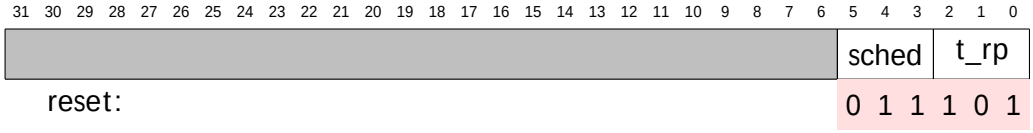
r11: t_rfc



The functions of these fields are described in the table below:

Name	bits	R/W	Function
sched	9:5	R/W	Auto-refresh cmnd time in aclk cycles −3
t_rfc	4:0	R/W	Auto-refresh cmnd time in memory clock cycles

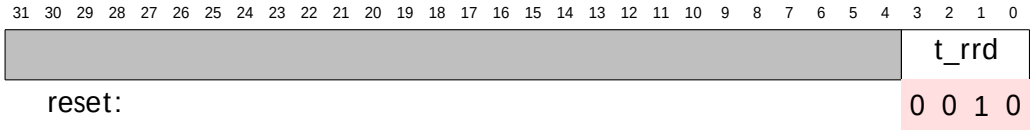
r12: t_rp



The functions of these fields are described in the table below:

Name	bits	R/W	Function
sched	5:3	R/W	Precharge to RAS delay in aclk cycles −3
t_rp	2:0	R/W	Precharge to RAS delay in memory clock cycles

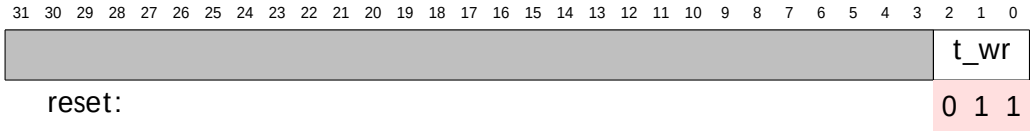
r13: t_rrd



The function of this field is described in the table below:

Name	bits	R/W	Function
t_rrd	3:0	R/W	Bank x to bank y delay in memory clock cycles

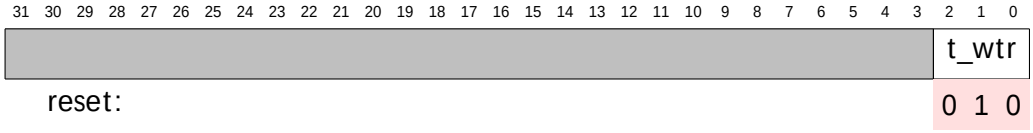
r14: t_wr



The function of this field is described in the table below:

Name	bits	R/W	Function
t_wr	2:0	R/W	Write to precharge dly in memory clock cycles

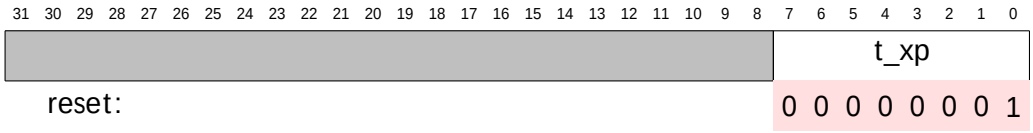
r15: t_wtr



The function of this field is described in the table below:

Name	bits	R/W	Function
t_wtr	2:0	R/W	Write to read delay in memory clock cycles

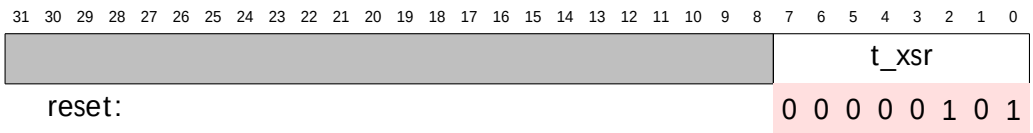
r16: t_xp



The function of this field is described in the table below:

Name	bits	R/W	Function
t_xp	7:0	R/W	Exit pwr-dn cmnd time in memory clock cycles

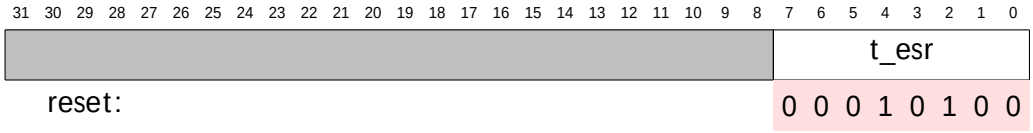
r17: t_xsr



The function of this field is described in the table below:

Name	bits	R/W	Function
t_xsr	7:0	R/W	Exit self-rfsh cmnd time in mem clock cycles

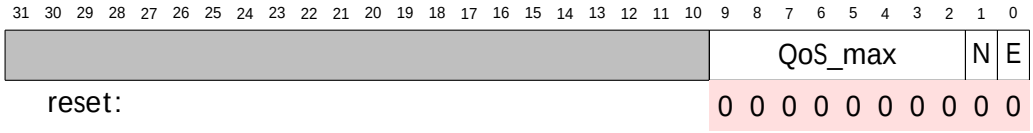
r18: t_esr



The function of this field is described in the table below:

Name	bits	R/W	Function
t_esr	7:0	R/W	Self-refresh cmnd time in memory clock cycles

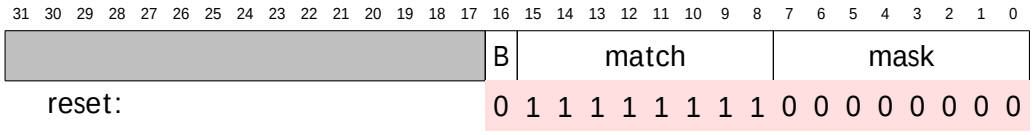
id_n_cfg



The functions of these fields are described in the table below:

Name	bits	R/W	Function
QoS_max	9:2	R/W	maximum QoS
N	1	R/W	minimum QoS
E	0	R/W	QoS enable

chip_n_cfg



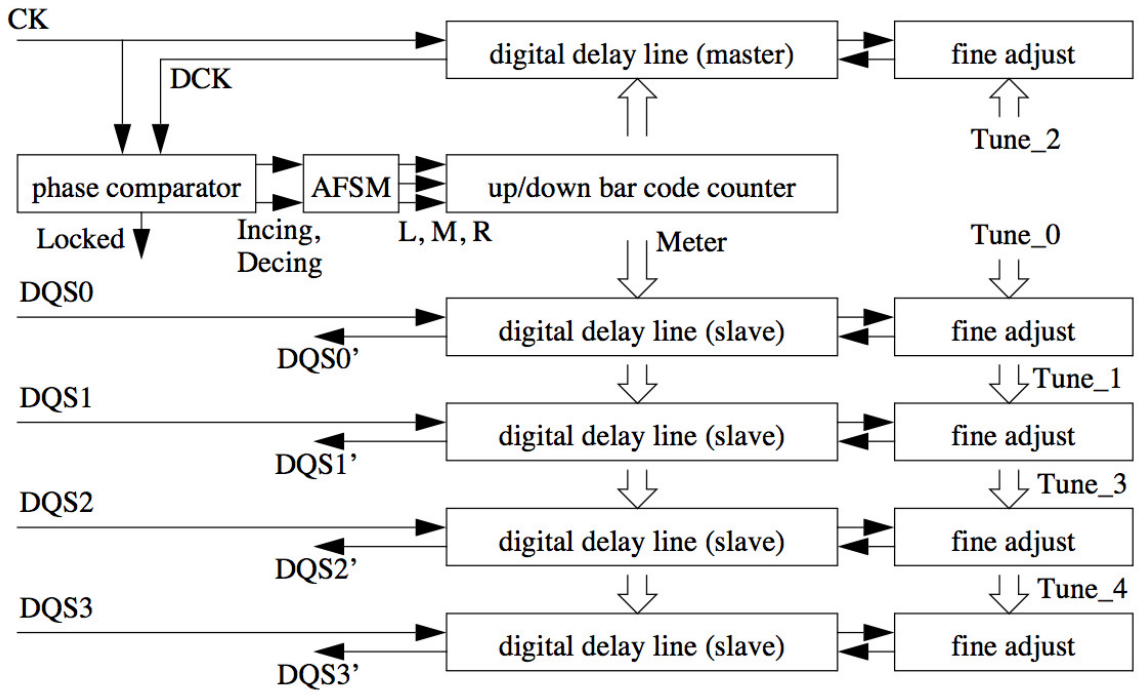
There is one of these registers for each external chip that is supported. The functions of these fields are described in the table below:

Name	bits	R/W	Function
B	16	R/W	bank-rol-column/row-bank-column
match	15:8	R/W	address match
mask	7:0	R/W	address mask

3.2.13.4 The delay-locked loop (DLL)

The SDRAM interface incorporates a delay-locked loop which, though outside the PL340, is controlled via the PL340 user status and configuration registers.

The general organisation of the DLL is shown below:



The basic operation is that a reference clock, CK, running at twice the SDRAM clock (i.e. nominally 333 MHz for a 166 MHz SDRAM), is passed through a master delay line and the output, DCK, inverted and compared with the original clock. A phase comparator drives an asynchronous finite state machine (AFSM) that in turn drives an up/down bar code counter to line these two signals up. The SDRAM data strobes, DQS0-3, are passed through matched delay lines to line up with the middle of the data valid period. Software can fine-tune the individual strobe timings.

There is a 6th, spare, delay line, that can be used if any of the five primary delay lines fails.

user_status: DLL test and status inputs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
									L	M	R		K	I	D	C3	S3	C2	S2	C1	S1	C0	S0		Meter							
reset:									0	0	0		0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0

The function of these fields is described in the table below:

Name	bits	R/W	Function
L, M, R	22:20	R	3-phase bar-code control output
K: locKed	18	R	Phase comparator is locked
I: Incing	17	R	Phase comparator is increasing delay
D: Decing	16	R	Phase comparator is reducing delay
C0, C1, C2, C3	9,11,13,15	R	Clock faster than strobe 0-3
S0, S1, S2, S3	8,10,12,14	R	Strobe 0-3 faster than Clock
Meter	6:0	R	Current position of bar-code output

user_config0: DLL test and control outputs

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
							E	TL	L	M	R	T5	ID	I	D					S5	S4	S3	S2	S1	S0							
reset:							0	0	0	0	0	0	0	0	0					0	0	0	0	0	0	0	0	0	0	0	0	0

The function of these fields is described in the table below:

Name	bits	R/W	Function
E: Enable	24	W	Enable DLL (0 = reset DLL)
TL: Test_LMR	23	W	Enable forcing of L, M, R
L, M, R	22:20	W	Force 3-phase bar-code control inputs
T5: Test_5	19	W	Substitute delay line 5 for 4 for testing
ID: Test_ID	18	W	Enable forcing of Incing and Decing
I: Test_Incing	17	W	Force Incing (if ID = 1)
D: Test_Decing	16	W	Force Decing (if ID = 1)
S0-S5	11:0	W	Input selects for the 6 delay lines {def, alt, 0, 1}

The default inputs for the 6 delay lines selected by S0-S5 are Tune_2 (master); Tune_0 (DQS0); Tune_1 (DQS1); Tune_3 (DQS2); Tune_4 (DQS3) as shown in the figure above.

The alternative inputs for the 6 delay lines are: Tune_3 (master); Tune_1 (DQS0); Tune_2 (DQS1); Tune_4 (DQS2); Tune_5 (DQS3).

user_config1: DLL fine-tune control

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								Tune_5				Tune_4				Tune_3				Tune_2				Tune_1				Tune_0			
reset:								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The function of these fields is described in the table below:

Name	bits	R/W	Function
Tune0 . . . 5	23:0	W	Fine tuning control on delay lines 0 . . . 5

3.2.13.5 Fault-tolerance

Fault insertion

The DLL can be driven by software into pretty much any defective state.

Fault detection

The DLL delay lines can be tested for stuck-at faults and relative timing accuracy.

Fault isolation

A defective or out-or-spec delay line can be isolated.

Reconfiguration

A defective or out-or-spec delay line can be isolated and replaced by using the spare delay line.

3.2.14 System Controller

The System Controller incorpora tes a number of functions for system start-up, fault-tolerance testing (invoking, detecting and resetting faults), general performance monitoring, etc.

3.2.14.1 Features

- 'Arbiter' read-sensitive register bit to determine Monitor Processor ID at start-up.
- 32 test-and-set registers for general software use, e.g. to enforce mutually exclusive access to critical data structures.
- individual interrupt, reset and enable controls and 'processor OK' status bits for each processor.
- sundry parallel IO and test and control registers.
- PLL and clock management registers.

3.2.14.2 Register summary

Base address: 0xe2000000 (buffered write), 0xf2000000 (unbuffered write).

These registers may only be accessed by a processor executing in a privileged mode; any attempt to access the System Controller from user-mode code will return a bus error. Only aligned word accesses are supported - misaligned word or byte or half-word accesses will return a bus error.

Name	Offset	R/W	Function
r0: Chip ID	0x00	R	Chip ID register (hardwired)
r1: CPU disable	0x04	R/W	Each bit disables a processor
r2: Set CPU IRQ	0x08	R/W	Writing a 1 sets a processor's interrupt line
r3: Clr CPU IRQ	0x0C	R/W	Writing a 1 clears a processor's interrupt line
r4: Set CPU OK	0x10	R/W	Writing a 1 sets a CPU OK bit
r5: Clr CPU OK	0x14	R/W	Writing a 1 clears a CPU OK bit
r6: CPU Rst Lv	0x18	R/W	Level control of CPU resets
r7: Node Rst Lv	0x1C	R/W	Level control of CPU node resets
r8: Sbsys Rst Lv	0x20	R/W	Level control of subsystem resets
r9: CPU Rst Pu	0x24	R/W	Pulse control of CPU resets
r10: Node Rst Pu	0x28	R/W	Pulse control of CPU node resets
r11: Sbsys Rst Pu	0x2C	R/W	Pulse control of subsystem resets
r12: Reset Code	0x30	R	Indicates cause of last chip reset
r13: Monitor ID	0x34	R/W	ID of Monitor Processor
r14: Misc control	0x38	R/W	Miscellaneous control bits
r15: GPIO pull u/d	0x3C	R/W	General-purpose IO pull up/down enable
r16: I/O port	0x40	R/W	I/O pin output register
r17: I/O direction	0x44	R/W	External I/O pin is input (1) or output (0)
r18: Set IO	0x48	R/W	Writing a 1 sets IO register bit
r19: Clear IO	0x4C	R/W	Writing a 1 clears IO register bit
r20: PLL1	0x50	R/W	PLL1 frequency control
r21: PLL2	0x54	R/W	PLL2 frequency control
r22: Set flags	0x58	R/W	Set flags register
r23: Reset flags	0x5C	R/W	Reset flags register
r24: Clk Mux Ctl	0x60	R/W	Clock multiplexer controls
r25: CPU sleep	0x64	R	CPU sleep (awaiting interrupt) status
r26-28	0x68-70	R/W	Temperature sensor registers [2:0]
r32-63: Arbiter	0x80-FC	R	Read sensitive semaphores to determine MP
r64-95: Test&Set	0x100-17C	R	Test & Set registers for general software use
r96-127: Test&Clr	0x180-1FC	R	Test & Clear registers for general software use
r128: Link disable	0x200	R/W	Disables for Tx and Rx link interfaces

3.2.14.3 Register details

r0: Chip ID

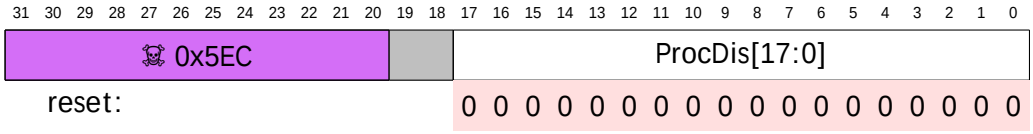
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
device												version				Year								# CPUs							
0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0

This register is configured at chip design time to hold a unique ID for the chip type. The device code is 591 in BCD. The version will increment with each design variant. Year holds the last two digits of the year of first fabrication, in BCD. The bottom byte holds the number

of CPUs on the chip.

The test chip ID is 0x59100902. The full chip ID is 0x59111012.

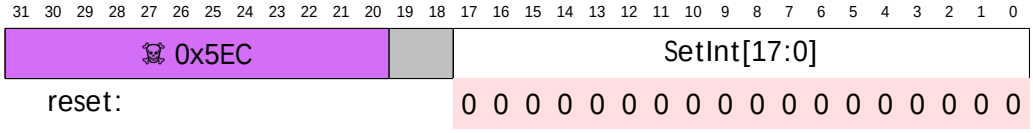
r1: CPU disable



Writing a 1 to bit[n] (n = 0. . . 17) will disable processor[n], stalling any attempted access to its local AHB and thereby preventing it from accessing any external resource. Writing a 0 will enable it. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX.

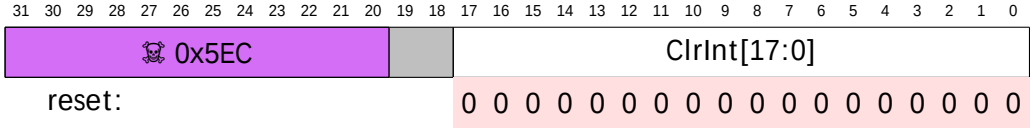
To ensure the processor is disabled in a low-power state it should be disabled and then reset via r9.Reading from this register returns the current status of all of the processor disable lines.

r2: Set CPU interrupt request



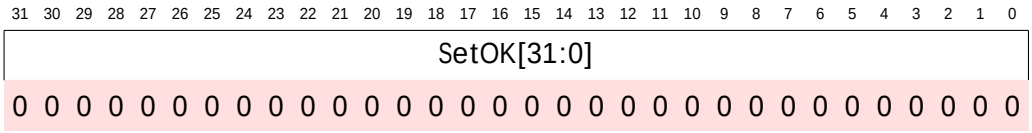
Writing a 1 to bit[n] (n = 0. . . 17) will set an interrupt request to processor[n], which can be enabled/ disabled and routed to IRQ or FIQ by that processor's local Vectored Interrupt Controller (VIC - see page 12). Writing a 0 has no effect. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of all of the processor interrupt lines.

r3: Clear CPU interrupt request



Writing a 1 to bit[n] (n = 0. . . 17) will clear an interrupt request to processor[n]. Writing a 0 has no effect. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of all of the processor interrupt lines.

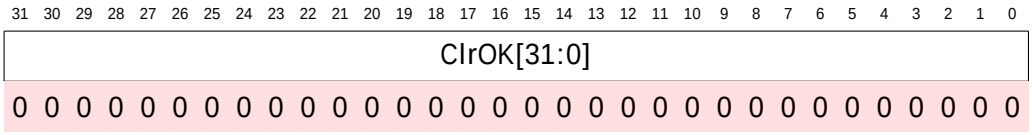
r4: Set CPU OK



Writing a 1 to bit[n] (n = 0. . . 31) will set that bit, indicating that processor[n] is believed to be functional. Writing a 0 has no effect. Reading from this register returns the current status of all of the processor OK bits. Any bits that do not correspond to a processor number can be used for any purpose - the functions of this register are entirely defined by software.

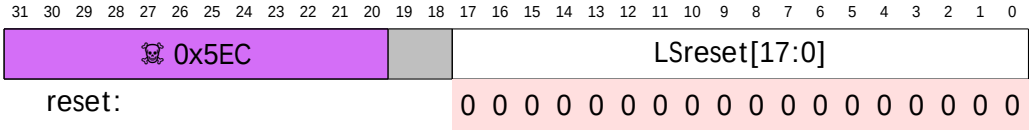
In normal use a processor will set its own bit after performing some functional self-testing. The Monitor Processor will read the register after the start-up phase to establish which processors are functional, and assign them tasks accordingly. The MP may attempt to restart faulty processors by resetting them via r6-11, or it may take them off-line by disabling their clocks via r1.

r5: Clear CPU OK



Writing a 1 to bit[n] (n = 0. . . 31) will clear that bit, indicating that processor[n] is not confirmed as functional or has detected a fault. Writing a 0 has no effect. Reading from this register returns the current status of all of the processor OK bits.

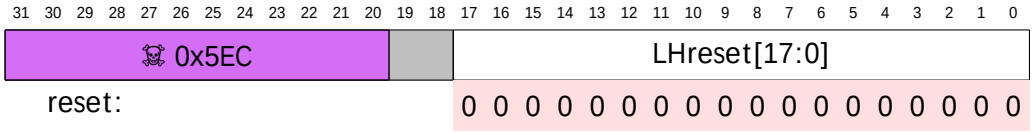
r6: CPU node soft reset - level



Writing a 1 to bit[n] (n = 0. . . 17) will set a level on the reset input of processor[n] which is ORed with the corresponding output of the pulse reset generator, r9. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of this register, that is the level before the OR with the pulse reset output.

This is a soft reset which resets the ARM9 processor core, thereby restarting its execution at the reset vector, and resets the Communication and DMA Controllers once active transactions have completed.

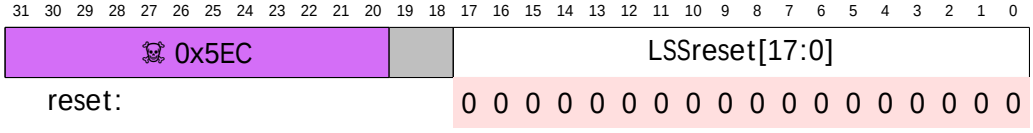
r7: CPU node hard reset - level



Writing a 1 to bit[n] (n = 0. . . 17) will set a level on the reset input of processor node[n] which is ORed with the corresponding output of the pulse reset generator, r10. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of this register, that is the level before the OR with the pulse reset output.

This is a hard reset which resets the entire ARM968 processor node, including the peripheral hardware components in that node.

r8: Subsystem reset - level

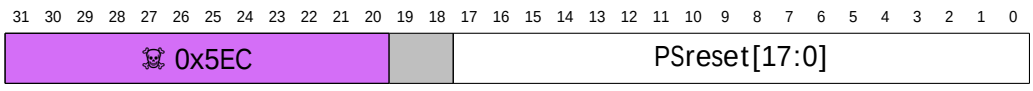


Writing a 1 to bit[n] (n = 0. . . 17) will set a level on the reset input of a subsystem. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of this register, that is the level before the OR with the pulse reset output.

The assignment of these bits to subsystems is given in the following table:

LSSreset	Reset target
0	Router
1	PL340 SDRAM controller
2	System NoC
3	Communications NoC
4-9	Tx link 0-5
10-15	Rx link 0-5
16	System AHB & Clock Gen (pulse reset only)
17	Entire chip (pulse reset only)
18-19	unassigned

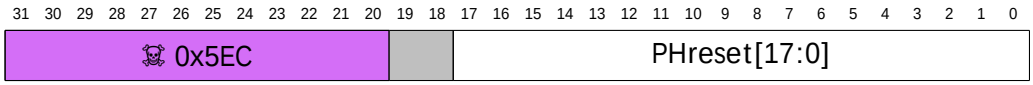
r9: CPU node soft reset - pulse



Writing a 1 to bit[n] (n = 0. . . 17) will generate a pulse (of 256 System Controller clock cycles) on the reset input of processor[n], which is ORed with the corresponding output of the reset level register r6. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of the reset lines after the OR with the level reset output.

The reset function is as described for r6.

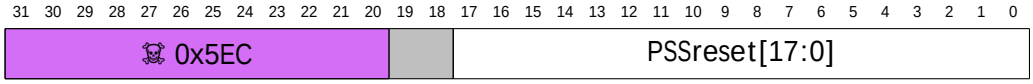
r10: CPU node hard reset - pulse



Writing a 1 to bit[n] (n = 0. . . 17) will generate a pulse (256 clock cycles long) on the reset input of processor node[n], which is ORed with the corresponding output of the reset level register r7. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of the reset lines after the OR with the level reset output.

The reset function is as described for r7.

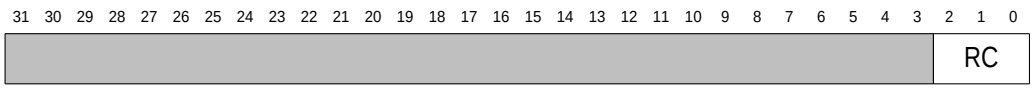
r11: Subsystem reset - pulse



Writing a 1 to bit[n] (n = 0. . . 17) will generate a pulse (256 clock cycles long) on the reset input of a subsystem. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of the reset lines after the OR with the level reset output.

The assignment of these bits to subsystems is the same as that described for r8.

r12: Reset code



These bits return a code indicating the last active reset source. The reset sources are given in the following table:

RC[2:0]	Reset source	Hard/soft reset actionbits
000	POR - Power-on reset	hard, everything
001	WDR - Watchdog reset	hard, all but MPID[4:0] in r13
010	UR - User reset	hard, all but MPID[4:0] in r13 & B in r14
011	REC - Reset entire chip (r11 bit 17)	hard, all but MPID[4:0] in r13 & B in r14
100	WDI - Watchdog interrupt	soft, only Monitor Processor if *R* = 1 in r13

The Power-on reset RC[2:0] = 000 hard resets everything, including setting MPID[4:0] = 11111 in r13 and B = 0 in r14.

WDR, UR and REC (RC[2:0] = 001, 010 or 011) do not reset MPID[4:0] in r13, which retains its value through the reset, thereby preventing the old Monitor Processor from competing to be Monitor Processor after the reset.

UR and REC (RC[2:0] = 010 or 011) do not reset B in r14, which will retain its value through the reset, thereby allowing booting from RAM.

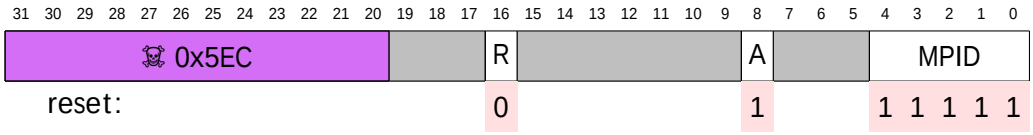
The Watchdog interrupt RC[2:0] = 100 only soft resets the Monitor Processor (with a 256 cycle pulse), and then only if this is enabled in r13.

r13: Monitor ID

This register holds the ID of the processor which has been chosen as the Monitor Processor, together with associated control bits.

Software must set the MPID value in the Router Control Register, which the Router uses to route P2P and NN packets to the Monitor Processor, to match MPID[4:0].

MPID[4:0] is initialised by power-on reset to an invalid value which does not refer to any processor. Other forms of reset do not change MPID[4:0]. It is set to the ID of the processor that wins the competition at start-up by reading its respective register r32 to r63 first.



The functions of these fields are described in the table below:

Name	bits	R/W	Function
R	16	R/W	Reset Monitor Processor on Watchdog interrupt
A	8	R/W	Write 1 to set MP arbitration bit (see r32-63)
MPID[4:0]	4:0	R/W	Monitor Processor ID

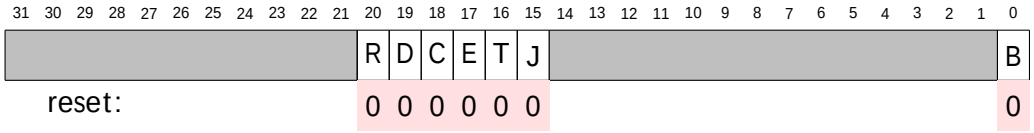
The 'R' bit causes the Watchdog interrupt signal to cause a soft reset of processor[MPID], which will override any interrupt masking by the Monitor Processor. In any case, this interrupt is available at all processor VICs and can therefore be enabled locally as an IRQ or FIQ source.

Reading bit[8] returns the current value of the MP arbitration bit (see r32-63).

For a write to r13 to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX.

r14: Misc control

This register supports general chip control.



The function of these fields is described in the table below:

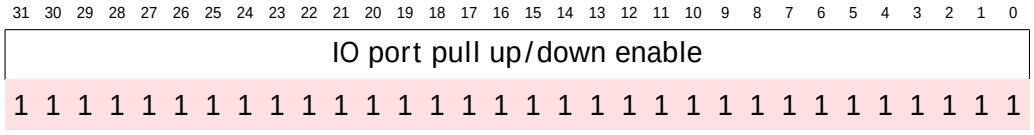
Name	bits	R/W	Function
R	20	R	read value on JTAG_RTCK pin
D	19	R	read value on JTAG_TDO pin
C	18	R	read value on Clk32 pin
E	17	R	read value on Ethermux pin
T	16	R	read value on Test pin
J	15	R/W	select on-chip (1) or off-chip (0) control of JTAG pins
B	0	R/W	map System ROM (0) or RAM (1) to Boot area

The JTAG port is controllable by software using r14 and r16. Bit[15] of r14 selects this option when high. When selected, the GPIO bits in r16 control the JTAG inputs: GPIO[27:24] drive JTAG_NTRST, JTAG_TMS, JTAG_TDI and JTAG_TCK respectively, and the JTAG outputs JTAG_TDO and JTAG_RTCK are readable via r14 as above.

When JTAG is being driven externally, reading the r14 bits[20:19] and r16 bits[27:24] returns the state of the JTAG pins.

B is reset by power-on reset (POR) and watchdog reset (WDR).

r15: GPIO pull up/down control



The functions of these bit fields are described in the table below:

bits	R/W	Function
31:29	R/W	GPIO[31:29] - on-package SDRAM control - pull-down
28:24	R/W	Unused
23:20	R/W	GPIO[23:20] & MII TxD port pull-down
19:16	R/W	GPIO[19:16] & MII RxD port pull-up
15:0	R/W	GPIO[15:0] pull-down

r16: IO port

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO port data																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This register holds a 32-bit value, most bits of which may be driven out through pins when the corresponding bit in r17 is 0. When read, the values in this register are returned. The number of physical IO pins available depends on whether or not the Ethernet interface is in use. The external EtherMux input, if driven high, enables the Ethernet Tx_D[3:0] and Rx_D[3:0] onto the pins used for IO[23:16]. If EtherMux is low these pins are available for general-purpose IO use.

The functions of these bit fields are described in the table below:

bits	R/W	Function
31:29	R/W	On-package SDRAM control
28	R/W	Unused
27:24	R/W	Can drive the JTAG interface
23:20	R/W	IO pins or MII TxD
19:16	R/W	IO pins or MII RxD
15:0	R/W	IO pins

Note: GPIO[15:14] can be configured to access the spare delay line in the DLL under the control of the external Test pin. If Test = 1 then spare_DLL_input = GPIO[14] and GPIO[15] = spare_DLL_output; if Test = 0 GPIO[15:14] connect to the System Controller GPIO pins.

r17: IO direction

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IO port direction																															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

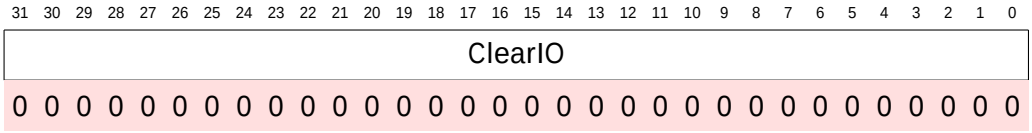
This register determines whether each IO port bit is an input (1) or an output (0). Setting a bit to an input does not invalidate the corresponding bit in r16 - that value will be held in r16 until explicitly changed by a write to r16. When read, this register returns the value last written.

r18: Set IO

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SetIO																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

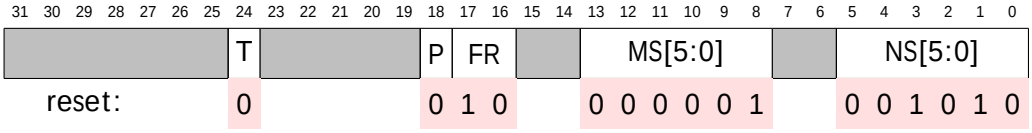
Writing a 1 sets the corresponding bit in r16. Writing a 0 has no effect.
Reading this register returns the values on the IO pins (if present).

r18: Clear IO



Writing a 1 clears the corresponding bit in r16. Writing a 0 has no effect. Reading this register returns the values on the IO pins (if present).

r20: PLL1 control, and register 21: PLL2 control

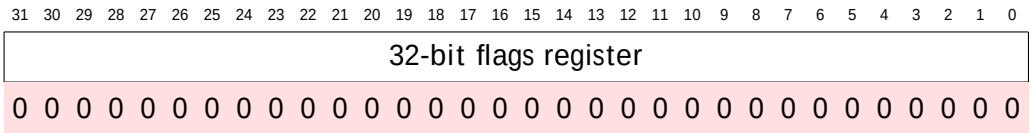


The function of these fields is described in the table below:

Name	bits	R/W	Function
T	24	R/W	test (=0 for normal operation)
P	18	R/W	Power UP
FR[1:0]	17:16	R/W	frequency range (25-50, 50-100, 100-200, 200-400 MHz)
MS[5:0]	13:8	R/W	output clock divider
NS[5:0]	5:0	R/W	input clock multiplier

The PLL output clock frequency, with a 10 MHz input clock, is given by 10 × NS/MS. Thus setting NS[5:0] = 010100 [=20] and MS[5:0] = 000001 [=1] will give 200 MHz.

r22: Set flags



Writing a 1 to any bit position sets the corresponding bit in the flags register. Writing a 0 has no effect. Reading returns the value of the flags register.

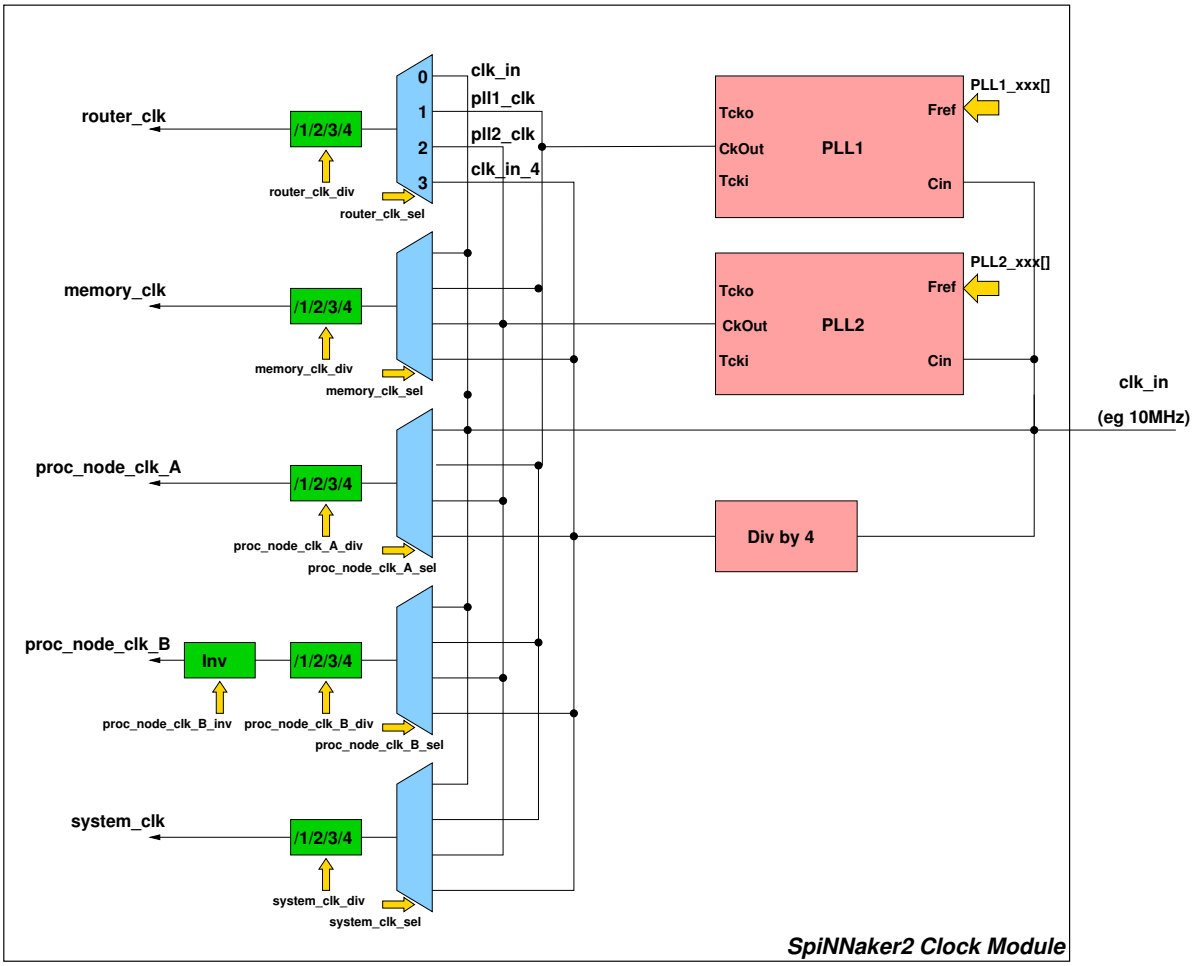
r23: Reset flags

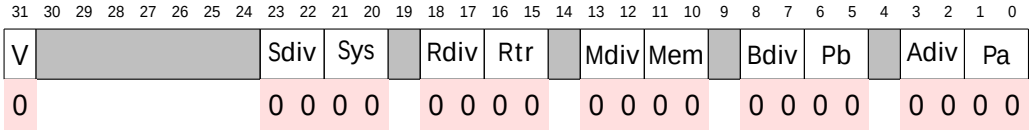
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32-bit flags register																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Writing a 1 to any bit position sets the corresponding bit in the flags register. Writing a 0 has no effect. Reading returns the value of the flags register.

r24: Clock multiplexer control

The clock generator circuits are organised as shown below:





The functions of these fields are described in the table below:

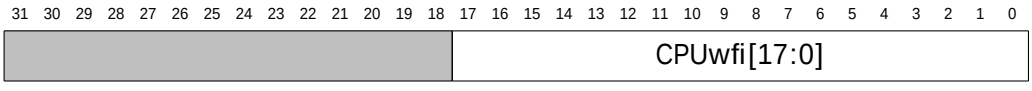
Name	bits	R/W	Function
V	31	R/W	invert CPU clock B
Sdiv[1:0]	23:22	R/W	divide System AHB clock by Sdiv + 1(= 1 – 4)
Sys[1:0]	21:20	R/W	clock selector for System AHB components
Rdiv[1:0]	18:17	R/W	divide Router clock by Rdiv + 1(= 1 – 4)
Rtr[1:0]	16:15	R/W	clock selector for Router
Mdiv[1:0]	13:12	R/W	divide SDRAM clock by Mdiv + 1(= 1 – 4)
Mem[1:0]	11:10	R/W	clock selector for SDRAM
Bdiv[1:0]	8:7	R/W	divide CPU clock B by Bdiv + 1(= 1 – 4)
Pb[1:0]	6:5	R/W	clock selector for B CPUs (0 3 5 6 9 10 12 15 17)
Adiv[1:0]	3:2	R/W	divide CPU clock A by Adiv + 1(= 1 – 4)
Pa[1:0]	1:0	R/W	clock selector for A CPUs (1 2 4 7 8 11 13 14 16)

All clock selectors choose from the same clock sources:

Sel[1:0]	Clock source
00	external 10MHz clock input
01	PLL1
10	PLL2
11	external 10MHz clock divided by 4

Clock switching is safe at any time once the PLLs have locked, which takes a defined time (maximum 80μs for the PLLs) after they have been configured.

r25: CPU sleep status

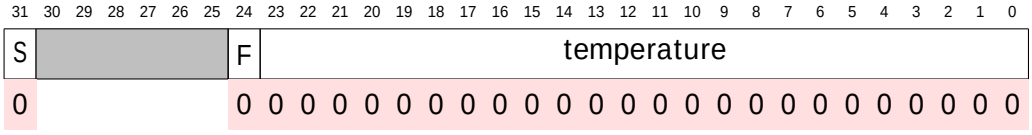


Each bit in this register indicates the state of the respective ARM968 STANDBYWFI (stand-by wait for interrupt) signal, which is active when the CPU is in its low-power sleep mode.

r26-28: Temperature sensor registers

There are three independent temperature sensors on the chip, each with its own control and sensor read-out register. The three sensors use different sensor mechanisms to enable the

temperature to be corrected for process and voltage variations.

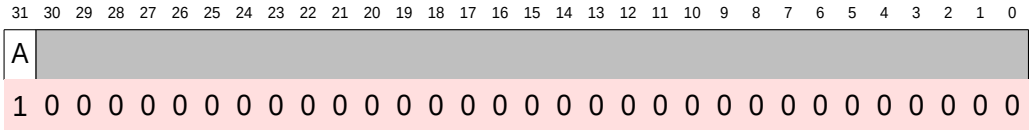


The functions of these fields are described in the table below:

Name	bits	R/W	Function
S	31	R/W	start temperature measurement
F	24	R	temperature measurement finished
temperature	23:0	R	temperature sensor reading

Setting S to 1 starts the temperature measurement process. When F reads as 1 the sensor reading is complete, and bits[23:0] may be read. Clearing S stops the sensing and clears F.

r32-63: Monitor Processor arbitration

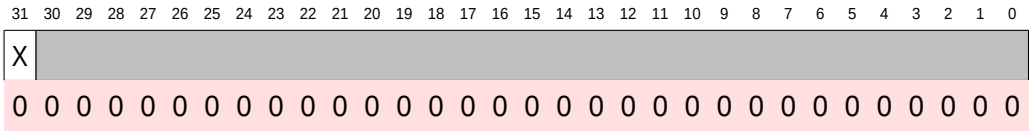


The same single-bit value 'A' appears in all registers r32 to r63.

'A' is set by a reset event (with RC[1:0] = 000, 001, 010 or 011 in r12) and can also be set by software via r13 bit[8]. A processor which has passed its self-test may read this register at address offset 0x80 + 4*N, where N is the processor's number. If A is set when the read takes place and N is not equal to the current value in r13 (the Monitor Processor ID register), 0x80000000 is returned, N is placed in r13, and A is cleared.

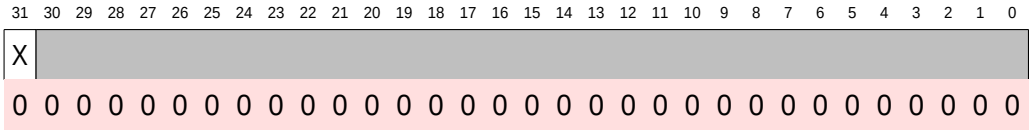
If A is clear when the read takes place, or N equals the current value in r13, then the value 0x00000000 is returned and A and r13 are unchanged.

r64-95: Test and Set



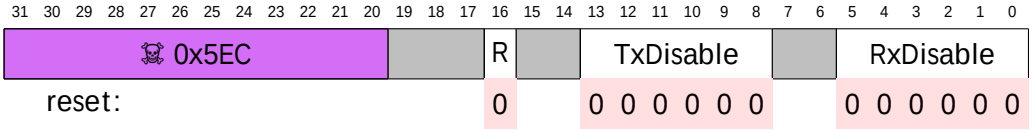
A unique single-bit value 'X' appears in each register r64 to r95. Reading each register returns 0x00000000 or 0x80000000 depending on whether its respective bit was clear or set prior to the read, and as a side-effect the bit is set by the read. Together with r96 to r127, these registers provide support for mutual exclusion primitives for inter- processor communication and shared data structures, compensating for the lack of support for locked ARM 'swap' instructions into the System RAM.

r96-127: Test and Clear



The same unique single-bit value 'X' appears in each register r96 to r127 as appears in r64 to r95 respectively. Reading each register returns 0x00000000 or 0x80000000 depending on whether its respective bit was clear or set prior to the read, and as a side-effect the bit is cleared by the read.

r128: Tx and Rx link disable



For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. The functions of these fields are described in the table below:

Name	bits	R/W	Function
R	16	R/W	Router parity control
TxDisable[5:0]	13:8	R/W	disables the corresponding link transmitter
RxDisable[5:0]	5:0	R/W	disables the corresponding link receiver

3.2.15 Ethernet MII interface

The SpiNNaker system connects to a host machine via Ethernet links. Each SpiNNaker chip includes an Ethernet MII interface, although only a few of the chips are expected to use this interface. These chips will require an external PHY. The interface hardware operates at the frame level. All higher-level protocols will be implemented in software running on the local monitor processor.

3.2.15.1 Features

- support for full-duplex 10 and 100 Mbit/s Ethernet via off-chip PHY
- outgoing 1.5Kbyte frame buffer, for one maximum-size frame
  - outgoing frame control, CRC generation and inter-frame gap insertion
- incoming 3Kbyte frame buffer, for two maximum-size frames
  - incoming frame descriptor buffer, for up to 48 frame descriptors
  - incoming frame control with length and CRC check
  - support for unicast (with programmable MAC address), multicast, broadcast and promiscuous frame capture
  - receive error filter
- internal loop-back for test purposes
- general-purpose IO for PHY management (SMI) and PHY reset
- interrupt sources for frame-received, frame-transmitted and PHY (external) interrupt

[The implementation does not provide support for half-duplex operation (as required by a CSMA/ CD MAC algorithm), jumbo or VLAN frames.]

3.2.15.2 Using the Ethernet MII interface

The Ethernet driver software must observe a number of sequence dependencies in initialising the PHY and setting-up the MAC address before the Ethernet interface is ready for use.

Details of these issues are documented in “SpiNNaker AHB-MII module” by Brendan Lynskey. The latest version of this is v003, February 2008.

3.2.15.3 Register summary

Base address: 0xe4000000 (buffered write), 0xf4000000 (unbuffered write).

User registers

The following registers allow normal user programming of the Ethernet interface:

Name	Offset	R/W	Function
Tx frame buffer	0x0000	W	Transmit frame RAM area
Rx frame buffer	0x4000	R	Receive frame RAM area
Rx desc RAM	0x8000	R	Receive descriptor RAM area
r0: Gen command	0xC000	R/W	General command
r1: Gen status	0xC004	R	General status
r2: Tx length	0xC008	R/W	Transmit frame length
r3: Tx command	0xC00C	W	Transmit command
r4: Rx command	0xC010	W	Receive command
r5: MAC addr ls	0xC014	R/W	MAC address low bytes
r6: MAC addr hs	0xC018	R/W	MAC address high bytes
r7: PHY control	0xC01C	R/W	PHY control
r8: Interrupt clear	0xC020	W	Interrupt clear
r9: Rx buf rd ptr	0xC024	R	Receive frame buffer read pointer
r10: Rx buf wr ptr	0xC028	R	Receive frame buffer write pointer
r11: Rx dsc rd ptr	0xC02C	R	Receive descriptor read pointer
r12: Rx dsc wr ptr	0xC030	R	Receive descriptor write pointer

Test registers

In addition, there are test registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
r13: Rx Sys state	0xC034	R	Receive system FSM state (debug & test use)
r14: Tx MII state	0xC038	R	Transmit MII FSM state (debug & test use)
r15: PeriphID	0xC03C	R	Peripheral ID (debug & test use)

See “SpiNNaker AHB-MII module” by Brendan Lynskey version 003, February 2008 for further details of the test registers.

3.2.15.4 Register details

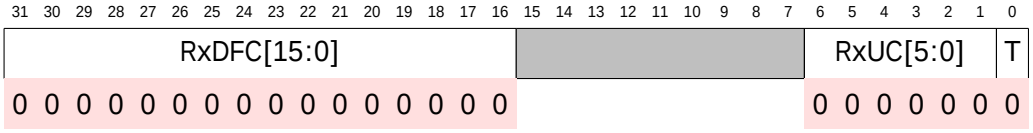
r0: General command register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																					H	D	V	P	B	M	U	F	L	R	T
reset:																					0	0	0	0	1	1	1	1	0	0	0

The functions of these fields are described in the table below:

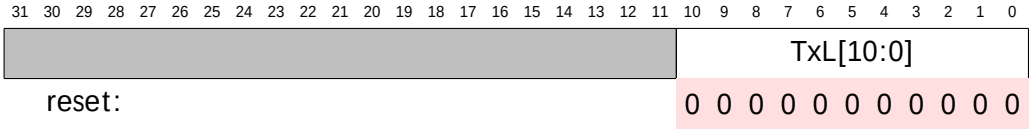
Name	bits	R/W	Function
H	10	R/W	Disable hardware byte reordering
D	9	R/W	Reset receive dropped frame count (in r1)
V	8	R/W	Receive VLAN enable
P	7	R/W	Receive promiscuous packets enable
B	6	R/W	Receive broadcast packets enable
M	5	R/W	Receive multicast packets enable
U	4	R/W	Receive unicast packets enable
F	3	R/W	Receive error filter enable
L	2	R/W	Loopback enable
R	1	R/W	Receive system enable
T	0	R/W	Transmit system enable

r1: General status register



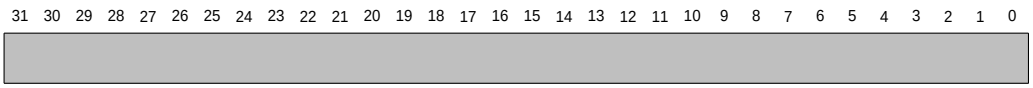
Name	bits	R/W	Function
RxDFC[15:0]	31:16	R	Receive dropped frame count
RxUC[5:0]	7:1	R	Received unread frame count
T	0	R	Transmit MII interface active

r2: Transmit frame length



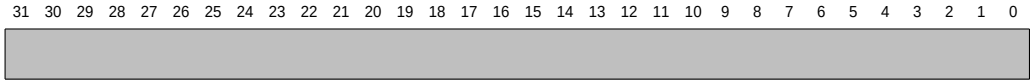
Name	bits	R/W	Function
TxL[10:0]	10:0	R/W	Length of transmit frame (60 - 1514 bytes)

r3: Transmit command register



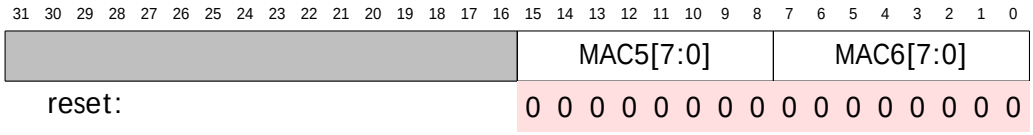
Any write to register 3 causes the transmission of a frame.

r4: Receive command register



Any write to register 4 indicates that the current receive frame has been processed and decrements the received unread frame count in register 1.

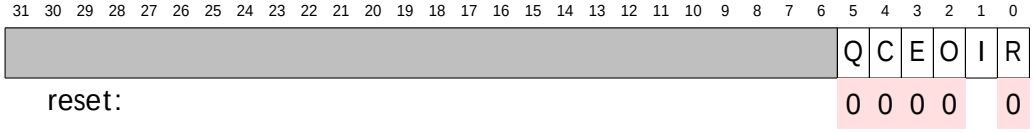
r6: MAC address high bytes



The functions of these fields are described in the table below:

Name	bits	R/W	Function
MAC5[7:0]	15:8	R/W	MAC address byte 5
MAC4[7:0]	7:0	R/W	MAC address byte 4

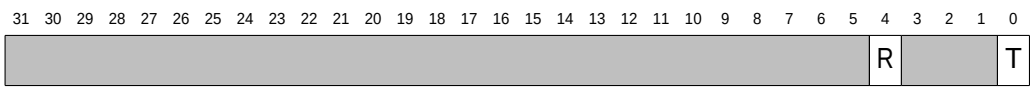
r7: PHY control



The functions of these fields are described in the table below:

Name	bits	R/W	Function
Q	5	R/W	PHY IRQn invert disable
C	4	R/W	SMI clock (active rising)
E	3	R/W	SMI data output enable
O	2	R/W	SMI data output
I	1	R	SMI data input
R	0	R/W	PHY reset (active low)

r8: Interrupt clear

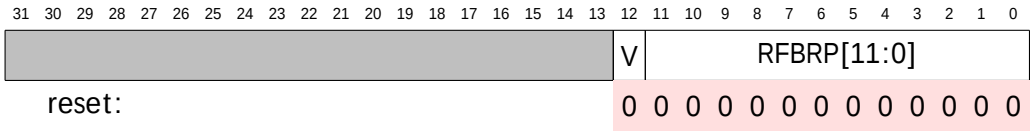


The functions of these fields are described in the table below:

Name	bits	R/W	Function
R	4	W	Clear receive interrupt request
T	0	W	Clear transmit interrupt request

Writing a 1 to bit [0] if this register clears a pending transmit frame interrupts. Writing a 1 to bit [4] clears a pending receive frame interrupt. There is no requirement to write a 0 to these bits other than in order to prevent unintentional interrupt clearance.

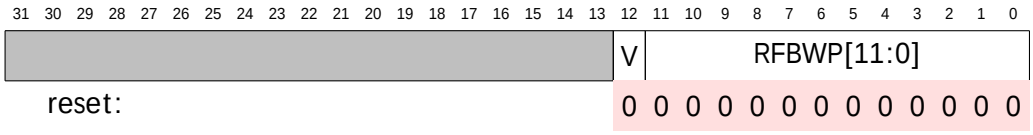
r9: Receive frame buffer read pointer



The functions of these fields are described in the table below:

Name	bits	R/W	Function
V	12	R	Rollover bit - toggles on address wrap-around
RFBRP[11:0]	11:0	R	Receive frame buffer read pointer

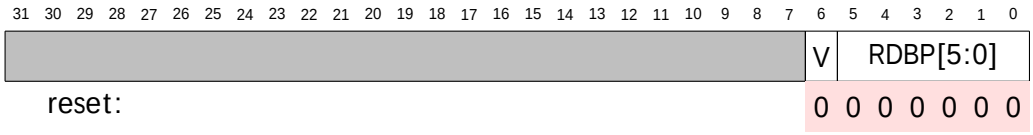
r10: Receive frame buffer write pointer



The functions of these fields are described in the table below:

Name	bits	R/W	Function
V	12	R	Rollover bit - toggles on address wrap-around
RFBWP[11:0]	11:0	R	Receive frame buffer write pointer

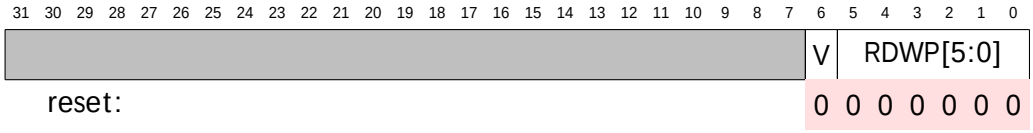
r11: Receive descriptor read pointer



The functions of these fields are described in the table below:

Name	bits	R/W	Function
V	6	R	Rollover bit - toggles on address wrap-around
RFBP[5:0]	5:0	R	Receive descriptor read pointer

r12: Receive descriptor write pointer



The functions of these fields are described in the table below:

Name	bits	R/W	Function
V	6	R	Rollover bit - toggles on address wrap-around
RFWP[5:0]	5:0	R	Receive descriptor write pointer

3.2.15.5 Fault-tolerance

The Ethernet interface will only be used on a small number of nodes; most nodes are insensitive to faults in its functionality as they will not attempt to use it.

3.2.16 Watchdog timer

The watchdog timer is an ARM PrimeCell component (ARM part SP805, documented in ARM DDI 0270B) that is responsible for applying a system reset when a failure condition is detected. Normally, the Monitor Processor will be responsible for resetting the watchdog periodically to indicate that all is well. If the Monitor Processor should crash, or fail to reset the watchdog during a pre-determined period of time, the watchdog will trigger.

3.2.16.1 Features

- generates an interrupt request after a programmable time period;
- causes a chip-level reset if the Monitor Processor does not respond to an interrupt request within a subsequent time period of the same length.

3.2.16.2 Register summary

Base address: 0xe3000000 (buffered write), 0xf3000000 (unbuffered write).

User registers

The following registers allow normal user programming of the Watchdog timer:

Name	Offset	R/W	Function
r0: WdogLoad	0x00	R/W	Count load register
r1: WdogValue	0x04	R	Current count value
r2: WdogControl	0x08	R/W	Control register
r3: WdogIntClr	0x0C	W	Interrupt clear register
r4: WdogRIS	0x10	R	Raw interrupt status register
r5: WdogMIS	0x14	R	Masked interrupt status register
r6: WdogLock	0xC00	R/W	Lock register

Test and ID registers

In addition, there are test and ID registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
WdogITCR	0xF00	R/W	Watchdog integration test control register
WdogITOP	0xF04	W	Watchdog integration test output set register
WdogPeriphID0-3	0xFE0-C	R	Watchdog peripheral ID byte registers
WdogPCID0-3	0xFF0-C	R	Watchdog Prime Cell ID byte registers

See AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003, for further details of the test and ID registers.

3.2.16.3 Register details

r0: Load

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wdog load																															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This read-write register contains the value the from which the counter is to decrement. When this register is written to, the count immediately restarts from the new value. The minimum value is 1.

r1: Count

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Wdog count																															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

This read-only register contains the current value of the decrementing counter. The first time the counter decrements to zero the Watchdog raises an interrupt. If the interrupt is still active the second time the counter decrements to zero the reset output is activated.

r2: Control

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																														E	I
reset:																														0	0

The functions of these fields are described in the table below:

Name	Offset	R/W	Function
E	1	R/W	Enable the Watchdog reset output (1)
I	0	R/W	Enable Watchdog counter and interrupt (1)

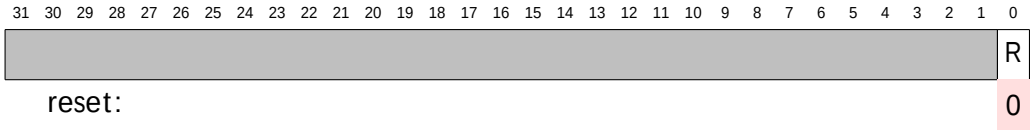
Once the Watchdog has been initialised both enables should be set to '1' for normal watchdog operation.

r3: Interrupt clear

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

A write of any value to this register clears the watchdog interrupt and reloads the counter from r1.

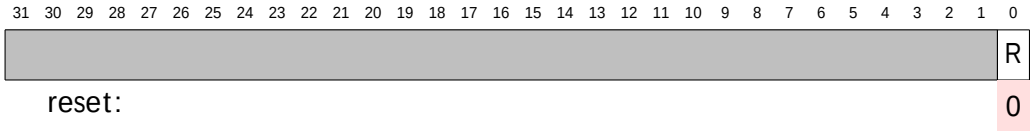
r4: Raw interrupt status



The function of this field is described in the table below:

Name	Offset	R/W	Function
R	0	R	Raw (unmasked) watchdog interrupt

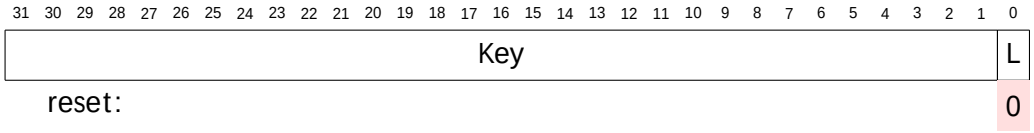
r5: Masked interrupt status



The function of this field is described in the table below:

Name	Offset	R/W	Function
W	0	R	Watchdog interrupt output

r6: Lock



The functions of these fields are described in the table below:

Name	Offset	R/W	Function
Key	31:0	W	Write 0x1ACCE551 to enable writes
L	0	R	Write access enabled (0) or disabled (1)

A read from this register returns only the bottom bit, indicating whether writes to other registers are enabled (0) or disabled (1). A write of 0x1ACCE551 enables write access to the other registers; a write of any other value disables write access to the other registers. Note that the 'Key' field is 32 bits and includes bit 0.

The lock function is available to ensure that the watchdog will not be reset by errant programs.

3.2.17 System RAM

The System RAM is an additional 32 Kbyte block of on-chip RAM used primarily by the Monitor Processor to enhance its program and data memory resources as it will be running more complex (though less time-critical) algorithms than the fascicle processors.

As the choice of Monitor Processor is made at start-up (and may change during run-time for fault- tolerance purposes) the System RAM is made available to whichever processor is Monitor Processor via the System NoC. Accesses by the Monitor Processor to the System RAM are non- blocking as far as SDRAM accesses by the fascicle processors are concerned.

The System RAM may also be used by the fascicle processors to communicate with the Monitor Processor and with each other, should the need arise.

3.2.17.1 Features

- 32 Kbytes of SRAM, available via the System NoC.
- can be used as source of boot code.

3.2.17.2 Address location

Base address: 0xe5000000 (buffered write), 0xf5000000 (unbuffered write). Can also appear at the Boot area at 0xff000000 if the 'Boot area switch' is set in the System Controller.

3.2.17.3 Fault-tolerance

Fault insertion

- It is straightforward to corrupt the contents of the System RAM to model a soft error – any processor can do this. It is not clear how this would be detected.

Fault detection

- The Monitor Processor may perform a System RAM test at start-up, and periodically thereafter.
- It is not clear how soft errors can be detected without some sort of parity or ECC system.

Fault isolation

- Faulty words in the System SRAM can be mapped out of use.

Reconfiguration

- For hard failure of a single bit, avoid using the word containing the failed bit.

- If the System RAM fails completely the only option is to use the SDRAM instead, which will probably result in compromised performance for the fascicle processors due to loss of SDRAM bandwidth. An option then would be to relocate some of the fascicle processors' workload to another chip.

3.2.17.4 Test

Production test

- run standard memory test patterns from one of the processing subsystems.

3.2.18 Boot ROM

3.2.18.1 Features

- a small (32Kbyte) on-chip ROM to provide minimal support for:
- initial self-test, and Monitor Processor selection
- Router initialisation for bootstrapping
- system boot.

The Test chip Boot ROM also supports the loading of code from an external SPI ROM using the GPIO[5:2] pins as an SPI interface.

3.2.18.2 Address location

Base address: 0xf6000000 and, after a hard reset and unless the 'Boot area switch' is set in the Sytem Controller, in the Boot area at 0xff000000.

3.2.18.3 Fault-tolerance

Fault insertion

Switch the 'Boot area switch' to remove the Boot ROM from the reset location.

Fault detection

If the Boot ROM fails the boot process will also fail, which will be detected at start-up.

Fault isolation

Switching the Boot ROM out of the boot area should render it harmless.

Reconfiguration

When the Boot ROM is switched out of the boot area the System RAM is switched into the boot area. A neighbour 'nurse' chip can initialise the System RAM with the boot code and retry initialisation.

3.2.19 JTAG

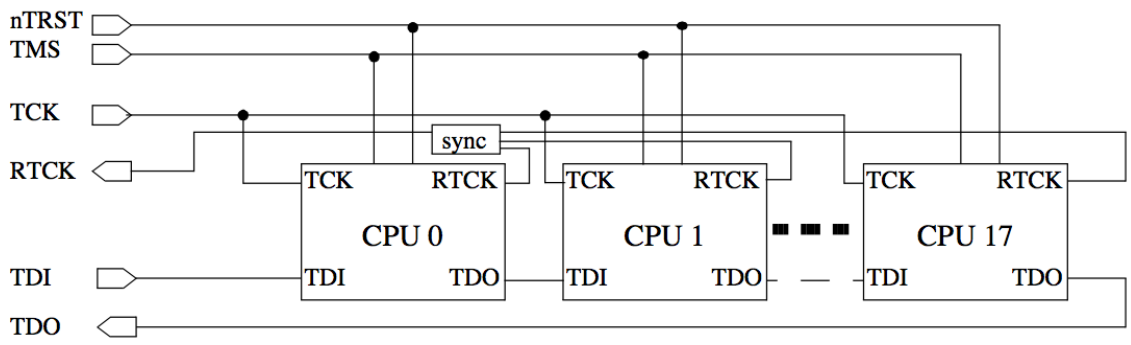
The JTAG IEEE 1149.1 system on the SpiNNaker chip provides access only to the ARM968 processors for software debug purposes. There is no provision for scan access to the SpiNNaker pins or other on-chip features.

3.2.19.1 Features

- standard ARM debug access to all 18 ARM968 processors
- device ID codes of 0x05968477

3.2.19.2 Organisation

The organisation of the ARM968 JTAG access is as shown below:



The ARM968 CPUs synchronize TCK to their respective local clocks, which may be different, so the ARM interface has an addition clock return signal, RTCK, which indicates when a transition on TCK has been recognised. TCK may then make a further transition. The RTCK signal allows TCK to be operated at the maximum safe frequency.

TCK and RTCK should obey a standard handshake protocol, so TCK may only rise when RTCK is low, and TCK may only fall when RTCK is high.

All of the processors are in series on the data scan path (TDI to TDO), with CPU0 coming before CPU1, etc. All processor TAP controllers have JTAG-standard bypass registers to support more efficient access to the other processor.

3.2.19.3 Operation

The JTAG interface supports direct connection of the ARM software development tools to the SpiNNaker test chip, giving those tools standard access to the ARM processors, their local memories, and all system functions visible from those processors.

It is expected that the JTAG interface will be used only with suitable JTAG-aware tools, for hardware debugging (if necessary) and software debugging as required.

3.2.20 Input and Output signals

The SpiNNaker chip has the following IO, power and ground pins. All IO is assumed to operate at 1.8V with CMOS logic levels; the SDRAM interface is 1.8V LVCMOS. All other IOs are non-critical, though output delay affects link throughput.

3.2.20.1 Key

The 'Drive' column in the tables uses the following notation:

Direction	Drive	Meaning
output	*N*mA	maximum drive current *N* mA
output	A/B	slow/fast slew rate
input	S	Schmitt trigger input
input	D/U	pull down/up resistor incorporated

3.2.20.2 SDRAM interface

Signal	Type	Drive	Function	#
DQ[31:0]	IO	8mA B	Data	1-32
A[13:0]	O	4mA B	Address	33-46
CK, CK#	O	8mA B	True and inverse clock	47, 48
CKE	O	4mA B	Clock enable	49
CS#[1:0]	O	4mA B	Chip selects	50, 51
RAS#	O	4mA B	Row address strobe	52
CAS#	O	4mA B	Column address strobe	53
WE#	O	4mA B	Write enable	54
DM[3:0]	O	8mA B	Data mask	55-58
BA[1:0]	O	4mA B	Bank address	59, 60
DQS[3:0]	IO	8mA DB	Data strobe	61-64
Vdd_18[23, 13:0]	1.8V		Power for SDRAM pins	65-79
Vss_18[23, 13:0]	Gnd		Ground for SDRAM pins	80-94

When the package incorporates an internal SDRAM die, all of the above signal pins apart from CS#[1] will be connected to it. They may or may not also be connected to package balls. CS#[1] connects only to a package ball.

3.2.20.3 JTAG

Signal	Type	Drive	Function	#
nTRST	I	SU	Test reset (active low)	95
TCK	I	SD	Test clock	96
RTCK	O	4mA A	Return test clock	97
TMS	I	SU	Test mode select	98
TDI	I	SU	Test data in	99
TDO	O	4mA A	Test data out	100

3.2.20.4 Ethernet MII

Signal	Type	Drive	Function	#
EtherMux	I	SD	select Ethernet or GPIO[23:16]	101
RX_CLK	I	SD	Receive clock	102
RX_D[3:0]	IO	4mA A SU	Receive data/GPIO[19:16]	103-106
RX_DV	I	SD	Receive data valid	107
RX_ERR	I	SD	Receive data error	108
TX_CLK	O	4mA A	Transmit clock	109
TX_D[3:0]	IO	4mA A SD	Transmit data/GPIO[23:20]	110-113
TX_EN	O	4mA A	Transmit data valid	114
TX_ERR	O	4mA A	Force transmit data error	115
MDC	O	4mA A	Management interface clock	116
MDIO	IO	4mA A	Management interface data	117
PHY_RSTn	O	4mA A	PHY reset (optional)	118
PHY_IRQn	I	SD	PHY interrupt (optional)	119
Vdd_18[15]	1.8V		Power for Ethernet MII pins	120
Vss_18[15]	Gnd		Ground for Ethernet MII pins	121

3.2.20.5 Communication links

Signal	Type	Drive	Function	#
L0in[6:0]	I	SD	link 0 2-of-7 input code	122-128
L0inA	O	12mA B	link 0 input acknowledge	129
L0out[6:0]	O	12mA B	link 0 2-of-7 output code	130-136
L0outA	I	SD	link 0 output acknowledge	137
L1in[6:0]	I	SD	link 1 2-of-7 input code	138-144
L1inA	O	12mA B	link 1 input acknowledge	145
L1out[6:0]	O	12mA B	link 1 2-of-7 output code	146-152
L1outA	I	SD	link 1 output acknowledge	153
L2in[6:0]	I	SD	link 2 2-of-7 input code	154-160
L2inA	O	12mA B	link 2 input acknowledge	161
L2out[6:0]	O	12mA B	link 2 2-of-7 output code	162-168
L2outA	I	SD	link 2 output acknowledge	169
L3in[6:0]	I	SD	link 3 2-of-7 input code	170-176
L3inA	O	12mA B	link 3 input acknowledge	177
L3out[6:0]	O	12mA B	link 3 2-of-7 output code	178-184
L3outA	I	SD	link 3 output acknowledge	185
L4in[6:0]	I	SD	link 4 2-of-7 input code	186-192
L4inA	O	12mA B	link 4 input acknowledge	193
L4out[6:0]	O	12mA B	link 4 2-of-7 output code	194-200
L4outA	I	SD	link 4 output acknowledge	201
L5in[6:0]	I	SD	link 5 2-of-7 input code	202-208
L5inA	O	12mA B	link 5 input acknowledge	209
L5out[6:0]	O	12mA B	link 5 2-of-7 output code	210-216
L5outA	I	SD	link 5 output acknowledge	217
Vdd_18[22:21,17:14]	1.8V		Power for link pins	218-223
Vss_18[22:21,17:14]	Gnd		Ground for link pins	224-229

3.2.20.6 Miscellaneous

Signal	Type	Drive	Function	#
GPIO[15:0]	IO	4mA A SD	General-purpose IO	230-245
PORIn	I	SD	Power-on reset	246
ResetIn	I	SD	Chip reset	247
Test	I	SD	Chip test mode	248
Clk10MIn	I	S	Main input clock - 10MHz	249
nClk10MOut	O	4mA A	Daisy-chain 10MHz clock out	250
Clk32kIn	I	S	Slow (global) 32kHz clock	251
Vdd_18[18:17]	1.8V		Power for miscellaneous pins	252-253
Vss_18[18:17]	Gnd		Ground for misc. pins	254-255
Vdd_12[13:0]	1.2V		Power for core logic	256-269
Vss_12[13:0]	Gnd		Ground for core logic	270-283
Vdd_PLL[3:0]	1.2V		Power for PLLs	284-287
Vss_PLL[3:0]	Gnd		Ground for PLLs	288-291
Tres	I	analogue	Temp. sensor analogue input	292
Int[1:0]	I	SD	Exteernal interrupt requests	293-294

3.2.20.7 Internal SDRAM interface

Signal	Type	Drive	Function	#
GPIO[31]	IO	4mA A SD	Connects to SDRAM TQ	293
GPIO[30]	IO	4mA A SD	SDRAM DPD input	294
GPIO[29]	IO	4mA A SD	Bond to Vdd	295

3.2.20.8 Internal SDRAM power & ground

In addition to the signal pins that connect the internal SDRAM to the SpiNNaker chip, the SDRAM also requires 1.8V Vdd and ground connections - 30 in total.

3.2.21 Packaging

The SpiNNaker chip is packaged in a 300LBGA package with 1mm ball pitch. The allocation of signals to balls is as shown below:

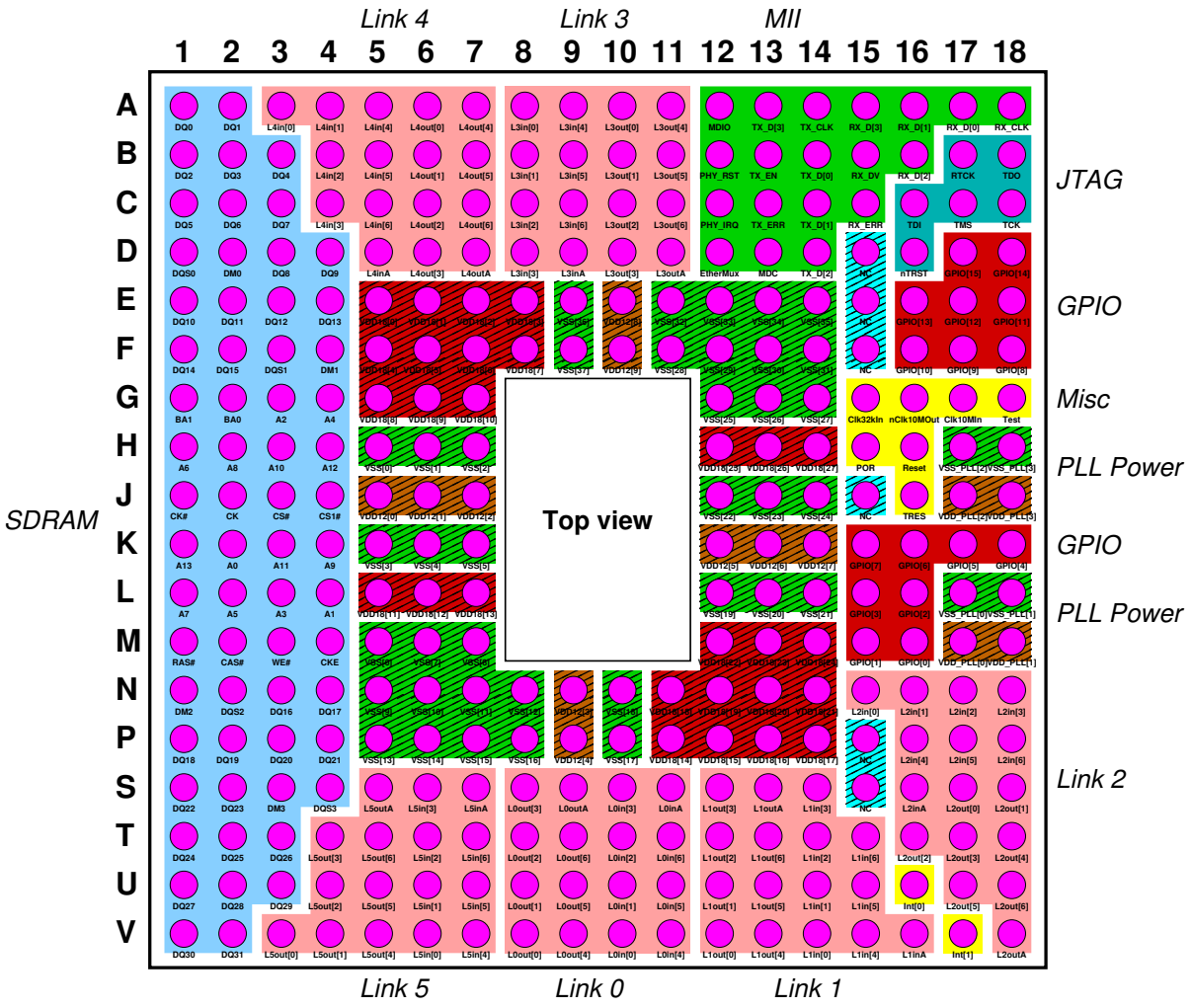


Figure 3.2.1: SpiNNaker 300LBGA Packaging

It is expected that a 128Mbyte Mobile DDR SDRAM will normally be incorporated into the package with the SpiNNaker chip, using wire-bonded Multi-Chip Package (MCP) assembly.

3.2.22 Application notes

3.2.22.1 Firefly synchronization

The local time phase, used for errant packet trapping, can be maintained across the system by a combination of local slightly randomized timers and local phase-locking using nearest-neighbour communication.

Time phase accuracy

If the system time phase is F and the skew is K (that is, all parts of the system transition from one phase to its successor within a time K), then a packet has at least $F - K$ to reach its destination and will be killed after at most $2F + K$. Thus, if we want to allow for a maximum packet transit time of $F - K = T$ and can achieve a minimum phase skew of K, then T and K are both system constants and we should choose $F = T + K$. The longest packet life is then $2T + 3K$.

3.2.22.2 Neuron address space

Neurons ocuppy an address space that identifies each Neuron uniquely within the domain of its multicast routing path (where this domain must include alternative links that may be taken during emergency routing). Where these domains do not overlap it is possible to reuse the same address, though this must be done with considerable care. Neuron addresses can be assigned arbitrarily; this can be exploited to optimize Router utilization (e.g. by giving Neurons with the same routing requirements related addresses so that they can be routed by the same Router entries).

3.3 SpiNNaker Software Datasheet

Background

SpiNNaker was designed at the University of Manchester within an EPSRC-funded project in collaboration with the University of Southampton, ARM Limited and Silistix Limited. Subsequent development took place within a second EPSRC-funded project which added the universities of Cambridge and Sheffield to the collaboration. The work would not have been possible without EPSRC funding, and the support of the EPSRC and the industrial partners is gratefully acknowledged.

Intellectual Property rights

Disclaimer

The details in this design document are presented in good faith but no liability can be accepted for errors or inaccuracies. The design of a complex chip multiprocessor and its associated software is a research activity where there are many uncertainties to be faced, and there is no guarantee that a SpiNNaker system will perform in accordance with the specifications presented here.

The APT group in the School of Computer Science at the University of Manchester was responsible for all of the architectural and logic design of the SpiNNaker chip, with the exception of synthesizable components supplied by ARM Limited and interconnect components supplied by Silistix Limited. All design verification was also carried out by the APT group. As such the industrial project partners bear no responsibility for the correct functioning of the device.

Error notification and feedback

Please email details of any errors, omissions, or suggestions for improvement to Steve Furber <steve.furber@manchester.ac.uk>

3.3.1 Run-time software

The SpiNNaker run-time software involves four different devices:

- The Host, used for application I/O and monitoring.
- Root Monitors (Monitor Processors with direct Ethernet access), used as Monitor Processors and, additionally, to communicate with the host over Ethernet.
- Monitor Processors, used for system-wide inter-processor communication, application support and system monitoring.
- Application Processors (APs), used to run applications.

3.3.1.1 Run-time software stack

Figure 3.3.1 illustrates the run-time software stack in the four devices. The stack is formed by three basic layers with well-defined interfaces between them: Application and monitoring, Run-time support and Hardware device drivers. The two interfaces are the Application Programming Interface (API) and the Hardware Programming Interface (HPI).

To support applications, each of the devices runs a run-time kernel (RTK). The kernel supports the following:

- Application control - the ability to start application execution or terminate gracefully.
- Resources - the ability to use the chip hardware/peripherals in an abstracted way. For example, starting a 1ms timer, setting an entry in the multicast routing table or installing a handler to deal with packet arrival.
- Communication - applications may want to get information either to other APs or to the outside world, for example, Tube-like output or writing files on a host machine.
- Monitoring and debugging - a host running some form of debugger may want to inspect a running application.

These services are available to the applications through the API, described in Section 3.3.2

3.3.1.2 Inter-processor communication

Processor Virtualisation

Each SpiNNaker chip has an address in a SpiNNaker network once a point-to-point (P2P) configuration has been set up during the system boot phase. Each core on the chip has an address - the core ID, which is hardwired. For practical purposes, however, this is not very useful as, viewed from outside, there is no knowledge of which core is the Monitor and which cores are non-functional.

Following the selection of the Monitor Processor, it allocates each working core a "virtual core number". Number zero is assigned to the Monitor Processor (MP) and numbers one onwards to the Application Processors (APs). The major advantage of this is that the core number of the Monitor is always known.

Addressing SpiNNaker Nodes

As SpiNNaker chips are usually connected together in a two-dimensional grid, it's convenient to address them by their (X, Y) coordinate in the grid. This is the basis for the P2P addressing., using a 256 x 256 grid (only partially filled!) where the P2P address is $256 * X + Y$.

Processors on each chip can be addressed using their virtual number as described above, so any processor in a SpiNNaker network can be addressed by the triplet $< X, Y, P >$ (where P is the virtual core number). It's unlikely that the number of cores on a chip will exceed 256 in the near future so three bytes is enough to specify $< X, Y, P >$. This triplet is the basis for a datagram protocol described below to allow SpiNNaker nodes to communicate.

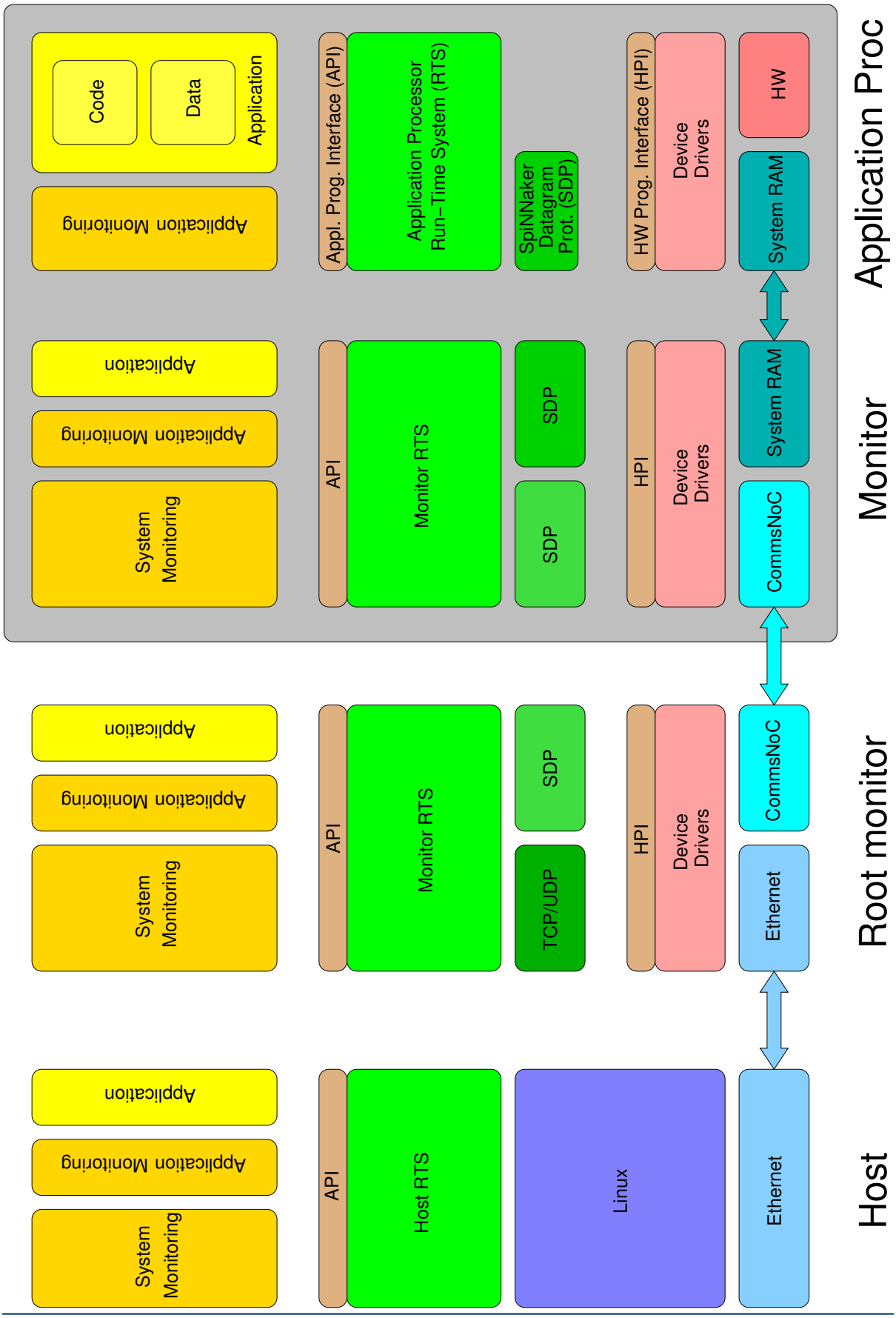


Figure 3.3.1: SpiNNaker run-time software stack.

SpiNNaker Datagram Protocol (SDP)

SDP is an unreliable datagram protocol (similar to Internet UDP). An SDP datagram (or packet) contains some addressing information and an arbitrary amount of data (the size of which is limited by the implementation - currently using 256+16 or 272 bytes). The addressing information consists of 8 bytes. There are two 3-byte triplets as above which specify the source and destination addresses and also a Tag byte and a Flag byte.

The Tag byte allows an SDP packet to be associated with a full Internet address (IP & Port) so that SDP can support communication between any SpiNNaker core and any IP-connected host. The Flag byte is used for a variety of nefarious things which most users won't want to mess with!

There is also a length associated with each SDP packet and a checksum and these are carried in a variety of ways depending on the underlying transport mechanism.

The current implementation of SDP transport layers for SpiNNaker use both P2P packets (for communication between arbitrary chips/cores) and NN packets. The latter allows communication with neighbouring chips if P2P addressing is not set up. SDP can also be carried over Internet UDP and this is the basis for the various bootloaders and debug mechanisms that are currently in use. SDP packets are passed between cores on the same chip by the use of shared memory (e.g., System RAM).

Software support for communication

The Monitor run-time kernel supports inter-processor communication. It receives SDP packets either from other SpiNNaker chips via P2P or NN, the Internet via the Ethernet interface or other cores on the same chip via shared memory. A (software) router is used to send SDP packets to their destination. Those chips which have an Ethernet interface maintain "IPTag" tables to route SDP packets to arbitrary IP addresses based on the Tag byte in the SDP header.

The APs do not perform the SDP packet routing as it's not needed. All cores are able to receive and respond to commands sent to them via SDP. In most cases it will be a host sending commands but, in principle, any core can send commands to any other.

The set of commands provided includes reading and writing memory, and causing the core to start execution at any address. This is enough to get arbitrary applications loaded onto any core and start them running.

3.3.1.3 Runtime memory map

Figure 3.3.2 shows the Application Processors run-time memory map.

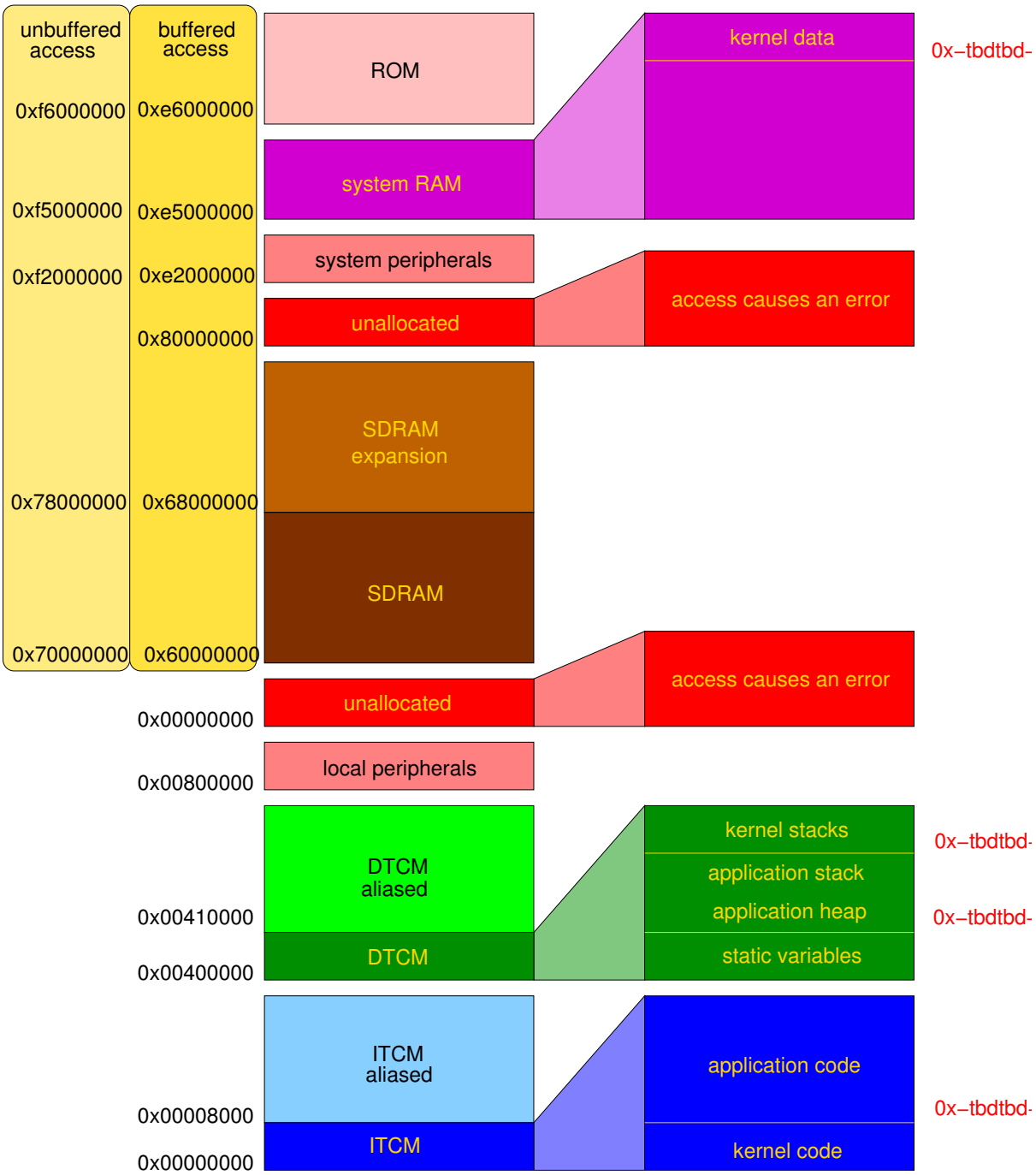


Figure 3.3.2: SpiNNaker run-time memory map.

3.3.2 Application programming interface (API)

3.3.2.1 Event-driven programming model

The SpiNNaker Programming Model (PM) is a simple, event-driven model. Applications do not control execution flow, they can only indicate the functions, referred to as callbacks, to be executed when specific events occur, such as the arrival of a packet, the completion of a Direct Memory Access (DMA) transfer or the lapse of a periodic time interval. An Application Run-time Kernel (ARK) controls the flow of execution and schedules/dispatches application callback functions when appropriate.

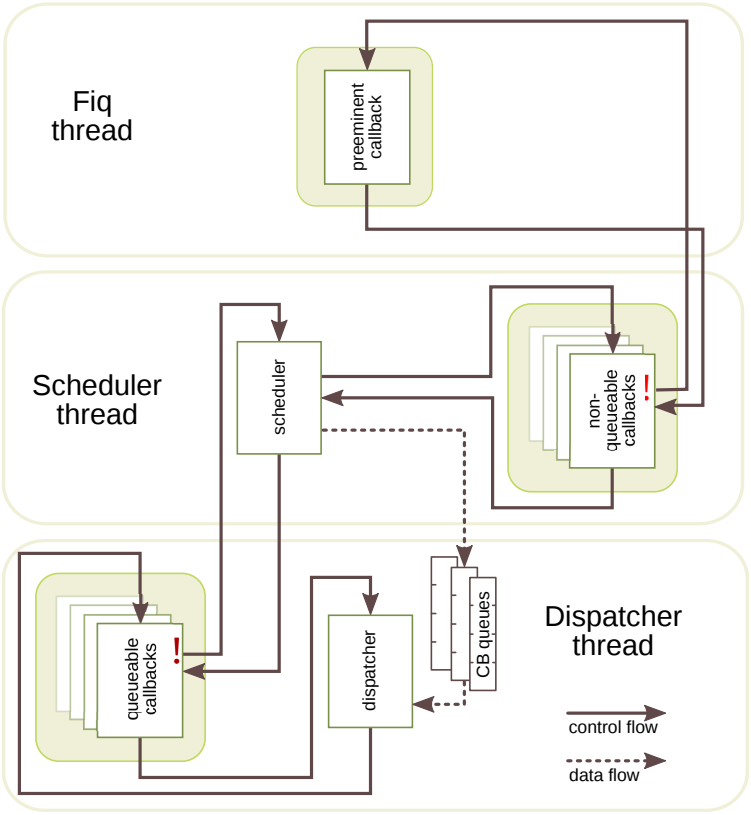


Figure 3.3.3: SpiNNaker event-driven programming framework.

Fig. 3.3.3 shows the basic architecture of the event-driven framework. Application developers write callback routines that are associated with events of interest and register them at a certain priority with the kernel. When the corresponding event occurs the scheduler either executes the callback immediately and atomically (in the case of a non-queueable callback) or places it into a scheduling queue at a position according to its priority (in case of a queueable callback). When control is returned to the dispatcher (following the completion of a callback) the highest-priority queueable callback is executed. Queueable callbacks do not necessarily execute atomically: they may be pre-empted by non-queueable callbacks if a

corresponding event occurs during their execution. The dispatcher goes to sleep (low-power consumption state) if the pending callback queues are empty and will be awakened by an event. Application developers can designate one non-queueable callback as the preeminent callback, which has the highest priority and can pre-empt other non-queueable callbacks as well as all queueable ones.

The preeminent callback is associated with a FIQ interrupt while other non-queueable callbacks are associated with IRQ interrupts. The API provides different functions to disable interrupts: spin1_irq_disable disables IRQs, spin1_fiq_disable disables FIQs while spin1_int_disable disables both FIQs and IRQs. The use of spin1_fiq_disable may lead to priority inversion.

Design considerations

- Non-queueable callbacks are available as a method of pre-empting long running tasks with short, high priority tasks. The allocation of application tasks to non-queueable callbacks must be carefully considered. The selection of the preeminent callback can be particularly important. Long-running operations should not be executed in non-queueable callbacks for fear of starving queueable callbacks.

- Queueable callbacks may require critical sections (i.e., sections that are completed atomically) to prevent pre-emption during access to shared resources. Critical sections may be achieved by disabling interrupts before accessing the shared resource and re-enabling them afterwards. Applications are executed in a privileged mode to allow the callback programmer to insert these critical sections. This approach has the risk that it allows the programmer to modify peripherals –such as the system controller– unchecked.

- Non-queueable callbacks may also require critical sections, as they can be pre-empted by the preeminent callback.

- Events –usually triggered by interrupts– have priority determined by the programming of the Vectored Interrupt Controller (VIC). This allows priority to be determined when multiple events corresponding to different non-queueable callbacks occur concurrently. It also affects the order in which queueable callbacks of the same priority are queued.

3.3.2.2 Programming interface

The following sections introduce the events and functions supported by the API.

Events

The SpiNNaker PM is event-driven: all computation follows from some event. The following events are available to the application:

event	trigger
MC packet received	reception of a multicast packet
DMA transfer done	successful completion of a DMA transfer
Timer tick	passage of specified period of time
SDP packet received	reception of a SpiNNaker Datagram Protocol packet
User event	software-triggered interrupt

In addition, errors can also generate events:

— events not yet supported —	
event	**trigger**
MCP parity error	multicast packet received with wrong parity
MCP framing error	wrongly framed multicast packet received
DMA transfer error	unsuccessful completion of a DMA transfer
DMA transfer timeout	DMA transfer is taking too long

Each of these events is handled by a kernel routine which may schedule or execute an application callback, if one is registered by the application.

Callback arguments

Callbacks are functions with two unsigned integer arguments (which may be NULL) and no return value. The arguments may be cast into the appropriate types by the callback. The arguments provided to callbacks (where 'none' denotes a superfluous argument) by each event are:

event	first argument	second argument
MC packet received	uint key	uint payload
DMA transfer done	uint transfer_ID	uint tag
Timer tick	uint simulation_time	uint none
SDP packet received	uint *mailbox	uint destination_port
User event	uint arg0	uint arg1

Pre-defined constants

logic value	value	keyword
true	(0 == 0)	TRUE
false	(0 != 0)	FALSE

function result	value	keyword
failure	0	FAILURE
success	1	SUCCESS

transfer direction	value	keyword
read (system to TCM)	0	DMA_READ
write (TCM to system)	1	DMA_WRITE

packet payload	value	keyword
no payload	0	NO_PAYLOAD
payload present	1	WITH_PAYLOAD

event	value	keyword
MC packet received	0	MC_PACKET_RECEIVED
DMA transfer done	1	DMA_TRANSFER_DONE
Timer tick	2	TIMER_TICK
SDP packet received	3	SDP_PACKET_RX
User event	4	USER_EVENT

Pre-defined types

type	value	size
uint	unsigned int	32 bits
ushort	unsigned short	16 bits
uchar	unsigned char	8 bits
callback_t	void (*callback_t) (uint, uint)	32 bits
sdp_msg_t	struct (see below)	292 bytes
diagnostics_t	struct (see below)	44 bytes

SDP message structure

```
typedef struct sdp_msg      // SDP message (=292 bytes)
{
  struct sdp_msg *next;     // Next in free list
  ushort length;            // length
  ushort checksum;          // checksum (if used)

  // sdp_hdr_t

  uchar flags;              // SDP flag byte
  uchar tag;                // SDP IPtag
  uchar dest_port;          // SDP destination port
  uchar srce_port;          // SDP source port
  ushort dest_addr;         // SDP destination address
  ushort srce_addr;         // SDP source address

  // cmd_hdr_t (optional)

  ushort cmd_rc;            // Command/Return Code
  ushort seq;               // Sequence number
  uint arg1;                // Arg 1
  uint arg2;                // Arg 2
  uint arg3;                // Arg 3

  // user data (optional)

  uchar data[SDP_BUF_SIZE]; // User data (256 bytes)

  uint _PAD;                // Private padding
} sdp_msg_t;
```

diagnostics variable structure

```
typedef struct
{
  uint exit_code;               // simulation exit code
  uint warnings;                // warnings type bit map
  uint total_mc_packets;        // total routed MC packets during simulation
  uint dumped_mc_packets;       // total dumped MC packets by the router
  uint discarded_mc_packets;    // total discarded MC packets by API
  uint dma_transfers;           // total DMA transfers requested
  uint dma_bursts;              // total DMA bursts completed
  uint dma_queue_full;          // dma queue full count
  uint task_queue_full;         // task queue full count
  uint tx_packet_queue_full;    // transmitter packet queue full count
  uint writeBack_errors;        // write-back buffer errror count
} diagnostics_t;
```

Pre-declared variables

variable	type	function
leadAp	**uchar**	TRUE if appointed chip-wise application leader
diagnostics	**diagnostics_t**	returns diagnostic information (if turned on in compilation)

Kernel services

The kernel provides a number of services to the application programmer:

Simulation control functions

Start simulation		
function	arguments	description
uint spin1_start	void	no arguments
returns:	EXIT_CODE (0 = NO ERRORS)	
notes:	• transfers control from the application to the ARK. • use spin1_kill to indicate a non-zero EXIT_CODE.	

Stop simulation		
function	arguments	description
void spin1_stop	void	no arguments
returns:	no return value	
notes:	• transfers control from the ARK back to the application.	

Stop simulation and report error		
function	arguments	description
void spin1_kill	uint error	error code to report
returns:	no return value	
notes:	• transfers control from the ARK back to the application. • The argument is used as the return value for spin1_start.	

Set the timer tick period		
function	arguments	description
void spin1_set_timer_tick	uint period	timer tick period (in microseconds)
returns:	no return value	

Request simulation time		
function	arguments	description
uint spin1_get_simulation_time	void	no arguments
returns:	timer ticks since the start of simulation.	

DEPRECATED!	Indicate which cores are involved in the simulation	
function	arguments	description
void spin1_set_core_map	uint chips	number of chips
	uint * core_map	bit map array of cores
returns:	no return value	
notes:	• sets the map of the cores that need to synchronise to start the simulation. • the numbers of chips & cores default to 1, thus no synchronisation is attempted.	

Core Map Examples

```
// chips are identified using Cartesian coordinates.
// Note that the core map is a uni-dimensional array but
// describes a bi-dimensional array of chips in x-major format
// i.e., the order is (0, 0), (0, 1), ... , (1, 0), (1, 1), ...

// 2 x 2 core map on SpiNN-2, SpiNN-3 and SpiNN-4 boards - 2 cores on each chip
uint const NUMBER_OF_CHIPS = 4;    // virtual 2 x 2 array of chips
uint core_map[NUMBER_OF_CHIPS] =
{
  0x6,      0x6,     // (0, 0),  (0, 1)
  0x6,      0x6      // (1, 0),  (1, 1)
};

// "hexagonal" 8 x 8 core map on SpiNN-4 board - 16 cores on each chip
uint const NUMBER_OF_CHIPS = 64;     // virtual 8 x 8 array of chips
uint core_map[NUMBER_OF_CHIPS] =
{
  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,        0,        0,        0,
  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,        0,        0,
  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,        0,
  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,
  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,
  0,        0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,
  0,        0,        0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,
  0,        0,        0,        0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe
};

// "notched" 5 x 5 core map on SpiNN-4 board - variable number of cores
uint const NUMBER_OF_CHIPS = 64;     // virtual 8 x 8 array of chips
uint core_map[NUMBER_OF_CHIPS] =
{
  6,        6,        2,        2,        0,        0,        0,        0,
  6,        6,        2,        2,        2,        0,        0,        0,
  6,        6,        2,        2,        2,        0,        0,        0,
  2,        2,        6,        2,        2,        0,        0,        0,
  2,        2,        2,        2,        2,        0,        0,        0,
  0,        0,        0,        0,        0,        0,        0,        0,
  0,        0,        0,        0,        0,        0,        0,        0,
  0,        0,        0,        0,        0,        0,        0,        0
};

// NOTE: core maps with "holes" may not synchronise in the current version.
// INCORRECT 8 x 8 core map on SpiNN-4 board - 7 cores on each chip
uint const NUMBER_OF_CHIPS = 64;     // virtual 8 x 8 array of chips
uint core_map[NUMBER_OF_CHIPS] =
{
  0xfe,     0xfe,     0xfe,     0xfe,     0,        0,        0,        0,
  0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0,        0,        0,
  0,        0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0,        0,
  0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0,
  0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,
  0,        0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,
  0,        0,        0xfe,     0xfe,     0xfe,     0xfe,     0xfe,     0xfe,
  0,        0,        0,        0xfe,     0xfe,     0xfe,     0xfe,     0xfe
};
```

		Indicate which cores are involved in the simulation
function	arguments	description
void spin1_application_core_map	uint xchips	map x dimension
	uint xchips	map y dimension
	uint * core_map	bit map array of cores
	returns:	no return value

notes:
- sets the map of the cores that need to synchronise to start the simulation.
- the numbers of chips & cores default to 1, thus no synchronisation is attempted.

Core Map Examples

```
// chips are identified using Cartesian coordinates.

// 2 x 2 core map on SpiNN-2, SpiNN-3 and SpiNN-4 boards - 2 cores on each chip
uint const NUMBER_OF_XCHIPS = 2;    // virtual 2 x 2 array of chips
uint const NUMBER_OF_YCHIPS = 2;
uint core_map[NUMBER_OF_XCHIPS][NUMBER_OF_YCHIPS] =
{
  {0x6,     0x6},   // (0, 0),  (0, 1)
  {0x6,     0x6}    // (1, 0),  (1, 1)
};

// "hexagonal" 8 x 8 core map on SpiNN-4 board - 16 cores on each chip
uint const NUMBER_OF_XCHIPS = 8;    // virtual 8 x 8 array of chips
uint const NUMBER_OF_YCHIPS = 8;
uint core_map[NUMBER_OF_XCHIPS][NUMBER_OF_YCHIPS] =
{
  {0x1fffe, 0x1fffe,  0x1fffe,  0x1fffe,  0,        0,        0,        0},
  {0x1fffe, 0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,        0,        0},
  {0x1fffe, 0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0,        0},
  {0x1fffe, 0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0},
  {0x1fffe, 0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe},
  {0,       0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe},
  {0,       0,        0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe},
  {0,       0,        0,        0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe,  0x1fffe}
};

// "notched" 4 x 5 core map on SpiNN-4 board - variable number of cores
uint const NUMBER_OF_XCHIPS = 4;    // virtual 4 x 5 array of chips
uint const NUMBER_OF_XCHIPS = 5;
uint core_map[NUMBER_OF_XCHIPS][NUMBER_OF_YCHIPS] =
{
  {6,       6,        2,        2,        0},
  {6,       6,        2,        2,        2},
  {6,       6,        2,        2,        2},
  {2,       2,        2,        2,        2}
};

// NOTE: core maps with "holes" may not synchronise in the current version.
// INCORRECT 6 x 7 core map on SpiNN-4 board - 7 cores on each chip
uint const NUMBER_OF_XCHIPS = 6;    // virtual 6 x 7 array of chips
uint const NUMBER_OF_YCHIPS = 7;
uint core_map[NUMBER_OF_XCHIPS][NUMBER_OF_YCHIPS] =
{
  {0xfe,    0xfe,    0xfe,    0xfe,    0,       0,       0},
  {0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0,       0},
  {0,       0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0},
  {0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0xfe},
  {0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0xfe},
  {0,       0xfe,    0xfe,    0xfe,    0xfe,    0xfe,    0xfe}
};
```

Event management functions

Register **callback** to be executed when **event_id** occurs		
function	arguments	description
void spin1_callback_on	uint event_id	event that triggers callback
	callback_t callback	callback function pointer
	uint priority	priority <0 denotes preeminent priority 0 denotes non-queueable priorities >0 denote queueable
returns:	no return value	
notes:	• a callback registration overrides any previous ones for the same event. • only one callback can be registered as preeminent. • a second preeminent registration is demoted to non-queueable.	

Deregister **callback** from **event_id**		
function	arguments	description
void spin1_callback_off	uint event_id	event that triggers callback
returns:	no return value	

Schedule a **callback** for execution with given **priority**		
function	arguments	description
uint spin1_schedule_callback	callback_t callback	callback function pointer
	uint arg0	callback argument
	uint arg1	callback argument
	uint priority	callback priority
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• this function allows the application to schedule a callback without an event. • priority <= 0 must not be used (unpredictable results). • function arguments are not validated.	

Trigger a **user event**		
function	arguments	description
uint spin1_trigger_user_event	uint arg0	callback argument
	uint arg1	callback argument
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• FAILURE indicates a trigger attempt before a previous one has been serviced. • arg0 and arg1 will be passed as arguments to the registered callback. • function arguments are not validated.	

Data transfer functions

		Request a DMA transfer
function	arguments	description
uint spin1_dma_transfer	uint tag	for application use
	void *system_address	address in system NoC
	void *tcm_address	address in TCM
	uint direction	DMA_READ / DMA_WRITE
	uint length	transfer length (in bytes)
returns:	unique transfer identification number (TID)	
notes:	• completion of the transfer generates a DMA transfer done event. • a registered callback can use TID and tag to identify the completed request. • DMA transfers are completed in the order in which they are requested. • TID = FAILURE (= 0) indicates failure to schedule the transfer. • function arguments are not validated. • may cause DMA error or DMA timeout events.	

		Copy a block of memory
function	arguments	description
void spin1_memcpy	void *dst	destination address
	void const *src	source address
	uint len	transfer length (in bytes)
returns:	no return value	
notes:	• function arguments are not validated. • may cause a data abort.	

Communications functions

		Send a multicast packet
function	arguments	description
uint spin1_send_mc_packet	uint key	packet key
	uint data	packet payload
	uint load	1 = payload present / 0 = no payload
returns:	SUCCESS (=1) / FAILURE (=0)	

		Flush software outgoing multicast packet queue
function	arguments	description
uint spin1_flush_tx_packet_queue	void	no arguments
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• queued packets are thrown away (not sent).	

		Flush software incoming multicast packet queue
function	arguments	description
uint spin1_flush_rx_packet_queue	void	no arguments
returns:	SUCCESS (=1) / FAILURE (=0)	
notes:	• queued packets are thrown away.	

SpiNNaker Datagram Protocol (SDP)

		Send an SDP message
function	arguments	description
uint spin1_send_sdp_msg	sdp_msg_t * msg	pointer to message
	uint timeout	transmission timeout
returns:	SUCCESS (=1) / FAILURE (=0)	

		Request a free SDP message container
function	arguments	description
sdp_msg_t * spin1_msg_get	void	no arguments
returns:	pointer to message (NULL if unsuccessful)	

		Free an SDP message container
function	arguments	description
void spin1_msg_free	sdp_msg_t *msg	pointer to message
returns:	no return value	

Critical section support functions

Disable IRQ interrupts		
function	arguments	description
uint spin1_irq_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Disable FIQ interrupts		
function	arguments	description
uint spin1_fiq_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Disable ALL interrupts		
function	arguments	description
uint spin1_int_disable	void	no arguments
returns:	contents of CPSR before interrupt flags altered.	

Restore core mode and interrupt state		
function	arguments	description
void spin1_mode_restore	uint status	CPSR state to be restored
returns:	no return value.	

System resources access functions

Get core ID		
function	arguments	description
uint spin1_get_core_id	void	no arguments
returns:	core ID in bits [4:0].	

Get chip ID		
function	arguments	description
uint spin1_get_chip_id	void	no arguments
returns:	chip ID in bits [15:0].	
notes:	• chip ID contains x coordinate in bits [15:8], y coordinate in bits [7:0].	

Get ID		
function	arguments	description
uint spin1_get_id	void	no arguments
returns:	chip ID in bits [20:5] / core ID in bits [4:0].	

Control state of board LEDs		
function	arguments	description
void spin1_led_control	uint p	new state for board LEDs
returns:	no return value.	
notes:	• the number of LEDs and their colour varies according to board version. • to turn LEDs 0 and 1 on: spin1_led_control (LED_ON (0) + LED_ON (1)) • to invert LED 2: spin1_led_control (LED_INV (2)) • to turn LED 0 off: spin1_led_control (LED_OFF (0))	

Set up a multicast routing table entry		
function	arguments	description
uint spin1_set_mc_table_entry	uint entry	table entry
	uint key	entry routing key field
	uint mask	entry mask field
	uint route	entry route field
returns:	SUCCESS (=1) / FAILURE (=0).	
notes:	• see SpiNNaker datasheet for details of the MC table operation. • entries 0 to 999 are available to the application. • routing keys with **bit[15] = 1 and bit[10] = 0** are reserved. • function arguments are not validated.	

Memory allocation

		Allocate a new block of DTCM
function	arguments	description
void * spin1_malloc	uint bytes	size of the memory block in bytes
	returns:	pointer to the new memory block.
notes:	• memory blocks are word-aligned. • memory is allocated in DTCM. • there is no support for freeing a memory block.	

Miscellaneous

		Wait for a given time
function	arguments	description
void spin1_delay_us	uint time	wait time (in microseconds)
returns:	no return value	
notes:	• the function busy waits for the given time (in microseconds). • prevents any queueable callbacks from executing (use with care).	

		Generate a 32-bit pseudo-random number
function	arguments	description
void spin1_rand	void	no arguments
returns:	32-bit pseudo-random number	
notes:	• Function based on example function in: • "Programming Techniques", ARM document ARM DUI 0021A. • Uses a 33-bit shift register with exclusive-or feedback taps at bits 33 and 20.	

		Provide a seed to the pseudo-random number generator
function	arguments	description
void spin1_srand	uint seed	32-bit seed
returns:	no return value	

Application Programme Structure

In general, an application programme contains three basic sections:

- **Application Functions**: General application functions to support the callbacks.
- **Application Callbacks**: Functions to be associated with run-time events.
- **Application Main Function**: Variable initialisation, callback registration and transfer of control to main loop.

The structure of a simple application programme is shown on the next page. Many details are left out for brevity.

```
// declare application types and variables
neuron_state state[1000];
spike_bin bins[1000][16];
. . .

/* ------------------------------------------------------------------ */
/* ------------------------ application functions -------------------- */
/* ------------------------------------------------------------------ */
void izhikevich_update(neuron_state *state){
    . . .
    spin1_send_mc_packet(key, 0, NO_PAYLOAD);
    . . .
}

syn_row_addr lookup_synapse_row(neuron_key key)
{
    . . .
}

void bin_spike(neuron_key key, axn_delay delay, syn_weigth weight)
{
    . . .
}

/* ------------------------------------------------------------------ */
/* ------------------------ application callbacks -------------------- */
/* ------------------------------------------------------------------ */
void update_neurons()
{
    . . .
    if (spin1_get_simulation_time() > 1000) // simulation time in "ticks"
        spin1_stop();
    else
        for (i=0; i < 1000; i++) izhikevich_update(state[i]);
    . . .
}

void process_spike(uint key, uint payload)
{
    . . .
    row_addr = lookup_synapses(key);
    tid = spin1_dma_transfer(tag, row_addr, syn_buffer, READ, row_len);
    . . .
}

void schedule_spike()
{
    . . .
    bin_spike(key, delay, weight);
    . . .
}

/* ------------------------------------------------------------------ */
/* ------------------------ application main ------------------------- */
/* ------------------------------------------------------------------ */
void c_main()
{
    //  initialise variables and timer tick
    . . .
    spin1_set_timer_tick(1000); // timer tick period in microseconds
    . . .
    // register callbacks
    spin1_callback_on(TIMER_TICK, update_neurons, 1);
```

```
    spin1_callback_on(MC_PACKET_RECEIVED, process_spike, 0);
    spin1_callback_on(DMA_TRANSFER_DONE, schedule_spike, 0);
    . . .
    // transfer control to the run-time kernel
    spin1_start();
    // control returns here on execution of spin1_stop()
}
```

3.3.3 Neural net simulation frameworks

3.3.3.1 Spiking Neural net simulation framework

SpiNNaker applications are event-driven (figure 3.3.4) in that all computational tasks follow from events in hardware. Neuron states are computed in discrete timesteps initiated in each processor by a local periodic timer event. At each timestep processors evaluate the membrane potentials of all of their neurons given prior synaptic inputs and deliver a packet to the router for each neuron that spikes. Spike packets are routed to all processors that model neurons efferent to the spiking neuron. Receipt raises a packet event that prompts the efferent processor to retrieve the appropriate synaptic weights from off-chip RAM using a background Direct Memory Access transfer. The processor is then free to perform other computations during the DMA transfer and is notified of its completion by a DMA done event that prompts calculation of the sizes of synaptic inputs to subsequent membrane potential evaluations.

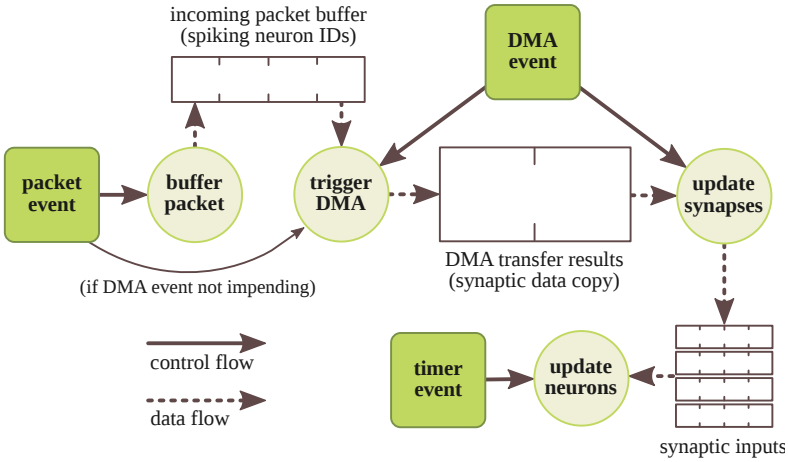


Figure 3.3.4: Events and corresponding tasks in a typical neural simulation.

Each SpiNNaker processor executes an instance of the Application Run-Time Kernel (ARK) which is responsible for providing computational resources to the tasks arising from events. The ARK has two threads of execution (figure 3.3.5) that share processor time: following events, control of the processor is given to the scheduler thread that queues tasks; upon its completion, the scheduler returns control to the dispatcher thread that dequeues tasks and executes them. In terms of figure 3.3.4, for example, a timer event schedules a neuron update task that is dispatched upon returning from the event.

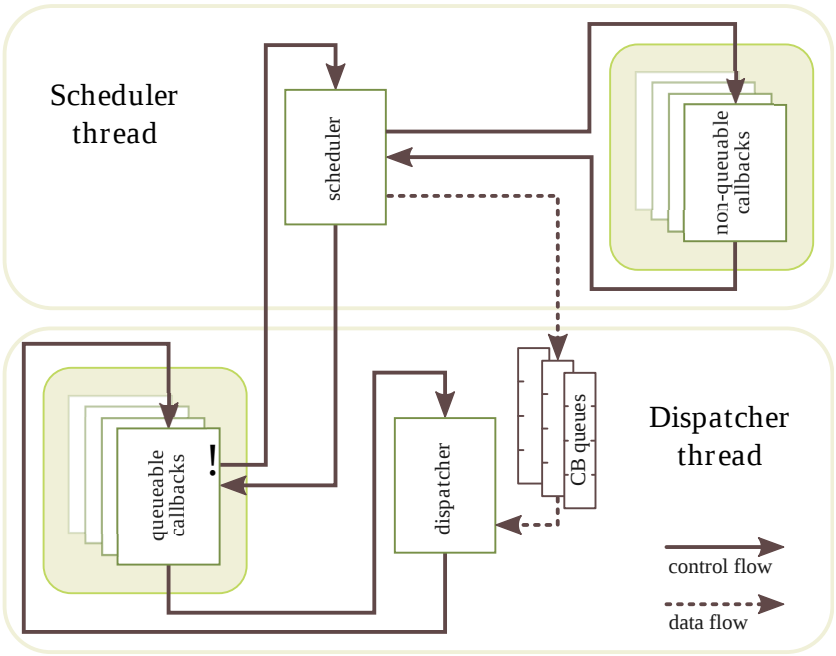


Figure 3.3.5: Control and data flow between the scheduler and dispatcher threads.

Tasks have priorities that dictate the order in which they are executed by the dispatcher. The scheduler places each task at the end of the queue corresponding to its priority and the dispatcher continually executes tasks from the highest-priority non-empty queue. To facilitate immediate execution, priority zero tasks are non-queueable and are executed by the scheduler directly, precluding any further scheduling or dispatching until the task is complete.

The SpiNNaker Application Programming Interface (API) allows a user to specify the tasks that are executed following an event. The user writes callback functions in C that encode the desired task and then registers them with the scheduler against a given event. The following example lists callbacks to compute the Izhikevich equations on the timer event, to buffer packets and kickstart DMA transfers on a packet event and to start subsequent DMA transfers (conditional on receipt of further packets) and process synaptic inputs on the DMA done event. In the `main` function the timer, packet and DMA done callbacks are registered.

```
int main() {
  // Call hardware and simulation configuration functions
  ...
  // Register callbacks and run simulation
  callback_on(PACKET_EVENT, packet_callback, PRIORITY_1);
  callback_on(DMA_DONE_EVENT, dma_done_callback, PRIORITY_2);
  callback_on(TIMER_EVENT, timer_callback_0, PRIORITY_3);
  start(800);
}

void feed_dma_pipeline() {
  // Start engine if idle and transfers pending
  if(!dma_busy() && !dma_queue_empty()) {
    void *source = lookup_synapses(packet_queue_get());
```

```
    dma_transfer(..., source, ...);
  }
}

void buffer_post_synaptic_potentials(synapse_row_t *synapse_row) {
  for(uint i = 0; i < synapse_row_length; i++) {
    // Get neuron ID, connection delay and weight for each synapse
    ...
    // Store synaptic inputs
    neuron[neuron_id].epsp[connection_delay] += synaptic_weight;
  }
}

void dma_done_callback(uint synapse_row, uint unused) {
  // Restart DMA engine if transfers pending
  feed_dma_pipeline();
  // Deliver synaptic inputs to neurons
  buffer_post_synaptic_potentials((synapse_row_t *) synapse_row);
}

void packet_callback(uint key, uint payload) {
  // Queue DMA transfer and start engine if idle
  packet_queue_put(key);
  feed_dma_pipeline();
}

void timer_callback_0(uint time, uint null) {
  for(int i = 0; i < num_neurons; i++) {
    uint current = neuron[i].epsp[time];
    // Compute neuron state given input and deliver spikes.
    // See Jin et al. "Efficient modelling of spiking neural networks"
    ...
    if(neuron[i].v > THRESHOLD){
      send_mc_packet(neuron[i].id);
    }
  }
}
```

3.3.3.2 MLP simulation framework

The Mulitilayer Perceptron (MLP) is a type of non-spiking computational neural network model. An MLP network arranges neurons in layers, each layer having no (or little) internal connectivity but usually strongly connected to other layers. Neurons themselves perform a simple, abstract operation:

$$O_j = T_j(\sum_i O_i w_{ij})$$

where $T_j(x)$ is a range-limited nonlinear transfer function, the most common being the sigmoid:

$$\frac{1}{1+e^{-kx}}$$

Indices i and j refer to the sending "presynaptic" neuron and the receiving "postsynaptic" neuron respectively. Such networks use a supervised learing method to adapt their weights (the w_{ij} terms; overwhelmingly the most popular is the backpropagation algorithm:

$$\Delta w_{ij} = \eta \delta_j O_i \quad (3.3.1)$$

$$\delta_j = \begin{cases} (C_j - O_j)\frac{dT_j}{dS_j} & \text{if j is an output layer} \\ \frac{dT_j}{dS_j}\sum_k \delta_j w_{jk} & \text{if j is not an output layer} \end{cases} \quad (3.3.2)$$

Here S_j refers to the neuron's summation: $\sum_i O_i w_{ij}$ and η is a constant, called the **learning rate**. C_j is the intended output of a neuron; what the neuron "should" have output if the network had been fully trained.

To promote an efficient on-chip mapping, the MLP implementation splits the processing of a neuron into 3 stages, each a separate process optimally residing on a separate core. These stages are:

Weight: This performs the input synaptic multiplication: $O_i w_{ij}$.

Sum: This performs the summation of synaptic inputs: $\sum_i O_i w_{ij}$.

Threshold: This computes the output nonlinearity: $T_j(S_j)$.

A fourth processing stage: **Input**, performs 2 roles: in the forward direction it supplies inputs to the network; in the backward direction, it computes the output errors (the $C_j - O_j$ terms above).

Weight processors each contain a square submatrix of inputs to a block of neurons in 2 layers: $M_{I_x J_y} = m_{ij}\,|_{i_{nx}:i_{n(x+1)};j_{ny}:j_{n(y+1)}}$. The complete architecture is a bidirectional compute-and-forward algorithm:fig. 3.3.6 For the *test chip* the architecture of necessity combines parts of the processing onto the same core: Weight and Sum processes lie on one, while Input and Threshold lie on the other.

The MLP is designed to implement the Lens simulator on SpiNNaker. For the current version, the implementation supports a limited subset of Lens constructs. In particular, it supports the following objects and parameters:

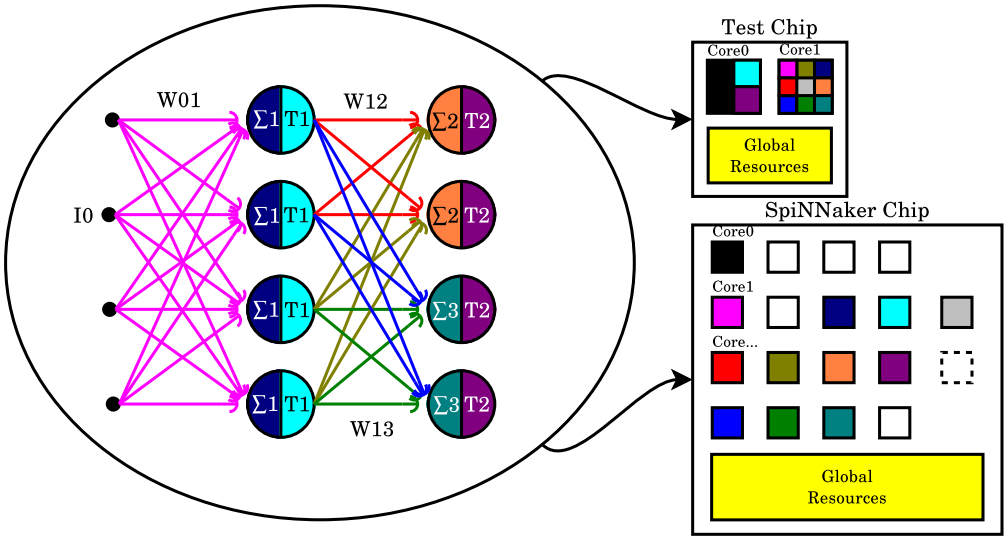


Figure 3.3.6: MLP network mapping.

object	supported properties
Algorithm	Steepest, Momentum, DougsMomentum
Net	Standard, Continuous
Group	Input, Output, Bias; STANDARD_CRIT; BIASED; WRITE_OUTPUTS
Input	Dot_Product, Product; IN_INTEGR, IN_NORM, IN_NOISE, IN_DERIV_NOISE
Output	Linear, Logistic, Ternary, Tanh, Exponential; HARD_CLAMP; OUT_INTEGR, OUT_NOISE, OUT_DERIV_NOISE, OUT_CROPPED
Error	Sum_Squared, Cross_Entropy, Divergence
Time	TimeIntervals, TicksPerInterval, HistoryLength
Training	NumUpdates, BatchSize, Criterion, TrainGroupCrit, TestGroupCrit, GroupCritRequired, MinCritBatches, LearningRate, WeightDecay
Simulation	Gain, TernaryShift, RandMean, RandRange, NoiseRange

Processing under the MLP model remains event-driven. In its basic form each processor in the MLP responds to a single hardware event (packet-received) and schedules software-generated events to complete processing. The packet-received event performs only 2 tasks: 1) it places the packet into an internal service queue; 2) it schedules a deferred event to dequeue and process the packet. The dequeue software event, having retrieved the packet, peforms the address decode and data processing required, as per each stage.

Each subcomponent of the output vector for a given processor may depend on the arrival of a different set of inputs. Thus there can be several output computations awaiting a given

input packet.

3.3.4 Neural net simulation development route

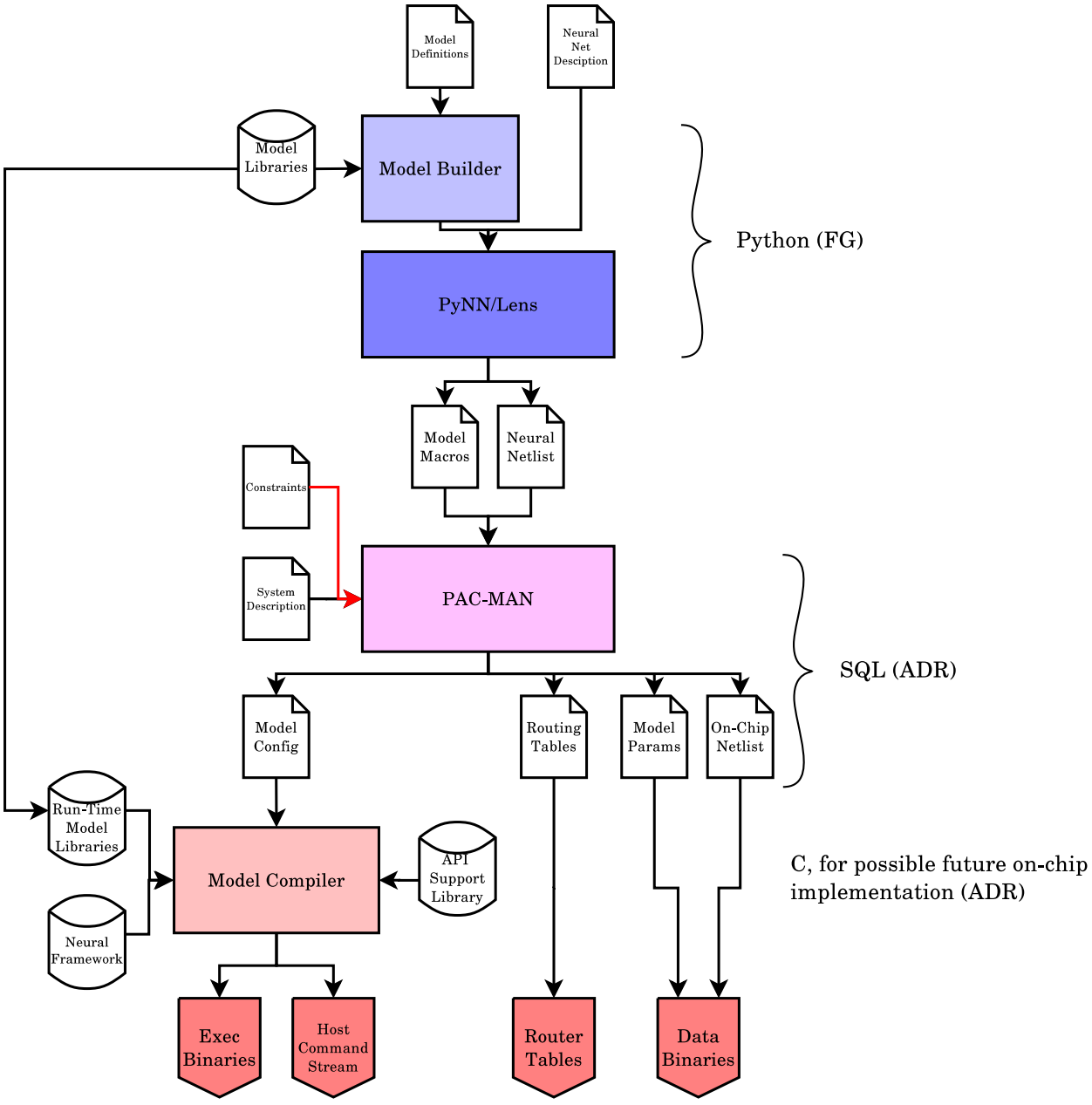


Figure 3.3.7: SpiNNaker neural net simulation development route.

3.3.4.1 pyNN.spiNNaker

PyNN is a standard description language for simulating networks of spiking neurons written in Python. The script is written accordingly to PyNN API and can be executed on the supported software/hardware simulator.

It aims to support modelling at a high-level of abstraction: Populations of neurons and Projections between them.

Objects in PyNN include:

- **Population:** is a group of neurons which share the same model and parameters (eg. Izhikevich Regular Spiking neurons), even if some model dependent initialization values can be randomized.
- **Projection:** represents the connections between two Populations. Describes the type of Connector (All To All, One To One, Random, From List), the target synapse and the connection parameters (weight and delay). It is possible to associate plasticity mechanism to Projections.
- **Input Sources:** they are divided into Spike Sources and Current Sources. Spike Sources are "dummy" neuron populations that produce spikes accordingly to a probability distribution function. Current sources inject currents into the target neurons which vary arbitrarily with time.
- **Recorder:** represent the selection of observables that will be saved eg. spikes, state variables.

The pyNN.spiNNaker module will compile the PyNN script into a list of populations, projections and associated plasticity algorithms, configure inputs and observables.

A **Population** object can be constructed in PyNN as

constructor	arguments	description
Population	uint population_id	a unique identifier for a Population
	uint size	Number of neurons in the Population
	cell_type	Neural Model (`cell_type` in PyNN). It corresponds to the neural application.
	dict parameters	Parameters for the neurons in the Population.
returns:	PyNN Population object Adds a Population to the netlist	

notes:

- Assemblies in PyNN are formed by adding two or more Populations together. They don't need to be explicitly modeled by the pyNN.spiNNaker module since it will reason at a Population level.
- PopulationViews are PyNN objects used to define and operate on subsets of Population objects. In order to deal with them properly the pyNN.spiNNaker plugin will divide them into two distinct Populations.
- The compiler will select the appropriate parsing accordingly to the neural model application selected.

A **Projection** object can be represented as:

constructor	arguments	description
Projection	uint projection_id	a unique identifier for a Projection
	uint presynaptic_population_id	identifies the presynaptic Population
	uint postsynaptic_population_id	identifies the postsynaptic Population
	string target	target synapse/receptor/effector of the Projection (eg. excitatory, NMDA)
	connector_type	describes the connection pattern between two Populations (see next section)
	dict parameters	Parameters for the Projection. Standard parameters are weight and delay
	dict plasticity	Parameters for the Plasticity Algorithm(s) associated with the projection.
returns:	PyNN Projection object Adds a Projection to the netlist	
notes:	• The target will be translated into an ID that will help the application to select the right branch upon a DMA complete. It will then need to be written in the synaptic word in SDRAM • Plasticity Algorithm is a dictionary containing the parameters for the Plasticity algorithm. The dictionary will have a standard entry `type` helping the partitioner and the compiler to identify and correctly position the population and compute plasticity data structures	

Connectors describe the connectivity pattern between two Populations and can be differentiated in:

constructor	arguments	description
OneToOne	weights	a value or a random process to initialize weights
	delays	a value or a random process to initialize delays
	bool allow_self_connections	allows a neuron to connect to another neuron with the same local ID (eg. neurons 0 of both source and destination)

notes:	connects the first neuron of the presynaptic Population to the first neuron of the postsynaptic Population and so on. If the source and destination population don't have the same number of neurons exceeding connections will be discarded

constructor	arguments	description
AllToAll	weights	a value or a random process to initialize weights
	delays	a value or a random process to initialize delays
	bool allow_self_connections	allows a neuron to connect to another neuron with the same local ID (eg. neurons 0 of both source and destination)

notes:	connects all the neurons of the presynaptic Population to all the neurons of the Postsynaptic Population

constructor	arguments	description
FixedProbability	weights	a value or a random process to initialize weights
	delays	a value or a random process to initialize delays
	bool allow_self_connections	allows a neuron to connect to another neuron with the same local ID (eg. neurons 0 of both source and destination)
	float p	probability of a neuron in the presynaptic Population to connect to a neuron in the postsynaptic Population

notes:	connects all the neurons of the source Population every neuron of the postsynaptic Population with probability p

constructor	arguments	description
FromList	list	a python list containing the connection specified one by one

notes:	takes an explicit list of connections in the format `source_id, target_id, params`. The source and target id will be represented relatively to the Population and the list will be contained in the Parameters section p

Current Sources can be thought as:

- fixed currents known a priori: In this case a table describing the changes in time of current amplitude for every input neurons must be generated and loaded
- dynamic currents arbitrarily varying with time: a state variable representing the input current for the neuron is changed

notes:
- currents can vary upon receival of an event (MC packet with particular target, Message from Host)
- Static current table can be loaded in the monitor/dedicated processor and have a process that leads to the change of the state variable in the target neuron/core
- In any case the partitioner/compiler needs to know which neurons can receive input currents in order to link the relative portion of application code

Spike Sources will be considered neural population of a particular type (SpikeSource). The partitioner and the compiler will then create only the connection structures while they will skip the neural data themselves. Spikes can be produced by:

- Random Process: in this case parameters for the process (eg. rate) must be passed to the component generating the spikes
- List: in this case the list needs to be created, parsed and compiled to the appropriate spike generator component
- Dynamic Source (eg. Silicon Retina): spikes will be injected to a link by an external source

TBD: how are spikes generated? Process on the host machine? Monitor (or dedicated) process on chip?

Recorders will enable logging options for the selected Populations. Log can either occur in SDRAM or can be streamed to the Ethernet TBD. Recorders can also be used to send spikes out of the Ethernet link. Will be defined as:

- Population: target Population
- Variable: the variable to log (u, v, i)
- Destination: Ethernet or SDRAM

The Population/Projection abstraction let the system deal with aggregated groups rather than with single neurons and can therefore be used as an efficient representation in the mapping and compiling binary phases as well.

TBD: The output format for this section can be an exchange file or python structures to be passed to the next stage, the partitioner. I suggest using a sqlite DB to store the configuration between different software layers, and be able to update retrieve information with standard SQL language. In this way information can be spread across all software components (mapping, compiling, managing input/output, visualising) and represented in a standard, easy to consult and efficient way.

3.3.4.2 PyNN API functions list

Contents PyNN API version 0.7

3.3.4.3 Simulation setup and control

setup(timestep=0.1, min_delay=0.1, max_delay=10.0, **extra_params)
end(compatible_output=True)
run(simtime)
reset() To be implemented
get_time_step() To be implemented
get_current_time() To be implemented
get_min_delay() To be implemented
get_max_delay(): To be implemented

3.3.4.4 Object-oriented interface for creating and recording networks

Population
__add__(self, other)
__getitem__(self, index)
__init__(self, size, cellclass, cellparams=None, structure=None, . . .
describe(self, template='population_default.txt', engine='default')
get(self, parameter_name, gather=False)
getSpikes(self, gather=True, compatible_output=True): Implemented as a standalone script using SDRAM/network output
get_v(self, gather=True, compatible_output=True): Implemented as a standalone script
id_to_index(self, id)
inject(self, current_source): To be implemented as an SDP message passing from the host machine and the application framework
printSpikes(self, file, gather=True, compatible_output=True): Implemented as a standalone script using SDRAM/network output
print_v(self, file, gather=True, compatible_output=True): : Implemented as a standalone script using SDRAM/network output
randomInit(self, rand_distr)
record(self, to_file=True): Implemented as record(self, save_to=True) where save_to defines if the data needs to be saved in SDRAM or sent through the ethernet (deprecated)
record_v(self, to_file=True): Implemented as record(self, save_to=True) where save_to defines if the data needs to be saved in SDRAM or sent through the ethernet (deprecated)
save_positions(self, file): To be implemented
set(self, param, val=None)

3.3.4.5 PopulationView

To be implemented, TBD how treat overlapping PopulationView

3.3.4.6 Assembly

Partially implemented at a PyNN level. __add__(self, other)
__getitem__(self, index)
__iadd__(self, other): To be implemented

__init__(self, *populations, **kwargs)
__iter__(self)
__len__(self)
describe(self, template='assembly_default.txt', engine='default')
get_gsyn(self, gather=True, compatible_output=True)
Classes for defining spatial structure Imported from PyNN
Classes for defining spatial structure

3.3.4.7 Object-oriented interface for connecting populations of neurons

Projection
__getitem__(self, i)
__init__(self, presynaptic_population, postsynaptic_population, method, . . .
getDelays(self, format='list', gather=True): To be implemented
getSynapseDynamics(self, parameter_name, format='list', gather=True): To be implemented
getWeights(self, format='list', gather=True): To be implemented
printWeights(self, file, format='list', gather=True): To be implemented
randomizeDelays(self, rand_distr): To be implemented (it is possible to define random weights/delays passing a RandomObject to the Projection constructor
randomizeSynapseDynamics(self, param, rand_distr): To be implemented
randomizeWeights(self, rand_distr): To be implemented (it is possible to define random weights/delays passing a RandomObject to the Projection constructor
saveConnections(self, file, gather=True, compatible_output=True): To be implemented
setDelays(self, d)
setSynapseDynamics(self, param, value): To be implemented (it is possible to set them when the Projection is created)
setWeights(self, w): To be implemented (it is possible to set them when the Projection is created)
size(self, gather=True): Partially implemented
AllToAllConnector
__init__(self, allow_self_connections=True, weights=0.0, delays=None, . . .
OneToOneConnector
__init__(self, weights=0.0, delays=None, space=<pyNN.space.Space object . . .
FixedProbabilityConnector
__init__(self, p_connect, allow_self_connections=True, weights=0.0, . . .
DistanceDependentProbabilityConnector: Translated as a FromList Connector
__init__(self, d_expression, allow_self_connections=True, weights=0.0, . . .
FromListConnector
__init__(self, conn_list, safe=True, verbose=False)
FromFileConnector
__init__(self, file, distributed=False, safe=True, verbose=False)

3.3.4.8 Procedural interface for creating, connecting and recording networks

Not implemented as this is the low level Api.

3.3.4.9 Neural Models

Standard Models: IF_curr_exp: 16 and 32 bit
IF_cond_exp: 32 bit
EIF_cond_exp_isfa_ista: Under implementation
SpikeSourcePoisson: Implemented, it generates spikes according to a Poisson process that is extracted from a uniformly distributed random variable.
SpikeSourceArray: Implemented so that a set of spikes is loaded on the SpiNNaker system and then parsed at simulation time, and the spikes are distributed according to the loaded pattern.
__init__(self, parameters)
Non Standard Models:

- IZK_curr_exp: an implementation of the Izhikevich neuron with 2 first order kinetic synaptic types
- NEF_input: Translates values to Population spike trains using the Neural Engineering Framework
- NEF_output: Translates Population spike trains to values using the Neural Engineering Framework
- SpikeSink: Gathers spikes and outputs them through the ethernet
- Dummy: population used for profiling
- SpikeSource: Receive spike packets from the host and propagates them in the neural network. It needs a standalone program on the host machine sending spike packets. The software on the host side has been called “SpikeServer”.

3.3.4.10 Specification of synaptic plasticity

SynapseDynamics
__init__(self, fast=None, slow=None)
describe(self, template='synapsedynamics_default.txt', engine='default')
STDPMechanism
__init__(self, timing_dependence=None, weight_dependence=None, . . . describe(self, template='stdpmechanism_default.txt', engine='default')
AdditiveWeightDependence
__init__(self, w_min=0.0, w_max=1.0, A_plus=0.01, A_minus=0.01)
SpikePairRule
__init__(self, tau_plus=20.0, tau_minus=20.0)
FullWindow
__init__(self, tau_plus=20.0, tau_minus=20.0)

TimeToSpike
__init__(self, L_parameter=-65, tau_plus=20.0, tau_minus=20.0)

SpiNNaker implements three learning rules:

1) Standard STDP rule, that can be instantiated using the class FullWindow. For details refer to the article “Implementing Spike-Timing-Dependent Plasticity on SpiNNaker Neuromorphic Hardware" by Xin Jin, Alexander Rast, Francesco Galluppi, Sergio Davies and Steve Furber

2) Spike-pair STDP, also known as nearest-neughbour STDP, that can be instantiated using the class SpikePairRule. The implementation is similar to the standard STDP rule, but the synaptic weight update is limited to the nearest pair of spikes.

3) STDP with Time-To-Spike forecast, that can be instantiated using the class TimeToSpike. This learning rule is suitable only for Izhikevich neurons. For details of the learning rule and its implementation refer to the article "A forecast-based STDP rule suitable for neuromorphic implementation" by Sergio Davies, Alexander Rast, Francesco Galluppi and Steve Furber

3.3.4.11 Current Injection

Current injection To be implemented via SDP message passing between the host and the application framework

Example

PyNN example script to run a multichip synfire chain model on the SpiNNaker test board.

A synfire chain (synchronous firing chain) is a feed-forward network of neurons with multiple layers or pools. In a synfire chain, neural impulses propagate synchronously back and forth from layer to layer. Each neuron in one layer feeds excitatory connections to neurons in the next, while each neuron in the receiving layer is excited by neurons in the previous layer. (http://en.wikipedia.org/wiki/Synfire_chain)

This scripts allocates pool_number layers on each chip, up to 4 chips and 512 neurons per chip.

```
#!/usr/bin/python

# Imports the pyNN.spiNNaker module
from pyNN.spiNNaker import *

# Defines the synfire chain model.
pool_size = 256          # Numbers of neurons in a pool
pool_number = 8          # Total numbers of pools

runtime = 1000           # Time of the simulation


fwd_weights = 7         # Feed forward weights
bck_weights = -0.1      # Inhibitory feedback

# setting up the PyNN environment
```

```
setup(timestep=1.0, min_delay = 1.0, max_delay = 16.0,
      spiNNChipAddr='spinn-1' # IP address of the spiNNaker board
      )


print "Total number of pools across system:", pool_number


# Defines neural parameters for the population
cell_params = { 'tau_m'       : 32,
                'v_init'      : -85,
                'v_rest'      : -75,
                'v_reset'     : -75,
                'v_thresh'    : -55,
                'tau_syn_E'    : 5,
                'tau_syn_I'    : 2,
                'tau_refrac'    : 10,
               }


#### Neural Populations creation

populations = []            # List containing all the populations in the model

# Loop creating the populations - each population models a pool in the synfire chain
for i in range(pool_number):              # Total number of pools in the system
    populations.append(Population(pool_size,            # Neurons per population
                                  IF_curr_exp,          # PyNN Standard Neuron Model.
                                  cell_params,          # Neuron parameters
                                  label='pool_%d' % i)  # Label for the population
    )
    populations[i].record()


#### Connections creation
connections = []            # List containing all the connections in the model

# Loop generating the feedforward connections. Pool N will be connected to pool N+1
for i in range(pool_number-1):                  # Cycling all populations in the model
    connections.append(Projection
                       (populations[i],       # Presynaptic population
                        populations[i+1],     # Postsynaptic population
# OneTo One will connect the first neuron in the presynaptic population
# to the first neuron in the postsynaptic population
                        OneToOneConnector(weights=fwd_weights, delays=delayDistr),
# Target synapse type - IF_curr_exp supports two differents current bins. one for
# excitatory synapses and one for inhibitory synapses with two different time constants
                        target='excitatory',
                        label='pool_%d-pool_%d' % (i, i+1)       # Connection label
                        )
                      )


# Last population connected to first population
# shows how to build inhibitory connections
connections.append(Projection(populations[pool_number-1],
                              populations[0],
                              OneToOneConnector(weights=bck_weights*0.1, delays=1),
                              target='inhibitory',
                              label='close_loop'
                              )
```

```
                    )

# Injecting currents in the first pool – setting up the input waveform
current_source = StepCurrentSource([0,   50,   1000],      # Time
                                   [0,   1,      0])       # Amplitude
# Injecting in the first pool
current_source.inject_into(populations[0])

run(runtime)                # Simulation time

end()                       # And that's all folks!
```

The script above builds a spiking neural network composed by 8 pools of 256 neurons each connected in a feed-forward way, where every nth neuron of each population is connected to the nth neuron in the next population. It records spikes from every population. The first population is injected with a step current.

Such a network can be represented with the structures defined above as:

Populations				
id	size	cell_type	parameters	label
0	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_0
1	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_1
2	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_2
3	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_3
4	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_4
5	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_5
6	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_6
7	256	IF_curr_exp	{'tau_m' : 32,'v_init' : -85, 'v_rest' : -75, ..}	pool_7

Projections						
ID	source	dest	target	parameters	plasticity	label
0	0	1	excitatory	{weights=7, delays=1}	none	pool_0-pool_1
1	1	2	excitatory	{weights=7, delays=1}	none	pool_1-pool_2
2	2	3	excitatory	{weights=7, delays=1}	none	pool_2-pool_3
3	3	4	excitatory	{weights=7, delays=1}	none	pool_3-pool_4
4	4	5	excitatory	{weights=7, delays=1}	none	pool_4-pool_5
5	5	6	excitatory	{weights=7, delays=1}	none	pool_5-pool_6
6	6	7	excitatory	{weights=7, delays=1}	none	pool_6-pool_7
7	7	0	inhibitory	{weights=7, delays=-0.01}	none	pool_7-pool_0

Recorders			
ID	population_id	observable	save_to
0	0	spikes	SDRAM
1	1	spikes	SDRAM
2	2	spikes	SDRAM
3	3	spikes	SDRAM
4	4	spikes	SDRAM
5	5	spikes	SDRAM
6	6	spikes	SDRAM
7	7	spikes	SDRAM

Currents		
id	population_id	parameters
0	0	(’type’:’list, ’times’:’[0, 50, 1000]’, ’amplitudes’:’[0, 1, 0]’

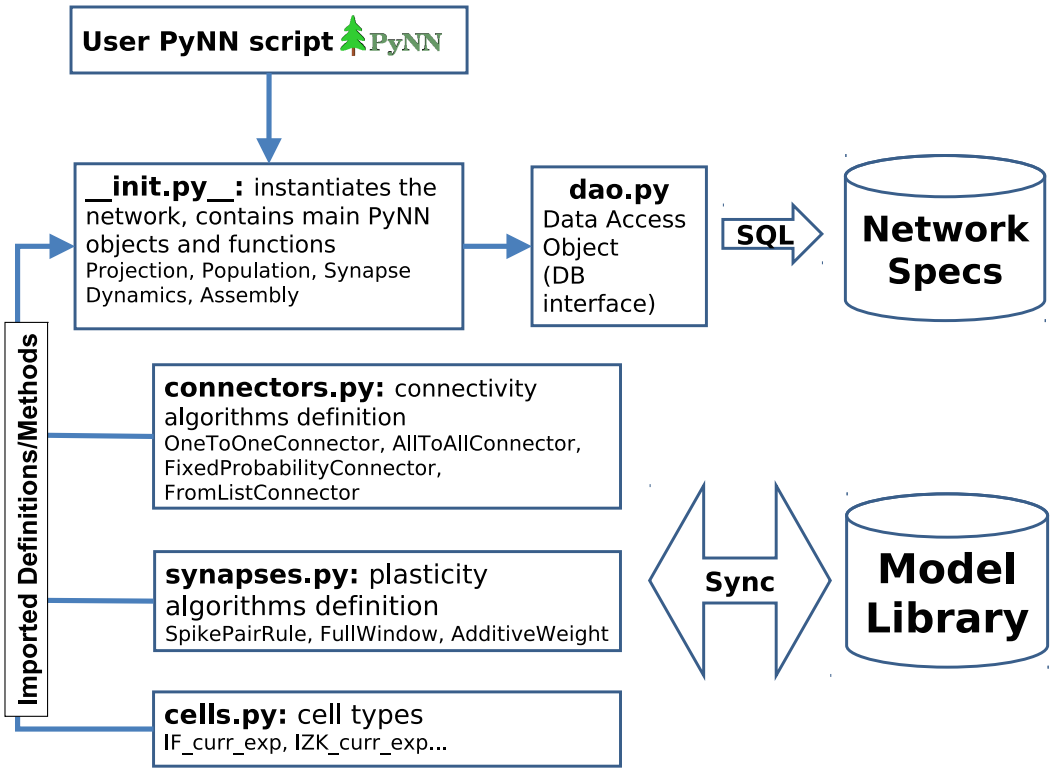


Figure 3.3.8: PyNN/SpiNNaker interface structure.

3.3.5 Damson development route

3.3.5.1 Damson program compilation

A Damson program for SpiNNaker consists of a single source file containing code for a number of nodes. Each node maps to a single application processor in SpiNNaker. When the compiler (damsonc) is run on a Damson program, the output is a number of object files (in ELF format) where each object file contains the code of a single node in the source.

3.3.5.2 Damson code components

The object files refer to a set of routines in a Damson library known as “damsonlib”. This provides arithmetic functions (multiply and divide) for the fixed point data type used by Damson as well as formatted output routines. A jump table is appended to the code of each node so that calls can be made into damsonlib from the code for each node. The code to be loaded onto each processor consists of the node code with jump table, a copy of damsonlib and also a separate runtime system which implements low-level SpiNNaker specific operations such as timers and packet transmission. The runtime system is currently implemented specifically for Damson but will be merged with the standard SpiNNaker API in due course.

3.3.5.3 Mapping code to SpiNNaker processors

The Damson compiler also produces a file which details the mapping between the code for each node and the object file containing it. This file also provides a packet communication map which indicates to which other nodes a given node sends packets. This latter information is needed to allow Damson nodes to be allocated to specific application processors in a SpiNNaker system and to allow generation of the multicast routing tables to route packets correctly. In due course the PACMAN program will be used to perform this function. For now, the routing tables are generated by hand. This limits the scale of Damson demonstration programs somewhat!

3.3.5.4 Runtime system

The Damson runtime system currently provides a set of support routines and interrupt handlers. A timer interrupt may be started by a Damson node at a specified clock rate. A "packet received" interrupt handler routes packets to a specific handler at a node depending on the source node of the packet.

3.3.5.5 Damson development flow

In the diagram below the box marked "Object Code" is the set of ELF object files produced by "damsonc". The box marked "Netlist" is the map file produced by the compiler. The netlist and a description of the target SpiNNaker system are fed to PACMAN (Partitioning and Configuration Manager) which generates a set of multicast routing tables (one per SpiNNaker chip) and also a driver file used by the code linking stage to build the image(s) to be loaded. The object files are fed to the linker where they are combined with the runtime system (based around the SpiNNaker API) to make the code images for loading.

3.3.6 PACMAN: partition and configuration manager

3.3.6.1 Introduction

The function of PACMAN - the Partitioning And Configuration MANager, is to transform the high-level representation from PyNN, Lens or DAMSON into a physical on-chip implementation: the instruction and data binaries the boot process loads in order to configure the system.

Example of network representation in PACMAN, showing two different mappings of a neural network model on the SpiNNaker system. The network consists of 5 populations interconnected in a random way. PACMAN is set to map the model by fitting up to 100 neurons in each application core. Two different mapping cases are presented: top) a single population of 150 neurons fits in 1 and 1/2 cores; bottom) two populations of 50 neurons can fit in a single core.

PACMAN is based on a Database that holds three representations of the neural network (fig.3.3.10):

- **Model Level**: the network as specified in the high-level language (PyNN, Damson, LENS etc.)

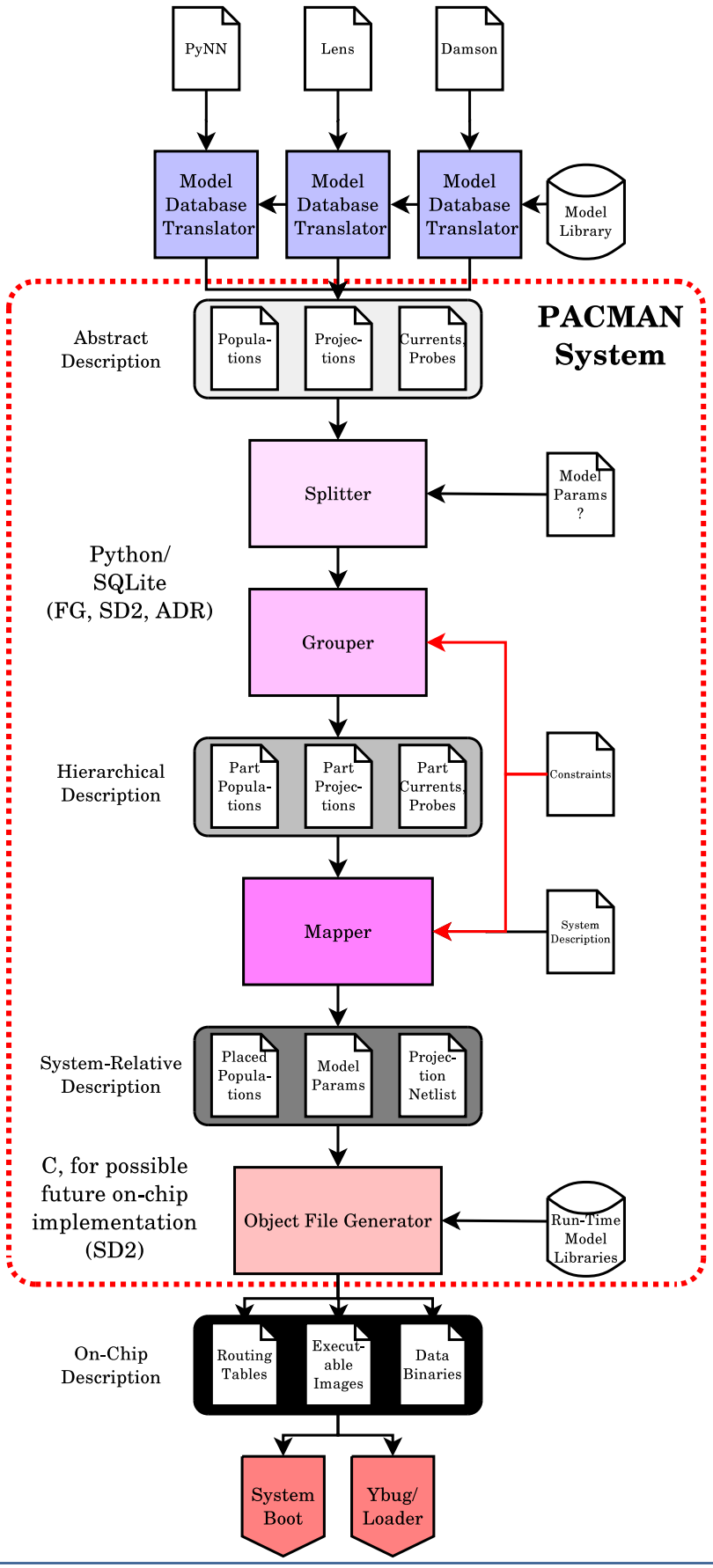


Figure 3.3.9: PACMAN internal structure

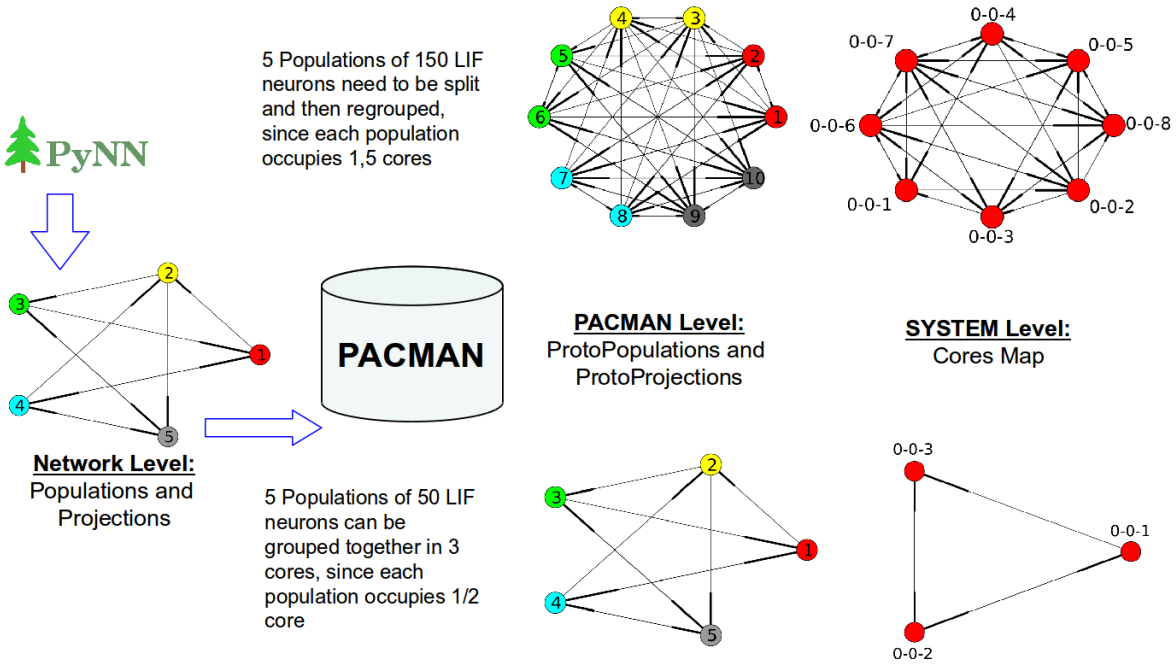


Figure 3.3.10: Example of network representation in PACMAN

- **PACMAN Level**: the network is partitioned in *Pre_* Populations that can be fit into a single computing core. Projections, probes and inputs are split accordingly. Pre_Populations that can be fit into a core a grouped together
- **System Level**: a map linking groups of Pre_Populations with a particular core (identified by its coordinates) in the system

Such translations enable the network to be mapped and deployed on the SpiNNaker system, by generating the binaries needed to configure the simulation components and topology.

In FPGA language, it may be considered similar to a configuration bitstream generator. Because of the large and highly associative nature of the data structures, it is *essential* that algorithms for PACMAN must be a) *incremental*, b) of *linear* (or at the very most NlogN) complexity in the number of neurons.

PACMAN itself (fig.3.3.9) is divided in 4 different steps:

- **Splitting**, responsible for splitting neural populations which won't fit in a single core (because of memory or comutational complexity limitations) into *Pre_Populations* that will fit in a core (like a neural "place" operation)
- **Grouping**, responsible of collating Pre_Populations which can be run using the same application code in order to fit more of them onto a single core. Those first two steps define the Partitioner
- **Mapping**, responsible of performing virtual-to-physical translation and allocate groups to cores

- **Object file generation**, which creates the actual data binaries from the partitioned and mapped network.

PACMAN works internally on an SQL database. Fig. 3.3.11 shows the schema for the database. As PACMAN is invoked the Network Specification schema has already been populated by the spiNNaker.pyNN, Lens's SpiNNaker.tcl or DAMSON plugin as described in the relative sections.

An example of network representation in PACMAN, showing two different mappings of a neural network model on the SpiNNaker system is presented in figure. The network consists of 5 populations interconnected in a random way. Each population receives connections from other populations (including self connections - not showed in figure for simplicity) PACMAN is set to map the model by fitting up to 100 neurons in each application core:

- (top) if populations are too big to fit in a single core (150 neurons per population, top portion of the figure) they are split in 2 Pre_Populations of 100+50 neurons. Projections and other network elements are split accordingly. The resulting model maps to 8 cores in the SpiNNaker system

- (bottom) if populations are small enough to be fit a sub-portion of a single core (50 neurons per population, bottom portion of the figure) they are grouped in the same core, up to the maximum number of neurons. Projections and other network elements are grouped accordingly. The resulting model maps to 3 cores in the SpiNNaker system

Note:	PyNN and Lens use different terminology to refer to associated blocks of neurons (*Populations* and *Groups*, respectively) and connections (*Projections* and *Blocks*, respectively). In addition a "neuron" in Lens goes under the name of *Unit* and a "synapse" under the name of *Link*. To avoid confusion we use the PyNN terminology throughout; the equivalent Lens names may be substituted in the appropriate places for an MLP network generation.

3.3.6.2 Splitting

During the Partitioning phase Populations that span more than one core will be divided into *Pre_Populations* that can be allocated into single cores. In order to do this the system needs to know:

- the maximum number of neurons that can be fitted in one single core. This information is stored in the *max_neuron_per_fasc* field of the *cell_type* table of the Model Library schema

After having split Populations into Pre_Populations Projections need to be exploded into Pre_Projections as well. A Pre_Projection is a Projection between 2 Pre_Populations.

The output of the Partitioner will be stored in the Pre_populations and Pre_projections tables which have the following structure:

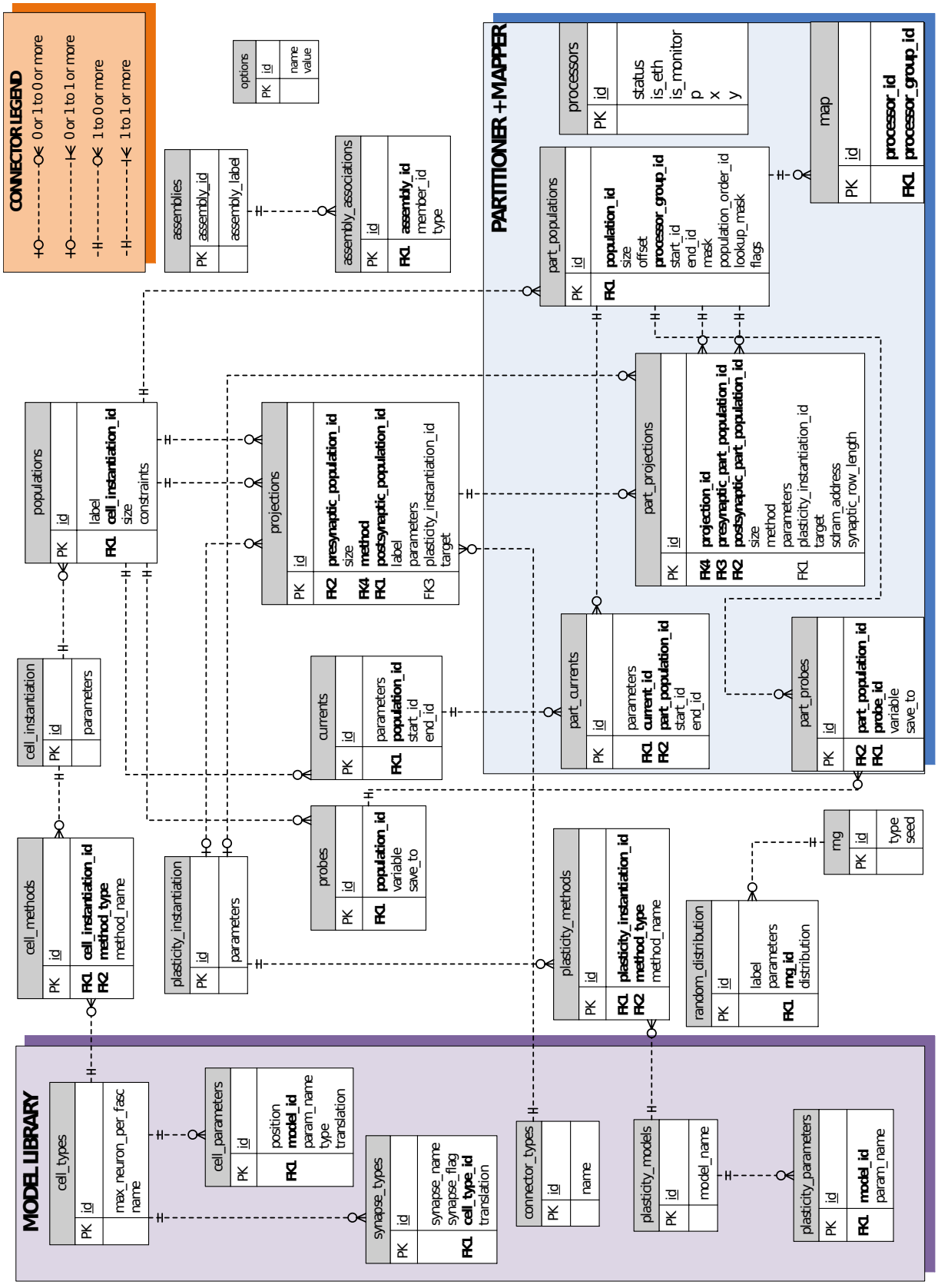


Figure 3.3.11: Database structure: model library and network specification

Pre_populations		
Field	type	description
id (PRIMARY KEY)	INTEGER	ID defining a Pre_Population
population_id (FOREIGN KEY: populations.id)	INTEGER	ID defining the source Population
cell_ids	TEXT	a slice object containing the subset of the Population cell ids mapped by the Pre_Population
start_id	INTEGER	the core-relative starting id for the Pre_Population in the core
end_id	INTEGER	the core-relative ending id for the Pre_Population in the core
mask	INTEGER	mask defining Population and Neuron ID in the routing key
population_order_id	INTEGER	order of the Pre_Population in the core

notes: group_id will be used during the Grouping phase. population_order_id, start_id, end_id and mask are using for Population-based routing. Pre_Populations need to be ordered decreasingly according to their size

Pre_populations		
Field	type	description
id (PRIMARY KEY)	INTEGER	ID defining a Pre_Projection
presynaptic_population_id (FOREIGN KEY: Pre_populations.id)	INTEGER	ID defining the presynaptic Pre_Population in the Pre_Projection
postsynaptic_population_id (FOREIGN KEY: Pre_populations.id)	INTEGER	ID defining the postsynaptic Pre_Population in the Pre_Projection
method (FOREIGN KEY: connector_types.id)	INTEGER	ID referring to the connector type between the 2 Pre_Populations
size	INTEGER	number of single connections (neuron to neuron) in the Pre_Projection
source	INTEGER	string specifying which attribute of the presynaptic cell signals action potentials
target (FOREIGN KEY: TBD)	INTEGER	ID referring to which synapse on the postsynaptic cell to connect to
parameters	TEXT	a string containing a Dictionary of parameters (eg. weights, delays)
plasticity_instantiation_id (FOREIGN KEY: plasticity_instantiation_id)	INTEGER	ID defining the type of plasticity algorithm and its parameters for the Pre_Projection
label	TEXT	human readable label for Pre_Projection

notes: The Pre_Projection table has the same structure of the Projection table, but presynaptic_population_id and postsynaptic_population_id refer to Pre_Populations rather than Populations. source is done for compatibility with PyNN

Implementation

partitioner/splitter.py

The process is set up by calling the following functions:

- split_populations: splits Populations accordingly to the maximum number of neurons for that model
- split_projections: splits Projections accordingly to Pre_populations. recalculates offsets for ids (FromListConnector)
- split_probes: splits Probes accordingly to Pre_populations

They all take as input an instantiation of the db. An example on how to run the splitter is reported below:

```
import sys
print "Loading DB:", sys.argv[1]
db = load_db(sys.argv[1])       # imports the DB passed as argv[1]
db.clean_part_db()              # cleans the part_* tables
split_populations(db)
split_projections(db)
split_probes(db)
```

3.3.6.3 Grouping

The Grouping stage needs to know information about the system and the model in order to translate the one to the other. At the model level the partitioner needs to know information passed either by the pyNN.spiNNaker plugin, or Lens' SpiNNaker.tcl script, in particular:

- Number of Pre_Populations, number of neurons in each Pre_Populations, neuron type
- Number and type of Projections
- Number and type of Inputs and Recorders
- Number of types and associated parameters for neurons, projections, and recorders

At the system level the partitioner needs to know

- Number of neurons that can be modelled in a single core for a specific neuron type-/application. This includes parameters from synaptic and plasticity model as well (eg. all neuron which share core-wise parameters as STDP tables can be put together in the same core)
- How neurons and other complex model objects are assembled, i.e. what component functions and parameters must be built into them.
- Obey constraints on the maximum number of neurons per core for a given application
- Only place Populations with the same (composite) neural type on the same core
- Only Populations with the same mapping constraint can be grouped together

The output from the Grouping stage will write its output in the group_id field of the Pre_populations table, grouping different populations in the same group.

Implementation

The Grouper joins homogeneous Pre_populations together up to the maximum number of neurons for that model

The process is set up by calling the following functions:

- get_groups: Retrieves all the Populations that can be grouped together. Such Populations are homogeneous for neural model and plasticity instantiations. outputs a list of lists where each element is a list of groupable Populations

- grouper: groups populations accordingly to the maximum number of neurons for that neural model
- update_core_offsets: sets the core offset for that Population (position of the Population in the group)

They all take as input the instantiation DB.

```
db = load_db(sys.argv[1])        # imports the DB
groups = get_groups(db)
grouper(db, groups)
update_core_offsets(db)
mapper(db)
create_core_list(db)
```

Grouping criteria can be defined in SQL language. In this case 2 queries need to be designed, accordingly to the grouping criteria defined. The first query (*get_grouping_rules* in *dao.py*) extracts the possible combination of criteria. For instance if we have the three criteria before mentioned (group populations with same neural model, plasticity instantiation and mapping constraint)

3.3.6.4 Mapper

This information can be passed to the Mapper stage along with model-specific data provided by the high-level generation tool, which has now all the information needed to locally generate each portion of the network.

The Mapper task is to assign groups, as organized by the grouper, to a specific core. Available cores are listed in the Model Library and they are dynamically used by the mapper. Information needed by the Mapper are:

- Size and health of the system: number of chip/cores available for neural simulation and their geometry
- Constraints relative to spike/current input/output (eg. neurons that send output must be on Ethernet attached chip)
- User and System constraints that affect e.g. model geometry or allowable activity rates

Mapping constraint are associated to Populations in the DB, and they define a range of chip/cores where the Population should be matched. This information can be set by a user with a custom function, or by a network analysis tool as networkx.

The Mapper will first process groups that have mapping constraints trying to satisfy them, then allocating all the non-constrained groups. If mapping constraints are inconsistent an Exception will be raised.

Output from the Mapper is a hierarchical physical description of the entire network (which will be in a series of tables as below). This in turn passes to an Object File generator (which for the moment will reside on the Host but could eventually be migrated to an on-SpiNNaker implementation) which flattens the network and generates the (flattened) actual data binaries.

Processor ID	X	Y	P	Type ID	Number of Neurons	Start ID
0	0	0	0	1	512	0
1	0	0	1	2	512	0
2	1	0	0	3	512	0

Projection ID	Type ID	Number of Synapses	Start ID	X	Y	SDRAM Off-set
0	1	256	0	0	0	0x0
1	1	256	256	0	0	0x40C
2	2	1024		0	0	0x80C
3	3	512	0	1	0	0x0

Model ID	Model
1	IF_curr_exp
2	IF_curr_exp_stdp
3	IZK_curr_exp_stdp

The Mapper will link the *Pre_populations* table with the *processor* table through the association table *map* so defined:

map		
Field	type	description
processor_id (FOREIGN KEY: processor.id)	INTEGER	ID defining a physical core in the system
group_id (FOREIGN KEY: Pre_populations.group_id)	INTEGER	ID defining the group to be mapped to the corresponding core

processor		
Field	type	description
processor_id (PRIMARY KEY: processor.id)	INTEGER	ID defining a physical core in the system
x	INTEGER	X coordinate for the chip containing the processor
y	INTEGER	Y coordinate for the chip containing the processor
p	INTEGER	Virtual ID for the core in the chip
status	TEXT	Health status for the processor
is_eth	BOOL	Identifies a root chip if True
is_monitor	BOOL	Identifies a monitor processor if True

Implementation

Mapping constraints are defined in the *constraint* field in the Populations table. They can be used to associate a Population to a specific range/value of chip id and core id.

Information in here can be written by any network analysis tool or manually defined by the user (eg. using the function *set_mapping_constraint* in the *pyNN.spiNNaker* module).

The Mapper dynamically retrieves the available cores list from the Model Library DB and tries to allocate groups with constraints first, then all the other groups, consuming available processors until groups are all allocated or there is no more space to allocate the group due to a map inconsistency.

3.3.6.5 Object File Generator

At this point the hierarchical description still contains abstract objects rather than single neurons. The mapper organises this information is organized so that binary file generation can be performed locally by target processor, evaluating the table produced by the mapper core by core.

TBD: The compilation process described here will occur on the host machine for the first version of the software, but the design ensures that it will be easily portable to the the SpiNNaker system so that file generation can occur on-chip.

Neural data compilation for a particular core will include these steps:

- Retrieve all the Populations associated with the core
- Load any necessary model configuration files (which describe how to build complex neural or synaptic models).
- Assemble the model files into an executable and create neural data structures in DTCM
- Build the application, linking the neural/synapse model type with extra information needed (eg. preconfigured lookup tables for STDP) and switches (eg. for logging)
- Generating the routing table files for each chip
- Build the connectivity information in SDRAM and routing look-up tables (second level of routing)

TBD: One way to define model configuration files for non-standard models (in PyNN) uses Translation XML or a translation table in the DB. We envisage a separate application, the Model Builder, that in future will allow automated generation of Translations for new models. The Mapper links the `cell_type` and the translation in one single configuration file.

The table for translating is defined as follows:

cell_parameters		
Field	type	description
id (PRIMARY KEY)	INTEGER	ID defining a cell parameter
model_id (FOREIGN KEY: cell_type.id)	INTEGER	ID defining the cell_type to which the parameter belongs
param_name	TEXT	Name for the parameter
type	TEXT	Variable type/dimension (short, int, uint etc.)
translation	TEXT	Translation for the parameter (eg. toInt(multiply(x,p1))), written in a form that can be evaluated by the Object file generator.
position	INTEGER	Position of the parameter in the compiled data structure

For the translation field specific operators have been developed to ensure the compatibility with all the possible type of input which may be provided (see following section). A number of operators have been defined for the basic functions:

Translation operators	
Operator	Description
add (a,b)	The operator adds the operands **a** and **b**, if they are numbers, or adds each element of the list **a** to the correspondent element of list **b**, if **a** and **b** are lists. If **a** is a number and **b** is a list (or vice-versa) then **a** is added to each element of the list **b** (or vice-versa).
subtract (a,b)	The operator subtracts the operands **a** and **b** ($a-b$), if they are numbers, or subtracts each element of the list **b** from the correspondent element of list **a** ($a[n]-b[n]$), if **a** and **b** are lists. If **a** is a number and **b** is a list (or vice-versa) then the operation is performed using the same number as one of the operands and each of the element of the list as the second operand.
multiply (a,b)	The operator multiplies the operands **a** and **b**, if they are numbers, or multiplies each element of the list **a** with the correspondent element of list **b**, if **a** and **b** are lists. If **a** is a number and **b** is a list (or vice-versa) then **a** is multiplied to each element of the list **b** (or vice-versa).
divide (a,b)	The operator divides the operands **a** and **b** (a/b), if they are numbers, or divides each element of the list **a** by the correspondent element of list **b** ($a[n]/b[n]$), if **a** and **b** are lists. If **a** is a number and **b** is a list (or vice-versa) then the operation is performed using the same number as one of the operands and each of the element of the list as the second operand.
power (a,b)	The operator computes the power a^b, if **a** and **b** are numbers, or computes the power of each element of the list **a** by the correspondent element of list **b** ($a[n]^{b[n]}$), if **a** and **b** are lists. If **a** is a number and **b** is a list (or vice-versa) then the operation is performed using the same number as one of the operands and each of the element of the list as the second operand.
exponential (a)	The operator computes the operation $\exp(a)$ if **a** is a number, or $\exp(a[n])$ if **a** is a list of numbers
toInt (a)	The operator returns the integer part of **a**, if it is a number, or, if **a** is a list, it returns the integer part of each element of the list

3.3.6.6 Neural Data Structure generation

The neural data structure writer cycles all mapped processor and generates the data structures for each of them. It outputs a different file for each core, containing all the data

structures for the neuron modelled by that processor. Neural data structures are compiled as it follows:

```
header (1x file)
– uint runtime
– unit max_synaptic_row_length
– unit max_delay
– uint num_pops
– uint total_neurons
– uint size_neuron_data
– uint reserved2 (NULL)
– uint reserved3 (NULL)

population metadata (1x population)
– uint pop_id
– uint flags
– uint pop_size (number of neurons in the population)
– uint size_of_neuron (size of a single neuron)
– uint reserved1 (NULL)
– uint reserved2 (NULL)
– uint reserved3 (NULL)

neural structures (1x neuron)
... list of parameters ...
```

The neural structures are computed by retrieving the translation from the cell_params table in the Model Library. This table also contains the position of the parameter in the neural structure and its size. Parameters can be defined as single values, random distribution or arrays explicitly defining the parameter value for each neuron.

3.3.6.7 Automatic Run Script generation

Pacman generates an automatic run script that is used to load the data in the right chip/-core/memory location. Doing so, it also selects which executables are to be loaded in each of the cores. In particular, it may need to select executables featuring plasticity behaviour to be loaded in specific cores. To be able to discern between executables with or without plasticity, different file names have been used, with this categorization:

Executable names	
Name	Description
lif(.aplx)	Binary featuring **leaky integrate-and-fire** neuron **without learning** capabilities
lif_stdp(.aplx)	Binary featuring **leaky integrate-and-fire** neuron and **standard STDP** rule
lif_stdp_sp(.aplx)	Binary featuring **leaky integrate-and-fire** neuron and **spike-pair STDP** rule
lif_cond(.aplx)	Binary featuring **leaky integrate-and-fire conductance-based** neuron **without learning** capabilities
lif_cond_stdp(.aplx)	Binary featuring **leaky integrate-and-fire conductance-based** neuron and **standard STDP** rule
lif_cond_stdp_sp(.aplx)	Binary featuring **leaky integrate-and-fire conductance-based** neuron and **spike-pair STDP** rule
izhikevich(.aplx)	Binary featuring **Izhikevich** neuron **without learning** capabilities
izhikevich_stdp(.aplx)	Binary featuring **Izhikevich** neuron and **standard STDP** rule
izhikevich_stdp_sp(.aplx)	Binary featuring **Izhikevich** neuron and **spike-pair STDP** rule
izhikevich_tts(.aplx)	Binary featuring **Izhikevich** neuron and **STDP with Time-To-Spike forecast** rule

The name of the executables (without the ".aplx" extension) is also the name of the target of the makefile to generate the correspondent binary file.

3.3.6.8 MLP PACMAN

A modification of the original PACMAN design handles the configuration of MLP networks from Lens scripts. The modification retains support for spiking models while adding functionality for the MLP. This requires some architectural changes.

Design Considerations

1) Conformity with PACMAN design principles. In the main, this means instantiation based on a Population/Projection model, not flattening the description internally until the final data-file generation step, and using the PACMAN database to hold the internal representation of the network. It also means using plug-in modules to implement necessary model-specific functionality that could not be placed in the main PACMAN tools without sacrificing commonality.

2) Separation of the basic "machinery" from the model-specific data. The code that generates the data structures and mapping is kept independent from the model data itself. As much as possible, functions and interfaces are designed to take parameters which specify the type of model being instantiated and its data structures.

3) Avoidance of methods that embed global knowledge about the model into the code. It is known, in some cases, how the MLP, or for that matter spiking networks, will be implemented in terms of the details of the mapping. This could either be coded implicitly, in the generation algorithms, or explicitly, through parameter settings. Where possible the MLP PACMAN extension uses the latter method (sometimes using helper functions that themselves are parameters.)

4) Maximal re-use of existing PACMAN facilities. The implementation uses existing PACMAN code unless a change is absolutely necessary for some form of support.

Revised schema

In order to provide the structures necessary to map and route the MLP model, and also in order to make the database more coherent, the MLP PACMAN implementation extends the schema with several new tables:

scenarios Added to permit a representation of Lens ExampleSets. A scenario is considered an ExampleSet; this feature could also be used in spiking models to specify a particular group of stimuli representing a complete real-time environment

stimuli Added to permit a representation of Lens Examples. A single example is a stimulus; likewise in spiking models this could be used to represent a multiple-input stimulus with common temporal parameters.

routes Represents a complete path from a source processor to a target processor. This is required in Lens because the same population may have multiple routes with different keys (e.g. for forward/backprop)

routing_entries Represents a single routing entry in a given chip. Both this and the routes table makes the PACMAN system more coherent, in that the Router now writes to the DB rather than only to an internal variable. (Thus after routing the routes can be inspected, queried, etc, and changes could potentially be made) Various other tables have had fields added or removed to support the MLP. Fig 3.3.12 shows the updated schema.

The components

The PACMAN MLP extension consists of the following components (fig. 3.3.13):

1) Front-end translator: An entirely new component that implements the interface to Lens. It is written in TCL.

2) MLP preprocessor plug-in: A new PACMAN component that transforms the input model in the database, prior to its being passed to the splitter. Its main function is to split groups into Weight, Sum, and Threshold populations.

3) pacman_objs: A new, object-oriented interface to PACMAN. This permits object-style interface to the database.

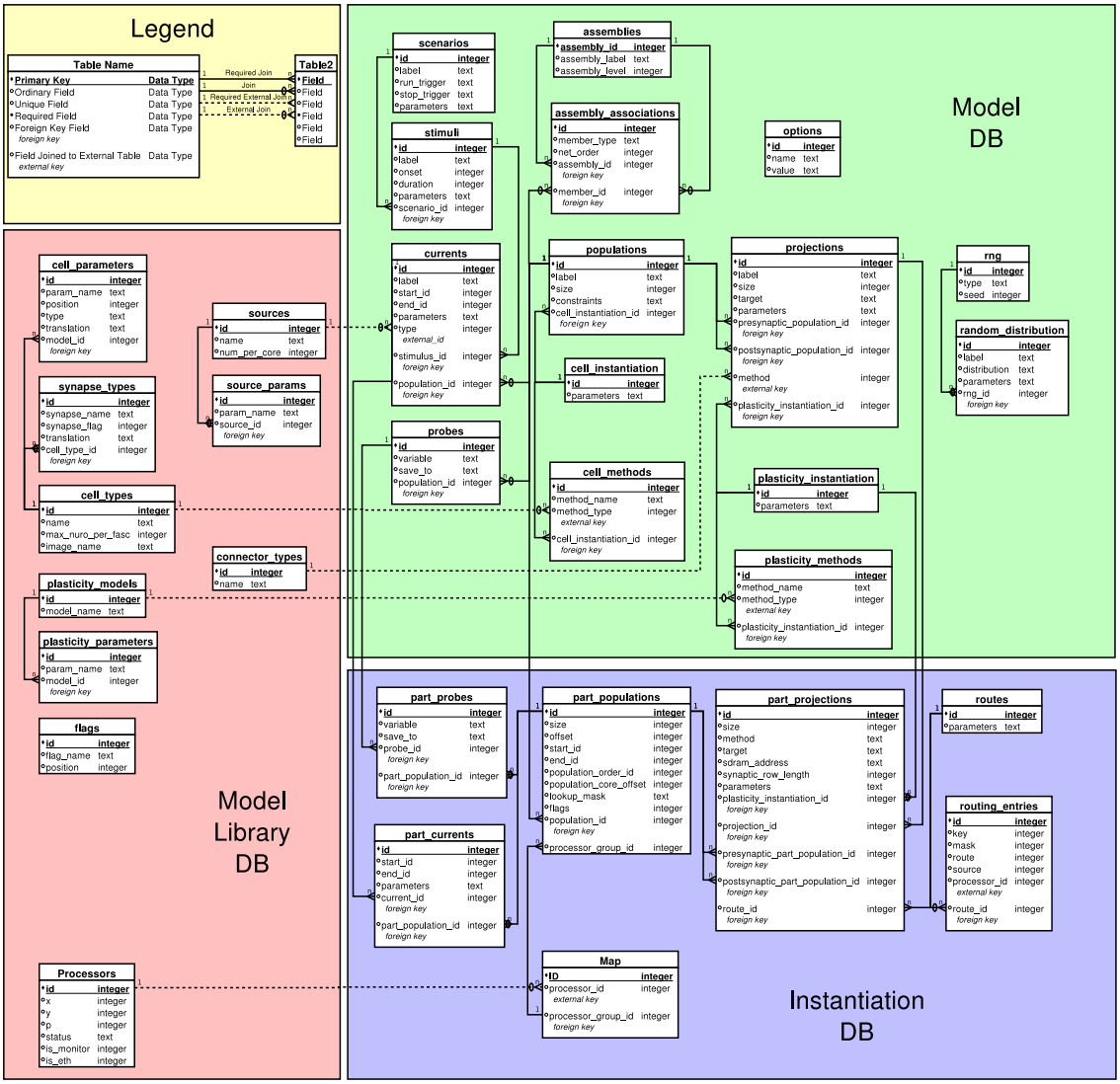


Figure 3.3.12: Updated PACMAN database schema for Lens support

4) Splitter: An existing PACMAN component. The Splitter has been slightly modified to handle arbitrary population splits.

5) MLP premapper plugin: A new PACMAN component that performs a preprocess prior to mapping, to constrain locations of processors.

6) Mapper: An existing PACMAN component. The Mapper has been modified to support relative as well as absolute constraints, e.g. it is possible to specify that a given population be mapped onto a core on the same chip as another, or at a fixed chip displacement, rather than requiring an absolute chip address.

7) Router: An existing PACMAN component, heavily modified. The original Router was part of the Binary File Generator and strongly relied on an expected population mapping for spiking networks. The new version dispenses with this coupling, allows for arbitrary key/mask mappings to population numbers, and extends the database schema to store routing information.

8) Binary File Generator: An existing PACMAN component, completely rewritten. The new Binary File Generator takes a template which specifies how to build a given data binary from fields in the database, and then builds the necessary binary. Although the Router has been moved into a separate module, the generator continues to handle the generation of the physical routing tables.

Front-end translator

This component contains a parser for Lens scripts and a tcl interceptor for the PACMAN commands. The parser - a pair of auxiliary utilities, LensScan.tcl and LensParse.tcl generated by Ylex and Yeti respecfively, accepts the fully-substituted original command as pre-processed by tcl. The interceptor - MLP_PACMAN.tcl passes commands to the translator after they have been fully substituted by tcl. Only PACMAN-relevant commands are passed to the translator; the remainder are passed back to the containing namespace (either Lens or the main tcl namespace, depending upon the execution context). MLP_PACMAN.tcl contains tcl functions for the commands returned by the parser which then interface to the PACMAN database. The following commands are currently supported. (Also see Lens documentation for more on command syntax)

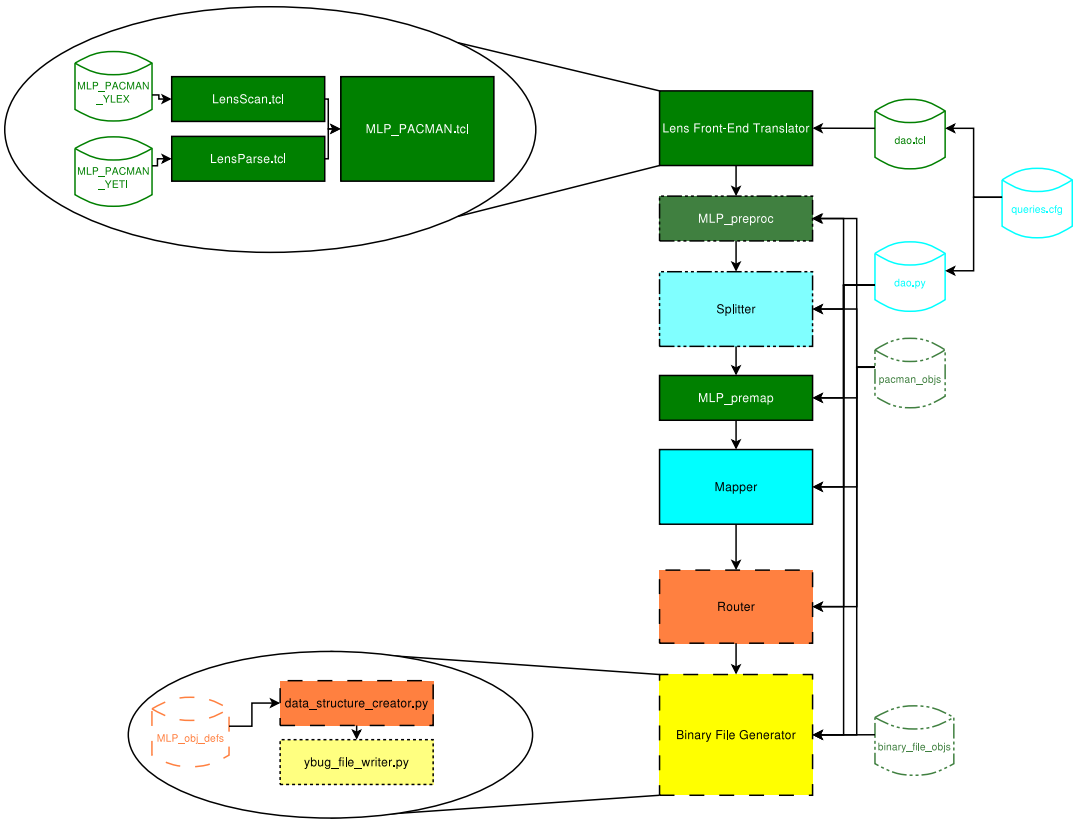


Figure 3.3.13: PACMAN Architecture including MLP extensions

Front-end translator		
Command	args (**required**/ optional)	description
addNet	**name** intervals ticks type groupList	Add a network. A groupList per Lens may instantiate groups as well.
addGroup	**name size** groupType	Add a group (population). groupTypes per Lens specify an extensive range of possible types.
setTime	intervals ticks history dtfixed	Sets Lens time parameters
connectGroups	**sources** intermediates **targets** connectortype strength mean range linktype bidirectional	Add a connection (projection). Following Lens standards a linked chain may be created in one command between sources, any number of intervening populations in intermediates, and final targets.
randWeights	group unit mean range type	Initialises weight randomisations. Parameters allow various subsets of weights to be randomised with various parameters
train	num_updates report algorithm setOnly	Configures training. If setOnly is applied the network will NOT be set to run automatically after being built.
test	num_examples noreset	Configures testing (runs the test set). The *return* option is not supported.
seed	seed	Seeds random number generators.
setObject	**name** value	Sets any Lens-configurable object. Supports the various flavours of Lens object references.

MLP Preprocessor

This is a PACMAN preprocessor for MLP networks, that transforms the top-level Lens representation into an internal representation suitable for splitting and mapping. It performs 2 main tasks: 1) splitting populations into weight, sum and threshold subpopulations, creating the necessary intermediate projections in both the forward and backward direction, and 2) computing source and target populations for the Splitter when projections are split. The preprocessor creates one-to-one connections in the forward and backward direction between the weight, sum, and threshold subpopulations in a population, creates a backward connection from weights to sums of the population's source groups (the sources of its forward projections) in the backpropagation direction, and generates forward and backward sync connections from weights to thresholds (fig. 3.3.14). No weight part population in the forward direction, and thus no sum population in the backward, will be assigned a subrange of the thresholds that corresponds to multiple populations. (It is possible, for example, for a given population to be connected to several populations in previous layers. This would result in the created Weight population having several different projections. However, the preprocessor ensures that the split will create part populations whose projections are asso-

ciated with a specific population at both the presynaptic and postsynaptic terminals.) The preprocessor contains the following externally-visible functions:

MLP preprocessor		
Command	args (**required**/ optional)	description
split_pop_function	None	Reads the database and splits populations into Weight, Sum, and Threshold populations.
set_wt_max_units	**population**	Compute the maximum dimensionality of a Weight population. The function generates square matrices of forward and backward indices.
set_sum_max_units	**population**	Compute the maximum size of a Sum population. For most cases this will be large - considerably larger than the dimensionality of any other population.
set_threshold_max_units	**population**	Compute the maximum size of a Threshold population.
get_post_wt_part_pops	**db projection presynaptic populations**	Identify a subset of the eligible postsynaptic part_populations to be used to connect to the available presynaptic populations of a given projection. This function computes the backpropagation indices for weight units as well as selecting the weight part populations to use for a given projection.
get_pre_input_subrange	**db projection postsynaptic population**	Computes the source part_populations for a given target part_population in a projection.

The MLP preprocessor splits groups into Weight, Sum, and Threshold populations. It then expands the projections into forward and backward projections, adding sync projections as well. When the Splitter divides this pre-processed network, it will ensure that any given part_population projects to only one given associated part_population in any direction

pacman_objs

PACMAN_objs.py adds a standard object interface to the SQLite db, so that the major tables, queries, etc. can be manipulated as objects with defined access and modification methods. It contains the following objects:

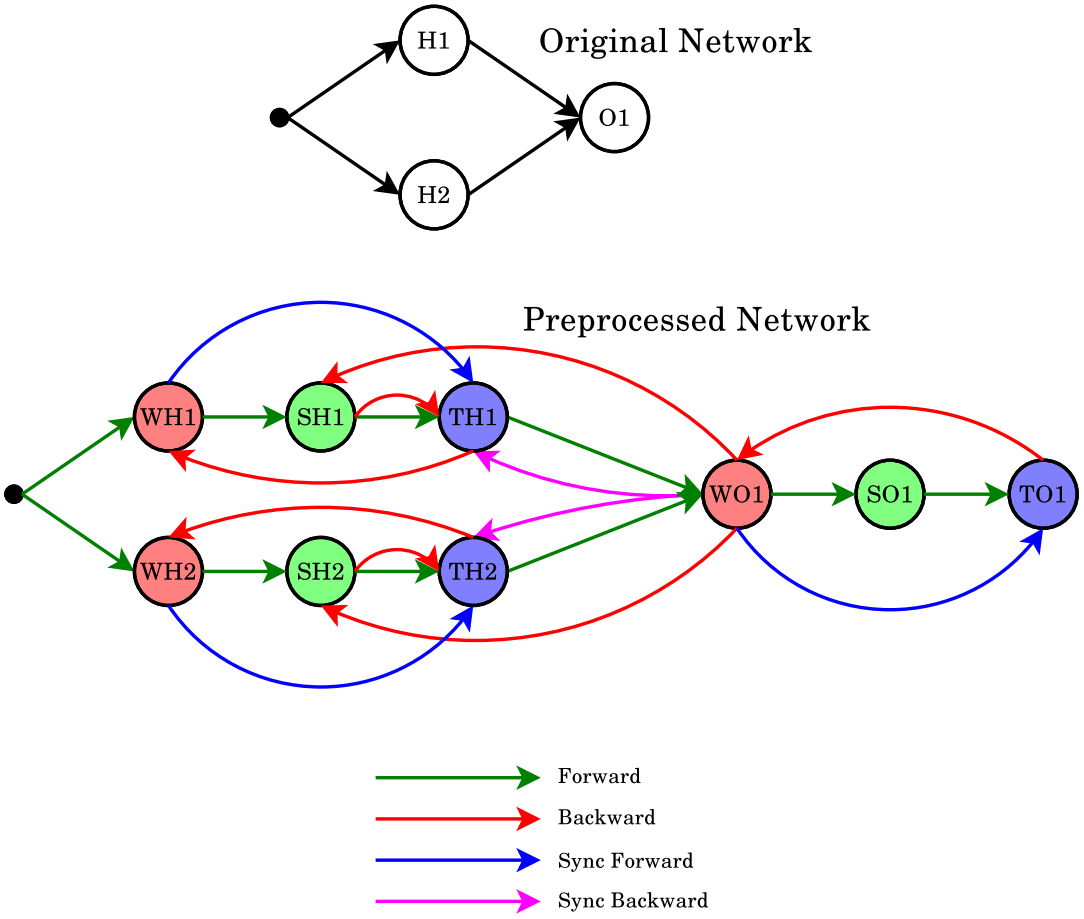


Figure 3.3.14: Mapping of the MLP preprocessor

pacman_objs		
Object	initialiser args (**required**/ optional/ **multiple args)	description
_db	path db_file	Top-level object to represent a database. Initialises basic data access using SQLite.
db_obj	db id **fields	A single object, corresponding to a single query or table row. A db_obj can accept an arbitrary field specification for the row.
db_query	db rows qclass expression **args	A query view, essentially a list of db_objs with usual iterator and access protocols. qclass gives the class of the row object. expression identifies the query, which can either be a function (e.g. standard functions in dao.py) or a literal SQL expression. args accepts any number of parameters for expression. Inherits from _db.
<table object>	db id <field list>	Each PACMAN table has a convenience object by its own name. It takes initialisation parameters which are the fields of the object. Inherits from db_obj.

db_obj supports the following methods:

db_obj		
Method	args (**required**/ optional/ **multiple args)	description
copy	obj	Create a copy of the object. If obj is specified, the function will copy the specified object rather than the instance referenced. Typically the obj argument would only be specified if this was being called as an unbound method.
insert	obj	Inserts the object into the database. The obj argument, here as in other functions, operates similarly to that in the copy method.
delete	None	Removes the object from the database.
update	obj	Updates the fields in the database object. The function assumes that the current values of the object specify the update values.
get	id	Retrieve the object from the database whose id is given. Usually this will be called as an unbound method.

and db_query supports the following methods:

db_query		
Method	args (**required**/ optional/ **multiple args)	description
append	obj	Merges two db_queries, creating a joint list. The object types must match.
insert	query	Runs a bulk-update insert query. Inserts all the rows specified by the insert query into the database. If no query is specified the function inserts all the rows present in the current query instance.
get	qclass expression **args	Runs the query specified by expression with parameters args in order to generate rows, with class qclass. This is the main method of db_query

Splitter

The Splitter reuses, as much as possible, the existing Splitter from the first PACMAN version. The major change is the addition of a case to test for split sources and split targets in projections, so that e.g. weight part populations, which must be associated with a specific source population as well as target population, are properly split. The Splitter detects these restrictions in the constraints field of the Populations table.

MLP Premapper

MLP_premap is a simple PACMAN plugin, that allows for chip-relative constraints. This is necessary for Weight and Sum part_populations, which need to be placed on the same chip, within a population, but which chip it is does not need to be specified. The plugin extends the constraint syntax with a 'rel' dictionary entry that permits the chip/core values to be specified by population id rather than by physical core location.

Mapper

Like the Splitter, the Mapper reuses as much as possible extant PACMAN code. Note that in the MLP implementation at present, there is no Grouper; one might be implemented at a later date to collect weight part populations with contiguous indices in both directions, and likewise Sum units, but for the moment it was felt this is an unnecessary refinement applicable only for certain probably fairly exotic situations. The obvious change in the Mapper is the addition of a test for relative constraints, with appropriate mapping logic. It will be noted that such a modification is quite general and not limited to MLP-style networks; any network may contain a relative chip constraint which the Mapper can then handle. As implemented, the Mapper handles absolute (chip-specific) constraints first, then relative constraints, then any unconstrained Populations and Projections.

Router

Binary File Generator

This consists of 2 main components, the Data Structure Creator and the ybug File Writer. The ybug File Writer follows the pattern of reuse of existing components. The Data Structure Creator is a completely rewritten component. A general object interface, binary_file_objs.py, provides a model-independent engine for the generation of binary files. This reads a configuration file that provides the data generation specification for a given model - in this case for the MLP. Each specification may have up to 5 sections: model_global, chip_common, core_common, element_specific, and component_specific, each of which gives the specification for data blocks at the indicated scope. It is expected that element_specific structures will be laid out in arrays within a core's DTCM, while component_specific structures will be arrays of structures within SDRAM. The specification file should also provide a path to a function file that contains the functions needed to generate a given data object from a given series of PACMAN database rows. data_structure_creator itself contains the generator functions for each of the sections (e.g. gen_chip_common()) that read the specification file, interrogate the database, and then generate the packed data structures.

3.3.7 Coding guidelines

SpiNNaker software is written in C, ARM assembly and Python. Style guidelines are suggested here to help make code easily readable and therefore, hopefully, more easily maintainable. Except where noted these guidelines are soft and should be broken where it is sensible to do so, especially where the reason given for each style point does not apply.

3.3.7.1 All languages

Comments: A Robodoc comment (see section 3.3.8) should be written documenting the purpose of each file and function. Further comments within functions may be useful to describe certain complex operations. **Comments must be kept up to date.**

Identifiers: File, function and variable name should be written in lower case with words separated by underscores to make it easy to recall or guess items in the namespace. Descriptive (potentially verbose) identifiers should be used to make their purpose clear.

Indentation: Code should be indented with 4 spaces per indentation level. **Tabs and spaces must never be mixed** as this quickly makes code completely unreadable when viewed in different editors.

Line length: TODO discuss…?

3.3.7.2 C

Consistency: Styles for C-like languages vary widely so consistency within a function, file and project (in that order of importance) may be the best approach to maintaining

readability. When writing or modifying code, read a little of the existing program to get an idea of the style before beginning.

Compilation: Makefile rules should include both the .c and all `#included` .h files for each target. Also, **.h files must not `#include` other .h files**. This ensures correct recompilation behaviour on calling `make`.

Example: An example of code in the Application Programming Interface is provided:

```
uint dma_transfer(uint tag, void *system_address,
                  void *tcm_address, uint direction, uint length)
{
    uint cpsr = irq_disable();
    uint id = 0;

    if((dma_queue.end + 1) % DMA_QUEUE_SIZE != dma_queue.start)
    {
        id = dma_id++;

        dma_queue.queue[dma_queue.end].id = id;
        dma_queue.queue[dma_queue.end].tag = tag;
        ...
    }

    ...
}
```

3.3.7.3 ARM assembly

Comments: Assembly code should be commented in detail, in some cases with one comment per line in addition to the required Robodoc comments to help readers follow the code. It can also be useful to regularly summarise the content of each working register.

Commenting-out: When commenting out lines of code, do so with a comment characters immediately preceding the instruction rather than at the start of the line. For example:

```
        ;;This is clear:
        ADD     r0, r1, r2
        ;;SUB     r3, r4, r5
        MUL     r0, r1, r2

        ;; This isn't clear:
        ADD     r0, r1, r2
;;       SUB     r3, r4, r5
        MUL     r0, r1, r2
```

Please use two semicolons for Robodoc's sake.

Indentation: Two indentations before opcodes leaves room for labels. One indentation between opcodes and operands leaves enough room for long instructions. It is often easier to read the code when opcodes and operands line up. An example:

```
label   ADD     r0, r1, r2
        STMFDNE sp!, {r4-r9}
```

```
        MULEQ   r0, r1, r2
```

3.3.7.4 Python

General: Code should adhere to the Python style guide which is intended to make the language consistently readable. Note that some style (such as indentation practice) is enforced by the interpreter but the guide is otherwise flexible. See `http://www.python.org/dev/peps/pep-0008/`.

3.3.8 Documentation guidelines

Every file should include an information header formatted for Robodoc. Moreover, where appropriate, each function, class, or section of code should be documented using a template that Robodoc can convert in documentation.

Each programming language has its own format for comments so here we define an header format different for each programming language used in this project. However in all the documentation headers there are several keywords in use with the syntax $keywords$ (words between $ signs). These keywords are substituted by the svn repository with the appropriate value.

3.3.8.1 C / C++

The following templates are for C/C++ source code and header files

File header documentation template

```
/****a* filename.extension/filename
*
* SUMMARY
*  abstract
*
* AUTHOR
*  author - email
*
* DETAILS
*  Created on       : creation date
*  Version          : $Revision: 1226 $
*  Last modified on : $Date: 2011-07-01 11:27:06 +0100 (Fri, 01 Jul 2011) $
*  Last modified by : $Author: plana $
*  $Id: docguide.tex 1226 2011-07-01 10:27:06Z plana $
*  $HeadURL: https://solem.cs.man.ac.uk/svn/spinnSoft_design_doc/docguide.tex $
*
* COPYRIGHT
*  Copyright (c) The University of Manchester, 2010-2011. All rights reserved.
*  SpiNNaker Project
*  Advanced Processor Technologies Group
*  School of Computer Science
*
*******/
```

The substitutions operated by the svn repository are of the type described in the table:

$Revision$	$Revision: 1097 $
$Date$	$Date: 2011-06-02 15:37:34 +0100 (Thu, 02 Jun 2011) $
$Author$	$Author: plana $
Id	$Id: docguide.tex 1097 2011-06-02 14:37:34Z plana $
$HeadURL$	$HeadURL: file:///home/amu4/spinnaker/svn/spinnSoft_design_doc/docguide.tex $

Function documentation template

The following header should be used to describe each of the C/C++ functions:

```
/****f* filename/functionName
*
* SUMMARY
*  abstract
*
* SYNOPSIS
*  function prototype
*
* INPUTS
*  parameter: description
*
* OUTPUTS
*  value
*
* SOURCE
*/

FUNCTION CODE

/*
*******/
```

"FUNCTION CODE" indicates where the function should be written. Doing so, the code will appear also in the documentation generated automatically by Robodoc, with the links to other functions.

Structure documentation template

```
/****s* filename/structureName
*
* SUMMARY
*  abstract
*
* FIELDS
*  variable: description
*
* SOURCE
*/

STRUCTURE CODE

/*
*******/
```

"STRUCTURE CODE" indicates where the function should be written.

3.3.8.2 Assembly language

The following templates are for assembly language source code files

File header documentation template

```
;;****a* filename.extension/filename
;;*
;;* SUMMARY
;;*   abstract
;;*
;;* AUTHOR
;;*   author — email
;;*
;;* DETAILS
;;*   Created on        : creation date
;;*   Version           : $Revision: 1226 $
;;*   Last modified on  : $Date: 2011-07-01 11:27:06 +0100 (Fri, 01 Jul 2011) $
;;*   Last modified by  : $Author: plana $
;;*   $Id: docguide.tex 1226 2011-07-01 10:27:06Z plana $
;;*   $HeadURL: https://solem.cs.man.ac.uk/svn/spinnSoft_design_doc/docguide.tex $
;;*
;;* COPYRIGHT
;;*   Copyright (c) The University of Manchester, 2010-2011. All rights reserved.
;;*   SpiNNaker Project
;;*   Advanced Processor Technologies Group
;;*   School of Computer Science
;;*
;;*******
```

The substitutions operated by the svn repository are of the type described in the table:

$Revision$	$Revision: 1097 $
$Date$	$Date: 2011-06-02 15:37:34 +0100 (Thu, 02 Jun 2011) $
$Author$	$Author: plana $
Id	$Id: docguide.tex 1097 2011-06-02 14:37:34Z plana $
$HeadURL$	$HeadURL: file:///home/amu4/spinnaker/svn/spinnSoft_design_doc/docguide.tex $

Function documentation template

The following header should be used to describe each of the assembler routines:

```
;;****f* filename.extension/functionName
;;*
;;* SUMMARY
;;*   abstract
;;*
;;* SYNOPSIS
;;*   function prototype
;;*
;;* INPUTS
;;*   register: description
;;*
;;* OUTPUTS
;;*   register: value
;;*
;;* SOURCE
;;*
```

FUNCTION CODE

```
;;*
;;*******
```

"FUNCTION CODE" indicates where the function should be written. Doing so, the code will appear also in the documentation generated automatically by Robodoc, with the links to other functions.

3.3.8.3 Robodoc configuration file

Robodoc is an automated documentation generator. It needs some information on how to interpret appropiately the source files to extract the relevant documentation. These information are passed to Robodoc through the configuration file "robodoc.rc" which must reside in the root folder of the project. The output will be stored in the "doc" folder. The following configuration file allows Robodoc to interpret both the C/C++ files and assembler files. However, since there is a usage clash for semicolon in C/C++ and assembler source code, assembler code must use for comments a double semicolon ";;" to start.

```
#robodoc.rc
#
items:
    NAME
    FUNCTION
    SUMMARY
    SYNOPSIS
    INPUTS
    OUTPUTS
    AUTHOR
    COPYRIGHT
    SOURCE
    SEE_ALSO
    NOTES
    TODO
item_order:
    NAME
    FUNCTION
    SUMMARY
    SYNOPSIS
    INPUTS
    OUTPUTS
    AUTHOR
    COPYRIGHT
    SOURCE
    SEE_ALSO
    NOTES
    TODO
source_items:
    SOURCE
options:
    --src ./
    --doc ./doc
    --html
    --multidoc
    --index
    --tabsize 4
    --toc
    --syntaxcolors
    --nogeneratedwith
    --documenttitle "SpiNNaker - documentation"
    --source_line_numbers
    --nosort
headertypes:
    a   "Summary"           robo_summary
```

```
ignore files:
    .svn
    *.txt
    *~
accept files:
    *.c
    *.h
    *.s
header markers:
    /****
    ;;****
remark markers:
    *
    ;;*
end markers:
    ****
    ;;****
remark begin markers:
    /*
remark end markers:
    */
source line comments:
    //
    ;;
keywords:
    if
    else
    do
    while
    for
    return
    void
    unsigned
    short
    int
    uint
    const
    char
    #define
    #if
    #elif
    #endif
```

Part 4

Benchmarks

4.1 Overall goals

Benchmarking of neuromorphic hardware puts numbers on performance to allow measuring progress and comparing different designs. This is useful both for the developers of the Neuromorphic Computing Platform and for potential users.

Specifically, benchmarks define a set of reference tasks aiming at a direct comparison of different neuromorphic (and non-neuromorphic) hardware systems. The benchmarks are coming with a set of quality measures. It is left to to user to decide whether specific measures are relevant for the particular application in mind.

4.2 Quality criteria for neuromorphic benchmark tests

Once benchmark tasks are defined, it is essential to have quality criteria that can be used to evaluate the performance. In traditional computing, the number of floating point operations per second (FLOPS) in performing a standard set of tasks was established as a quality criterium for high performance computers. This well know benchmarking procedure led to the establishment of the TOP500 list of supercomputers which, although often criticized, is recognized by computer manufacturers and their customers. During recent years, energy consumption of computing became a major concern. This led to the establishment of of the TOP GREEN500 list which uses a FLOPS per Watt (or FLOP per Joule) metric.

The following list of quality criteria are proposed for neuromorphic systems:

- Energie usage for a fundamental operation;
- Computational resource usage (neurons, synapses, transistors;
- Silicon area or volume;
- Execution time for specific task
- Number or events/ spikes processed per second
- Time configure / upload a network
- Precision of the solution compared to a software code
- Trial-to-trial reproducibility of the result
- Robustness against hardware mismatch

The quality criteria need to be related to benchmark tasks.

4.2.1 What units should be benchmarked?

With regard to neuromorphic hardware, it was realized that benchmarking of neuromorphic circuits needs to target different components and levels, from individual neurons and synapses, to simple and more complex networks and multi-network architectures.

The most ambitious benchmarking scenarios approach definitions of real scientific challenges like the ten listed by Stanislas Dehaene during his plenary talk at the HBP kickoff meeting.

The numbers that come out from the benchmarking can concern:

- how biomimetic components are ("neuronicity", "synapticity");
- how brain-like its network architecture is;
- what functions it can perform;
- and at what level of performance in terms of solution of the task, speed, and energy expenditure.

Also we could compare state variables' time courses with software simulation, also for performance level, for instance number of correctly retrieved patterns in a simple storage capacity measurement, the number of correctly classified items in a classification task, or the progress of learning in a reinforcement type of task.

It is important to match the model properties to what it will be used for. In many cases a simple or reduced model may be sufficient to replicate biological and dynamic phenomena as well as task performance. But sometimes there is a need for a high degree of detail in the neuron and synapse models to capture phenomena seed in experiments. Thus, a hardware that is unable to reproduce the latter might still be very useful for other purposes. So it is not at all necessary to "pass" all the benchmarks discussed below. The benchmark suite should rather be seen as a way to quantitatively characterize the capabilities of the hardware.

The method will to a large extent be to compare output from hardware runs with the corresponding simulations using software. To the extent that the hardware typically implements some kind of mathematical neuron and synapse models, this is quite straight forward. On the other hand, digital hardware implementations may use lower precision computation and analog hardware has intrinsic noise which may or may not be of a similar nature as in the biological system. Therefore, it may occasionally be motivated to compare the hardware directly with data from the relevant biological components and systems.

4.3 Use cases

4.3.1 Tracking the performance of a neuromorphic computing system over time

Primary actor Alice, a neuromorphic system developer.

Description Over time, as new versions of the neuromorphic hardware and associated software are developed, Alice wishes to determine how the new versions affect the performance of the system, according to several measures, including throughput (how many jobs of a given complexity can be run on the system in a given time), power consumption, and accuracy (how closely the output of the neuromorphic systems matches the expected behaviour.)

Success scenario

1) Alice selects a number of tasks from a library of benchmark tasks.

2) For each task, she runs a job on the Neuromorphic Computing Platform, with careful instrumentation of the time required for different stages and of any discrepancies or errors produced.

3) She compares the numerical measures she obtains to previous runs of the same benchmarks.

4.3.2 Determining whether the Neuromorphic Computing Platform is suitable for a specific task

Primary actor Boris, a computational neuroscientist.

Description Boris has a model that runs on the HPC Platform, and which he would like to run an adapted version of the model on the Neuromorphic Computing Platform. Before taking the time to adapt the model, Boris wants to be sure that the adaptation is likely to be successful.

Success scenario

1) Boris searches the library of benchmark tasks to find the benchmarks that have features in common with his model.

2) By examining the records of previous runs of these benchmarks, he determines that the expected discrepancies between hardware and numerical simulations are unlikely to affect the qualitative behaviour of his model.

4.4 Functional requirements

1) The benchmark library/database shall contain benchmarks to examine:
   a) the behaviour of individual neurons;
   b) post-synaptic responses of individual synapses;
   c) effects of tranmission delays, and discrepancies between the nominal (requested) and actual distribution of delays;
   d) the accuracy and correctness of synaptic plasticity implementations;
   e) microcircuit behaviour;
   f) the capabilities of the system as a whole.
2) Each benchmark task shall produce one or more numerical measures.
3) Such measures may include:
   a) how closely the neuromorphic system matches the results of numerical simulations;
   b) how well the neuromorphic system performs a certain computational task;
   c) how long the neuromorphic system takes to complete a certain task;
   d) how closely a given model can be mapped to the neuromorphic circuits;
   e) how large is the impact of discrepancies between numerical and neuromorphic models;
   f) the energy expenditure required to complete a certain task.
4) except where physical access to the hardware is absolutely required, all benchmarks shall be automatable, able to run without direct user intervention.
5) the results of benchmark runs shall be stored in a database so as to allow comparisons across benchmark tasks and across time.
6) for each run of the benchmarks, the exact state of the system (software and hardware versions) shall be recorded.

4.5 Architectural overview

- In the first stage of benchmark development, each benchmark will be implemented as a self-contained Python script, using either the PyNN API or one of the lower-level, hardware-specific APIs as appropriate.
- In later stages of development, it may be desirable to implement a framework to make it easier to implement new benchmarks by taking care of common functionality and eliminating boilerplate code.
- Each benchmark script should be tracked using version control.
- Each benchmark script should be registered with a central registry. This could be as simple as a version-controlled text file containing the URL of each benchmark script, or could be a more full-featured system making use of a relational database.
- All benchmarks should write the numerical output measures to file using a standardized format (e.g. JSON, XML).
- The numerical output measures could also be stored in a relational database, allowing faster and more sophisticated queries.

Part 5

Following the platform building: Key Performance Indicators and time plans

5.1 KPIs and time plans

5.1.1 KPIs of the NMPM

5.1.1.1 Wafer Production

Value	Status values	Target
Wafers	ordered at UMC received from UMC post-processing started post-processing finished mounted into Wafer Module, contact tests finished	M18: 20 mntd
	operational (complete defect map available)	M30: 20

5.1.1.2 Printed Circuit Board Production

Explanation of status values that are used for PCB KPIs:

- Ordered: A prototype has been tested and the design has been signed off for production. A manufacturer has been selected and an order for fully assembled PCBs has been placed there.
- Manufactured: PCBs have been produced, assembled and received from the manufacturer. Bare PCBs have passed electrical tests and are assumed error-free. Assembled PCBs have passed visual inspection by the manufacturer.
- Tested: Functional tests of the assembled PCB have been completed and it is ready for usage in NM-PM1.

Wafer Module Main PCB Production

Due to its complexity, the MainPCB will be assembled by a company that is different from the PCB manufacturer. This gives an additional status value.

Value	Status values	Target
MainPCBs	bare PCBs ordered bare PCBs manufactured PCBs assembled tested	M18: 20 tested

Monitoring and Control PCB Production

Value	Status values	Target
Cure boards	ordered manufactured tested	M18: 160 tested

FPGA Communication PCB Production

Value	Status values	Target
FCPs	ordered manufactured tested	M18: 960 tested

PowerIt Main Power Supply PCB Production

Value	Status values	Target
PowerIts	ordered manufactured tested	M18: 20 tested

Auxiliary Power Supply PCB Production

Value	Status values	Target
AuxPwrs	ordered manufactured tested	M18: 40 tested

Analog Readout Components

Value	Status values	Target
Flyspis	ordered manufactured tested	M18: 60 tested

Value	Status values	Target
AnaFPs	ordered manufactured tested	 M18: 60 tested

Value	Status values	Target
AnaRMs	tested and mounted into NM-PM1	M18: 60

5.1.1.3 Wafer Module Production

Mechanical components

Value	Status values	Target
Mech. components for one Wafer Module	material delivered manufactured electrosilvered and coated	M18: 20 electrosilvered and coated

Wafer Modules

Value	Status values	Target
Wafer Modules	all components delivered assembled integrated into server racks	 M18: 20 integrated
	operational	M30: 20

5.1.1.4 Software and Hardware Usage KPIs

Value	Range	Target
Code coverage of hardware abstraction layers	0–100 %	M18: 100 %
Code coverage of calibration toolchain	0–100 %	M30: 100 %
Code coverage of frontend and mapping layer	0–100 %	M30: 100 %
Func. coverage of hardware abstraction layers	0–100 %	M18: 100 %
Func. coverage of calibration toolchain	0–100 %	M30: 100 %
Func. coverage of frontend and mapping layer	0–100 %	M30: 100 %
Number of defect maps available (1/wafer)	0–20	M30: 20
Wafers available for PyNN users	0–100 %	M30: 70 %
Number of calibration routines for hardware model parameters	0–15	M30: 15
Number of neural network experiments exec'd	Count	

5.1.2 KPIs of the NMMC

5.1.2.1 Cabinet Assembly

Value	Status values	Target
Cabinet (47U)	ordered received assembled tested operational	M18: 5 assembled

5.1.2.2 Sub-rack assembly

Value	Status values	Target
6U sub-rack	ordered received assembled tested operational	M18: 25 assembled

Value	Status values	Target
Card guides	ordered received assembled tested operational	M18: 1200 assembled

Value	Status values	Target
Backplane PCB	ordered received assembled tested operational	M18: 75 operational

Value	Status values	Target
Spin5 PCB	ordered received assembled tested operational	M18: 600 operational

Value	Status values	Target
SpiNNaker chip	ordered received assembled tested operational	 M18: 28800 tested M30: 28800 operational

Value	Status values	Target
SATA cables	ordered received assembled tested operational	 M18: 1800 operational

Value	Status values	Target
Mains cables	ordered received assembled tested operational	 M18: 100 operational

5.1.2.3 Network

Value	Status values	Target
Switch – Netgear FS726T	ordered received assembled tested operational	 M18: 25 operational

Value	Status values	Target
Network cables	ordered received assembled tested operational	 M18: 625 operational

5.1.2.4 Fan Tray Assembly

Value	Status values	Target
Fan tray metalwork	ordered received assembled tested operational	M18: 25 operational

Value	Status values	Target
120mm fan	ordered received assembled tested operational	M18: 150 operational

Value	Status values	Target
Display module	ordered received assembled tested operational	M18: 25 operational

5.1.2.5 Power Supply Assembly

Value	Status values	Target
Power supply unit (650W)	ordered received assembled tested operational	M18: 75 operational

Value	Status values	Target
Power supply panel	ordered received assembled tested operational	M18: 25 operational

5.1.3 KPIs of the common software part

The UI function blocks are described in chapter 1.7 (page 45). The implementation of the defined blocks will be followed as KPI.

5.1.4 KPIs of the benchmark part

- M12: An initial suite of 4 benchmarks fully specified and defined as Python scripts, one for each level of neuron, synapse, microcircuit, and network. Initial tests of these on NM-PM1 (verified via ESS) implemented in PyNN and on NM-MC1 performed.
- M18: 2x4 benchmarks of each type is specified as PyNN scripts and entered into the repository. 3 complete benchmark runnable on NM-PM1 (verified via ESS) implemented in PyNN and on NM-MC1. 1 benchmark successfully run on NM-PM1 and 2 different on NM-MC1, with results entered into the benchmark database.
- M30: 2x3 benchmarks successfully run on NM-PM1 (verified via ESS) and on NM-MC1. The repeated 3 used to observe and verify improvements from M18.

Targets for benchmarks for Neuron | Synapse | Microcircuit | Network:

Value	Status values	Target
Benchmark	fully specified and defined in PyNN	M18: 2 each
	initial tests on NM-PM1 (ESS) and NM-MC1 performed	M12: 1 each
	complete bench. runnable on NM-PM1 (ESS) + NM-MC1	M18: 3
	benchmark successfully run on NM-PM1	M18: 1, M30: 3
	benchmark successfully run on NM-MC1	M18: 2, M30: 3

Bibliography

[1] IEEE Standard Test Access Port and Boundary-Scan Architecture. IEEE Std 1149.1-2001, pages i–200, 2001.

[2] Maxim DS18B20 Datasheet, 2008.

[3] Open Core Protocol Specification 2.2, 2008.

[4] Texas Instruments PCA9544A datasheet, 2008.

[5] Vishay Siliconix Si7234DP datasheet, 2008.

[6] Vishay Siliconix SiA912DJ datasheet, 2008.

[7] Vishay Siliconix Si5903DC datasheet, 2010.

[8] Microchip dsPIC33FJ128GP710A datasheet, 2012.

[9] Intel NUC. `http://www.intel.com/content/www/us/en/nuc/overview.html`, 2014.

[10] Raspberry Pi. `http://www.raspberrypi.org`, 2014.

[11] C. Adams, S. Farrell, T. Kause, and T. Mononen. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP), September 2005.

[12] W. Barth. Nagios: System and Network Monitoring. No Starch Press Series. No Starch Press, 2008.

[13] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. 2008.

[14] Jose Luis Blanco. nanoflann Website. `http://code.google.com/p/nanoflann/`, 2013.

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

[16] Daniel Brüderle, Eric Müller, Andrew Davison, Eilif Muller, Johannes Schemmel, and Karlheinz Meier. Establishing a Novel Modeling Tool: A Python-based Interface for a Neuromorphic Hardware System. Front. Neuroinform., 3(17), 2009.

[17] Daniel Brüderle, Mihai Petrovici, Bernhard Vogginger, Matthias Ehrlich, Thomas Pfeil, Sebastian Millner, Andreas Grübl, Karsten Wendt, Eric Müller, Marc-Olivier Schwartz, Dan de Oliveira, Sebastian Jeltsch, Johannes Fieres, Moritz Schilling, Paul Müller, Oliver Breitwieser, Venelin Petkov, Lyle Muller, Andrew Davison, Pradeep Krishnamurthy, Jens Kremkow, Mikael Lundqvist, Eilif Muller, Johannes Partzsch, Stefan Scholze, Lukas Zühl, Christian Mayr, Alain Destexhe, Markus Diesmann, Tobias Potjans, Anders Lansner, René Schüffny, Johannes Schemmel, and Karlheinz Meier. A comprehensive workflow for general-purpose neural modeling with highly configurable neuromorphic hardware systems. Biological Cybernetics, 104:263–296, 2011.

[18] James O. Coplien. Curiously Recurring Template Patterns. C++ Rep., 7(2):24–27, February 1995.

[19] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. Front. Neuroinform., 2(11), 2008.

[20] E.W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1(1):269–271, 1959.

[21] György Dósa. The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is FFD(I) <= 11/9 OPT(I) + 6/9. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, volume 4614 of Lecture Notes in Computer Science, pages 1–11. Springer Berlin Heidelberg, 2007.

[22] TU Dresden. DNC Specification. FACETS project internal documentation, 2008.

[23] M. Ehrlich, C. Mayr, H. Eisenreich, S. Henker, A. Srowig, A. Grübl, J. Schemmel, and R. Schüffny. Wafer-Scale VLSI Implementations of Pulse Coupled Neural Networks. In Proceedings of the International Conference on Sensors, Circuits and Instrumentation Systems (SSD-07), March 2007.

[24] John Enck. Ethernet/802.3 and token ring/802.5. pages 265–295, 1994.

[25] Faraday Technology, www.faraday-tech.com. FSA0M_A Faraday Standard Cell Library, 2009.

[26] Faraday Technology, www.faraday-tech.com. FXPLL031HA0A_APGD Faraday Phase-Locked Loop, 2009.

[27] J. Fieres, 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.

[28] Michael L. Fredman and Robert Endre Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. J. ACM, 34(3):596–615, July 1987.

[29] Simon Friedmann. A new approach to learning in neuromorphic hardware. PhD thesis, Heidelberg, Univ., Diss., 2013, 2013.

[30] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1st edition, 1979.

[31] David E. Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[32] Marcel Gort and Jason Helge Anderson. Deterministic multi-core parallel routing for FPGAs. pages 78–86. IEEE, 2010.

[33] IEEE Ethernet Working Group. IEEE 802.3ab. http://www.ieee802.org/3.

[34] David Heeger. Poisson Model of Spike Generation, 2000.

[35] Dan Husmann and Holger Zoglauer. A Wafer-Scale-Intagration System(WSI). FACETS project internal documentation, 2010.

[36] IEEE. IEEE 802.3ak. http://www.ieee802.org/3.

[37] Fujipoly Inc. Silicon Rubber Interface Materials Company. http://www.fujipoly.com, 2013.

[38] Information Technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994, ISO, Geneva, Switzerland, November 1994.

[39] Fraunhofer-Institut für Zuverlässigkeit und Mikrointegration IZM. Fraunhofer IZM. Gustav-Meyer-Allee 25, 13355 Berlin, Germany, http://http://www.izm.fraunhofer.de.

[40] JEDEC. DDR3 SDRAM standard, 2012.

[41] Andrew B. Kahng and Gabriel Robins. A new class of iterative Steiner tree heuristics with good performance. IEEE Trans. on CAD of Integrated Circuits and Systems, 11(7):893–902, 1992.

[42] Vitali Karasenko. A communication infrastructure for a neuromorphic system. Master's thesis (English), University of Heidelberg, 2014.

[43] RichardM. Karp. Reducibility among Combinatorial Problems. In RaymondE. Miller, JamesW. Thatcher, and JeanD. Bohlinger, editors, Complexity of Computer Computations, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[44] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. Science, 220(4598):671–680, 1983.

[45] SchedMD LLNL et al. Simple Linux Utility for Resource Management. http://slurm.schedmd.com, 2014.

[46] M. Massie, B. Li, B. Nicholes, and V. Vuksan. Monitoring with Ganglia. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2012.

[47] Microchip Technology. Microchip EMC1412 Datasheet, 2012.

[48] Sebastian Millner. Development of a Multi-Compartment Neuron Model Emulation. PhD thesis, University of Heidelberg, 2012.

[49] Sebastian Millner, Andreas Grübl, Karlheinz Meier, Johannes Schemmel, and Marc-Olivier Schwartz. A VLSI Implementation of the Adaptive Exponential Integrate-and-Fire Neuron Model. In J. Lafferty et al., editors, Advances in Neural Information Processing Systems 23, pages 1642–1650, 2010.

[50] Melanie Mitchell. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA, 1996.

[51] Rajeev Motwani and Prabhakar Raghavan. Randomized Algorithms. Cambridge University Press, New York, NY, USA, 1995.

[52] NXP Semiconductors. I2C-bus specification and user manual, 2012.

[53] K.E. Parsopoulos and M.N. Vrahatis. Recent approaches to global optimization problems through Particle Swarm Optimization. Natural Computing, 1(2-3):235–306, 2002.

[54] S. Philipp. Generic ARQ Protocol in VHDL. Internal FACETS documentation., 2008.

[55] Jon Postel. RFC 768 User Datagram Protocol. RFC 768, Internet Engineering Task Force, August 1980.

[56] Jon Postel. RFC 791 Internet Protocol, September 1981.

[57] PyNN. A Python package for simulator-independent specification of neuronal network models – Website. `http://www.neuralensemble.org/PyNN`, 2008.

[58] Raspbian. Debian Linux distribution optimized for the Raspberry Pi. `http://www.raspbian.org`, 2014.

[59] J. Schemmel, 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), pages 3367–3370. IEEE Press, 2007.

[60] J. Schemmel, A. Grübl, K. Meier, and E. Muller. Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model. In Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN). IEEE Press, 2006.

[61] S. Scholze, H. Eisenreich, S. Höppner, G. Ellguth, S. Henker, M. Ander, S. Hänzsche, J. Partzsch, C. Mayr, and R. Schüffny. A 32 GBit/s Communication SoC for a Waferscale Neuromorphic System. Integration, the VLSI Journal, 2011. in press.

[62] S. Scholze, S. Henker, J. Partzsch, C. Mayr, and R. Schuffny. Optimized queue based communication in VLSI using a weakly ordered binary heap. In Mixed Design of Integrated Circuits and Systems (MIXDES), 2010 Proceedings of the 17th International Conference, pages 316–320, june 2010.

[63] Stefan Scholze, Stefan Schiefer, Johannes Partzsch, Stephan Hartmann, Christian Georg Mayr, Sebastian Höppner, Holger Eisenreich, Stephan Henker, Bernhard Vogginger, and Rene Schüffny. VLSI implementation of a 2.8GEvent/s packet based AER interface with routing and event sorting functionality. Frontiers in Neuromorphic Engineering, 5(117):1–13, 2011.

[64] Marc-Olivier Schwartz. Reproducing Biologically Realistic Regimes on a Highly-Accelerated Neuromorphic Hardware System. PhD thesis, Universität Heidelberg, 2013.

[65] National Semiconductor. LVDS Owner's Manual. LVDS.national.com, 2004.

[66] Jeremy Siek. Boost Graph Dijkstra's Algorithm Implementation 1.49.0 Website. `http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/dijkstra_shortest_paths.html`, 2001.

[67] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost Graph Library Version 1.49.0 Website. `http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/index.html`, 2001.

[68] Steven Skiena. The Algorithm Design Manual. Springer Verlag, 2nd edition, 2008.

[69] Inc SolidWorks, Dassault Systèmes S.A. 3D-CAD tool. `http://www.solidworks.com`, 2013.

[70] Andrew S. Tanenbaum and David J. Wetherall. Computer Networks. Pearson Education, 2012.

[71] The Human Brain Project. A Report to the European Commission, 2012.

[72] United Microelectronics Corporation (UMC). `http://www.umc.com`.

[73] Bernhard Vogginger. Testing the Operation Workflow of a Neuromorphic Hardware System with a Functionally Accurate Model. Diploma thesis, Ruprecht-Karls-Universität Heidelberg, HD-KIP-10-12, 2010.

[74] Xilinx, Inc. 7 Series FPGAs Overview, v1.15 edition, 2014. DS180.

[75] Xilinx IP Documentation. Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions v2.0 User Guide, 2013.

Glossary

1-wire
1-wire. 182

10GBASE-SR
10GBASE-SR. 177

10GbE
10-Gigabit Ethernet. 55, 175, 176

40GbE
40-Gigabit Ethernet. 55, 175, 176

ADC
Analog-to-Digital Converter. 57, 169, 170, 189, 250, 270
270

AdEx
Adaptive Exponential Integrate-and-Fire. 243, 271
271, 272

AL
Application Layer. 163, 164, 166

AnaB
Analog Breakout PCB. 56, 124, 131, 137, 139, 143, 180

AnaB to FsBo connector
16 pin connector for the connection of Analog Breakout PCB with Flyspi Breakout PCB. 139

AnaB to Rasperry Pi connector
26 pin connector for the connection of Analog Breakout PCB with Raspberry Pi. 139, 144

AnaFP
Analog Frontend PCB. 57, 169, 209, 491

AnaRM
Analog Readout Module. 8, 54, 55, 57, 139, 169, 170, 197, 203, 241, 242, 491

ANNCORE
Analog Neuronal Network Core. 65, 68, 82, 99, 244

API
Application Programming Interface. 59–61, 187, 240, 244, 245, 248, 274, 275

ARQ
Automatic Repeat Request. 86, 88, 90, 157, 160–162, 197, 198, 202, 203, 235, 236

ASIC
Application Specific Integrated Circuit. 53, 56

AuxPwr
Auxiliary Power Supply PCB. 56, 124, 132, 137–139, 179–181, 183, 184, 189, 190, 194, 195, 490

AXI
Advanced Extensible Interface. 157

BCB
Benzo Cyclo Butene. 111, 112

Calibration
Calibration. 60

calibtic
calibtic. 270, 274

CMOS
Complementary Metal-Oxide-Semiconductor. 53, 70, 71

Compute Cluster
A collection of computers interconnected by a dedicated network. 55, 57, 169

Compute Node
A single compute node as part of a Cluster. 56, 61, 176, 240, 247

CRC
Cyclic Redundancy Check. 86, 160

Cure
Monitoring and Control PCB for Reticles. 56, 130, 139, 140, 142, 143, 181–184, 188–190, 194, 195, 490

DAC
Digital-to-Analog Converter. 80, 249

DCON
Direct Wafer-to-Wafer Connections. 65, 68

DDR
Double Data Rate. 151, 152, 156–158, 203

DenMem
Dendrite Membrane Circuit. 80–82, 94, 245, 249

DI_VBias
DI_VBias: LVDS common mode voltage. 181

DI_Vbias
LVDS common mode voltage. 138

DI_VCC
DI_VCC: digital supply of DNC interface. 138

DI_VCC33ana
DI_VCC33ana:LVDS power supply of DNC interface. 138

DI_VCCana
DI_VCCana: analog supply of DNC interface. 138

DLL
Delay-Locked Loop. 71–73, 76

DNC
Digital Network Chip. 119, 120, 160, 239, 256, 257

DNC Interface
Digital Network Chip Interface. 86, 87, 163, 164

DRAM
Dynamic Random Access Memory. 57, 165, 169, 170

DRC
Design Rule Check. 114

DSC
Digital Synapse Control. 102–106, 222–224, 226, 227, 229

ElCo
Elastomeric Stripe Connector. 112, 123, 125–131, 143

ESS
Executable System Specification. 59, 62, 63, 240, 243–245

F-PLL
A Phase-Locked Loop provided by the standard cell vendor Faraday Technology [26]. 87, 88

FC-PLL
A Phase-Locked Loop that has been designed for the HICANN chip during the BrainScaleS project. 87, 88

FCP
FPGA Communication PCB. 54, 56, 65, 86, 88–90, 92, 94, 120, 124, 129, 131, 137, 139, 145, 146, 151–153, 155, 163–165, 175, 176, 180, 183, 184, 190, 195, 197, 240, 241, 247, 490

FIFO
First-In First-Out. 71, 87, 157, 158, 160, 164, 166

Flyspi
Flyspi FPGA PCB. 8, 57, 169, 170, 203, 490

FPGA
Field-Programmable Gate Array. 56, 57, 87, 120, 124, 146, 151–153, 155–162, 164, 169–173, 198–203, 231, 235, 236, 257, 273

FsBo
Flyspi Breakout PCB. 57

FSM
Finite State Machine. 87, 240

GbE
Gigabit Ethernet. 55, 56, 164, 175, 176, 197

GTX
Xilinx Gigabit Transceiver protocol. 151–153, 155, 156, 158

HAL
Hardware Abstraction Layer. 237, 244

HALbe
Hardware Abstraction Layer Backend. 61, 187, 237–245, 270

HDD
Hard disk drive. 56, 175

HICANN
High-Input Count Analog Neuronal Network Chip. 53, 56, 65, 66, 68–72, 74–79, 81, 82, 86–88, 90, 92, 93, 102, 111, 114–120, 129–131, 145, 146, 151–153, 156, 157, 160–164, 179, 180, 189, 194, 195, 199–203, 212, 213, 231, 233, 236–239, 241, 242, 244, 252, 260–264, 267, 270, 274

HICANN Wafer
A 20 cm silicon wafer with 384 HICANN ASICs interconnected by wafer-scale postprocessing. 54, 56, 123

HostARQ
Host ARQ protocol. 162

I/O
Input/Output. 55, 175

I/O Node
A single I/O node as part of a Cluster. 176

I2C
Inter-Integrated Circuit Link. 56, 139, 140, 143, 156, 158, 159, 182–188, 195, 196

InFra
Insertion frame for mounting of additional PCBs. 125, 145

IP
Intellectual Property. 156, 157

IPC
Inter-process Communication. 245, 275

IPv4
Internet Protocol version 4. 197

JTAG
Joint Test Action Group. 86–88, 117, 119, 120, 156, 162, 182, 231, 233–236

KIP
Kirchhoff-Institute for Physics. 177

L0
Layer 0. 68

L1
Layer 1. 68, 69, 71–74, 76, 80, 111, 115, 160, 161, 234, 236, 244, 252, 254–261, 263, 265, 273, 274

L2
Layer 2. 68, 71, 73, 86, 162, 234, 273

LUT
Look Up Table. 160, 161

LVDS
Low-Voltage Differential Signaling. 87, 117, 119, 120, 151, 153, 156, 160, 234

LVS
Layout Versus Schematic. 114

MAC
Media Access Controller. 197

MaCU
Main System Control Unit. 124, 136, 138–140, 143, 182

MainPCB
Wafer Module Main PCB. 56, 65, 111, 119, 120, 124–131, 135–137, 139, 140, 143, 145, 146, 181, 182, 184, 185, 188, 195, 196, 489, 490

MainPCB to AnaB-Master connector A
120 pin fine pitch connector between MainPCB and Analog Breakout PCB Master. 133

MainPCB to AnaB-Master connector B
120 pin fine pitch connector between MainPCB and Analog Breakout PCB Master. 133

MainPCB to AnaB-Slave connector A
120 pin fine pitch connector between MainPCB and Analog Breakout PCB Slave. 134

MainPCB to AnaB-slave connector B
120 pin fine pitch connector between MainPCB and Analog Breakout PCB Slave. 134

MainPCB to Auxilary Power Supply PCB connector
120 pin PC104plus power connector for the auxilary power supply of the MainPCB and the Wafer. 131, 137

MainPCB to Cure PCB connector
Sodimm connector between MainPCB and Cure PCB. 130, 140, 141

MainPCB to FCP connector
120 pin edge card socket between MainPCB and FPGA Communication PCB. 147

MainPCB to Wafer connector pair
connector pair between MainPCB and Wafer. 131

Mapping
Mapping. 60

marocco
marocco. 252

MCU
Microcontroller Unit. 56

MIG
Memory Interface Generator. 157

mongoDB
mongoDB. 271

MPW
Multi Project Wafer. 65

MRST
Minimum Rectilinear Steiner Tree. 262

MTREE
Merger Tree. 71, 72

NIC
Network Interface Controller. 175

NM-PM
Neuromorphic Physical Model. 6, 53, 55, 59–62, 175, 237, 240–245, 247–249, 274

NM-PM1
Neuromorphic Physical Model version 1. 6, 17, 54, 55, 112, 113, 121–123, 169, 176, 177, 197, 257, 489, 491, 511

NMOS
Negative Metal-Oxide Semiconducotr. 71

NP
Non-deterministic Polynomial-time. 262, 265

NUC
Next Unit of Computing. 57, 169

OCP
Open Core Protocol. 88, 92, 171

PCB
Printed Circuit Board. 57, 123, 126, 131, 146, 148, 169, 190, 489, 490

PCIe
Peripheral Component Interconnect Express. 56, 175

PLL
Phase-Locked Loop. 71, 87, 234

PMBus
Power Management Bus. 183, 190

PMk
Positioning Mask for the Elastomeric Stripe Connectors. 123, 126, 127

PMOS
Positive Metal-Oxide Semiconductor. 71

PMU
Power Management Unit. 247

Power-FET
Power Field-Effect Transistor. 56, 129, 181, 182, 184, 188–190, 194

PowerIt
PowerIt Main Power Supply PCB. 56, 124, 136, 137, 179, 180, 183, 184, 189, 194, 195, 490

PrePreg
Carbon fiber reinforced plastic CFRP. 131

PyHMF
Python for the Hybrid Multiscale Facility. 255, 259, 270

PyNN
PyNN. 59–61, 63, 248, 252, 273–275

PyNN.hardware.nmpm
NM-PM backend for PyNN. 9, 274, 275

Python
Python Programming Language. 59, 237, 241, 243, 252, 270

QSFP
Quad SFP. 55, 176

RAM
Random Access Memory. 98, 160, 161

Raspberry Pi
Raspberry Pi. 143, 179, 180, 182–184, 187, 190, 195, 196

RCF
Remote Call Framework. 187

RDMA
Remote Direct Memory Access. 54

RTL
Register Transfer Level. 243

Scheriff
State Checking and Error Identification Framework. 240

SFP+
Small Form-Factor Pluggable. 55, 176

SGMII
Serial Gigabit Media Independent Interface. 153, 155, 157

SimDenMem
HALbe simulation backend for analog circuits. 245

SLURM
Simple Linux Utility for Resource Management. 247

SMS
System Managment Software. 187, 190

SpL1
Synchronous Parallel Layer 1. 73, 74, 99, 102, 239, 254–260, 262–264

SRAM
Static Random Access Memory. 107, 214, 225

SSD
Solid-state Disk. 56, 175

STDP
Spike Timing Dependent Plasticity. 102, 103, 223, 229, 244, 273
273

StHAL
Stateful Hardware Abstraction Layer. 61, 241–245, 270

STP
Short-term plasticity. 266, 268, 269

TAP
Test Access Port. 86

ToCo
Top Cover. 124–127, 143, 145, 184

ToR
Top-of-Rack. 54–56, 176

UDP
User Datagram Protocol. 156, 157, 164–167, 197, 202

UMC
NM-PM1 semiconductor manufacturer: United Microelectronics Corporation UMC [72]. 113–115, 120, 489

USB 2.0
Universal Serial Bus version 2.0. 57, 169, 170, 197

V5_Stby
5 V standby supply voltage for the Wafer Module. 136, 139, 179, 180

V_intermed
intermediate voltage for the Wafer Module (7-13.5 V). 131, 136, 179, 180

V_MainIn
main input voltage of the Wafer Module (-48 V). 136, 179, 190

V_OH
V_OH: upper voltage level for the Layer 1 signaling. 138, 180, 181, 189

V_OL
V_OL: lower voltage level for the Layer 1 signaling. 138, 180, 181, 189

VDD
1.8 V digital power supply voltage for the Wafer (1.8 V). 131, 136, 179–181, 189, 190, 245

VDD12
VDD12: floating-gate programming voltage. 138, 181

VDD25
VDD25: floating-gate programming supply. 138

VDD5
VDD5: floating-gate readout supply. 138, 181

VDDA
1.8 V analog power supply voltage for the Wafer (1.8 V). 131, 136, 179, 180, 189, 190

VDDBUS
VDDBUS: synapse line driver supply. 138, 180, 181, 189

VerCL
Virtual Environment for Closed-Loop Experiments. 240

Wafer
silicon wafer used as the basis of micro-chip production. 120, 123–126, 128–131, 137, 139, 140, 143, 145, 160, 169, 179–182, 184, 194, 489

Wafer Module
Assembly of an HICANN wafer, a Main PCB, 48 FPGA communication PCBs and power supply PCBs. 53–57, 65, 123–125, 136, 137, 139, 143, 145, 146, 169, 175, 176, 179, 182, 183, 187, 190, 196, 240, 247, 489, 491

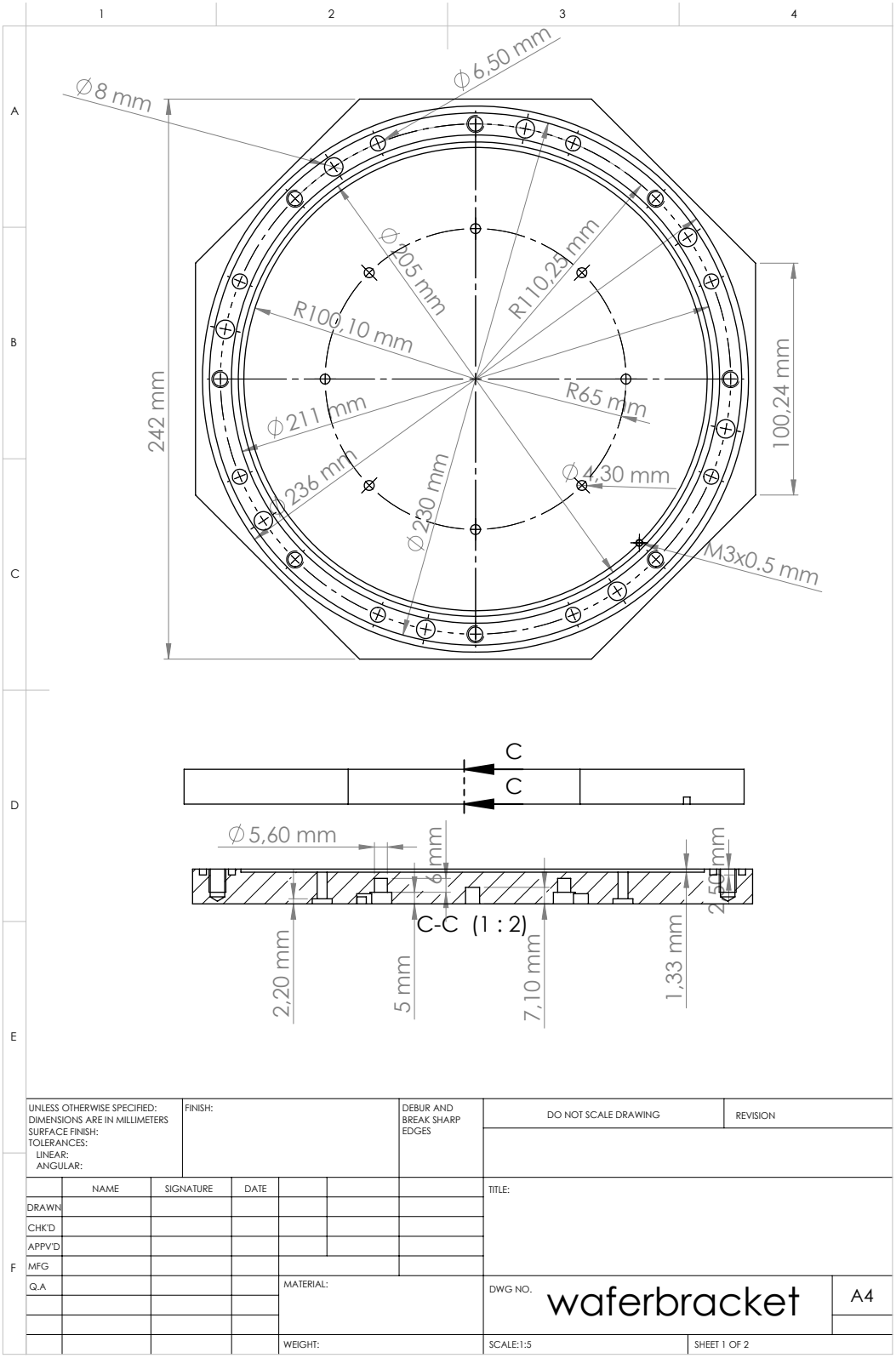
WBr
Wafer Bracket. 123, 125, 126, 143, 182, 184, 185, 195

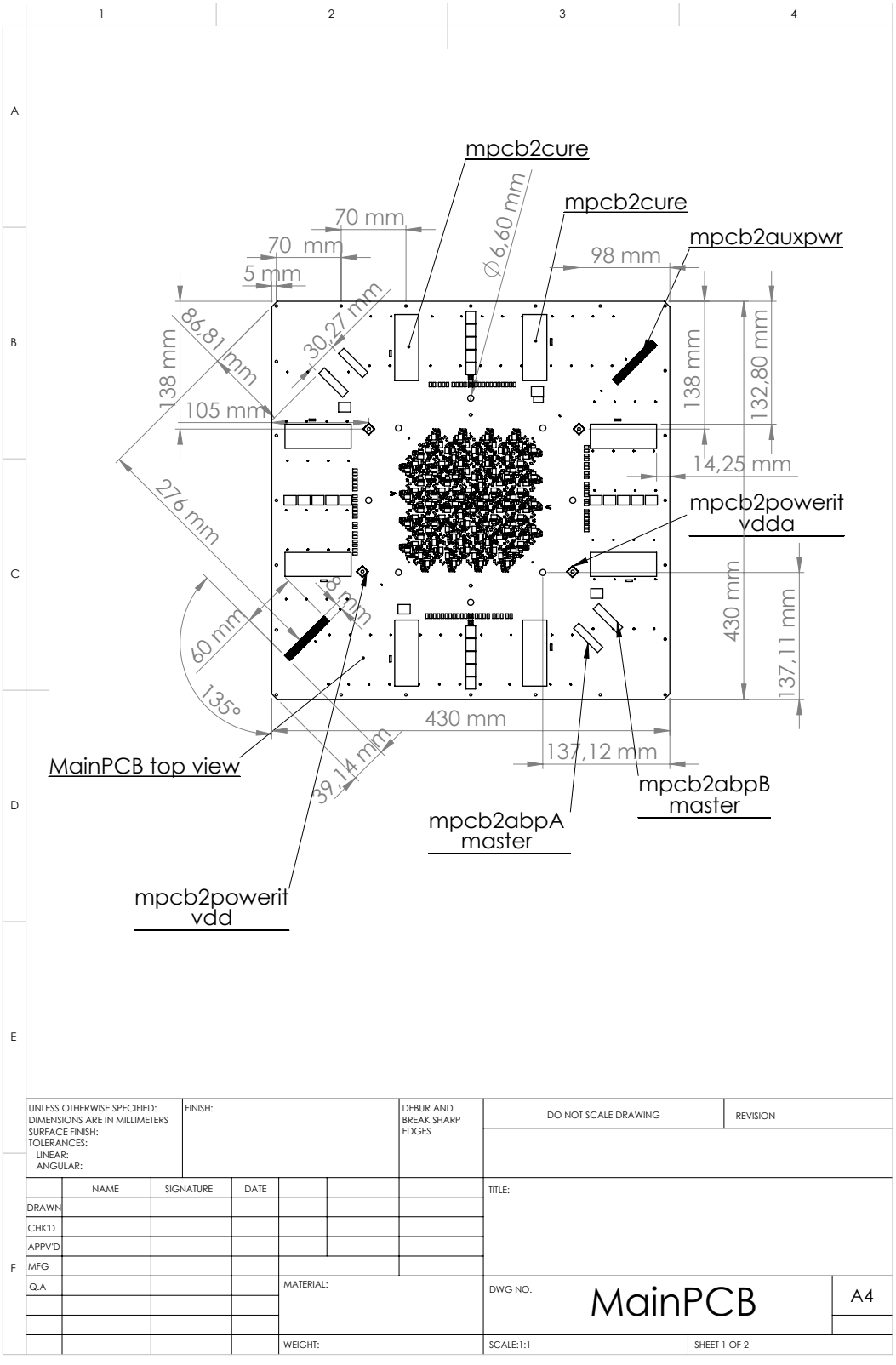
WIO
Wafer I/O PCB. 56, 137, 145, 146

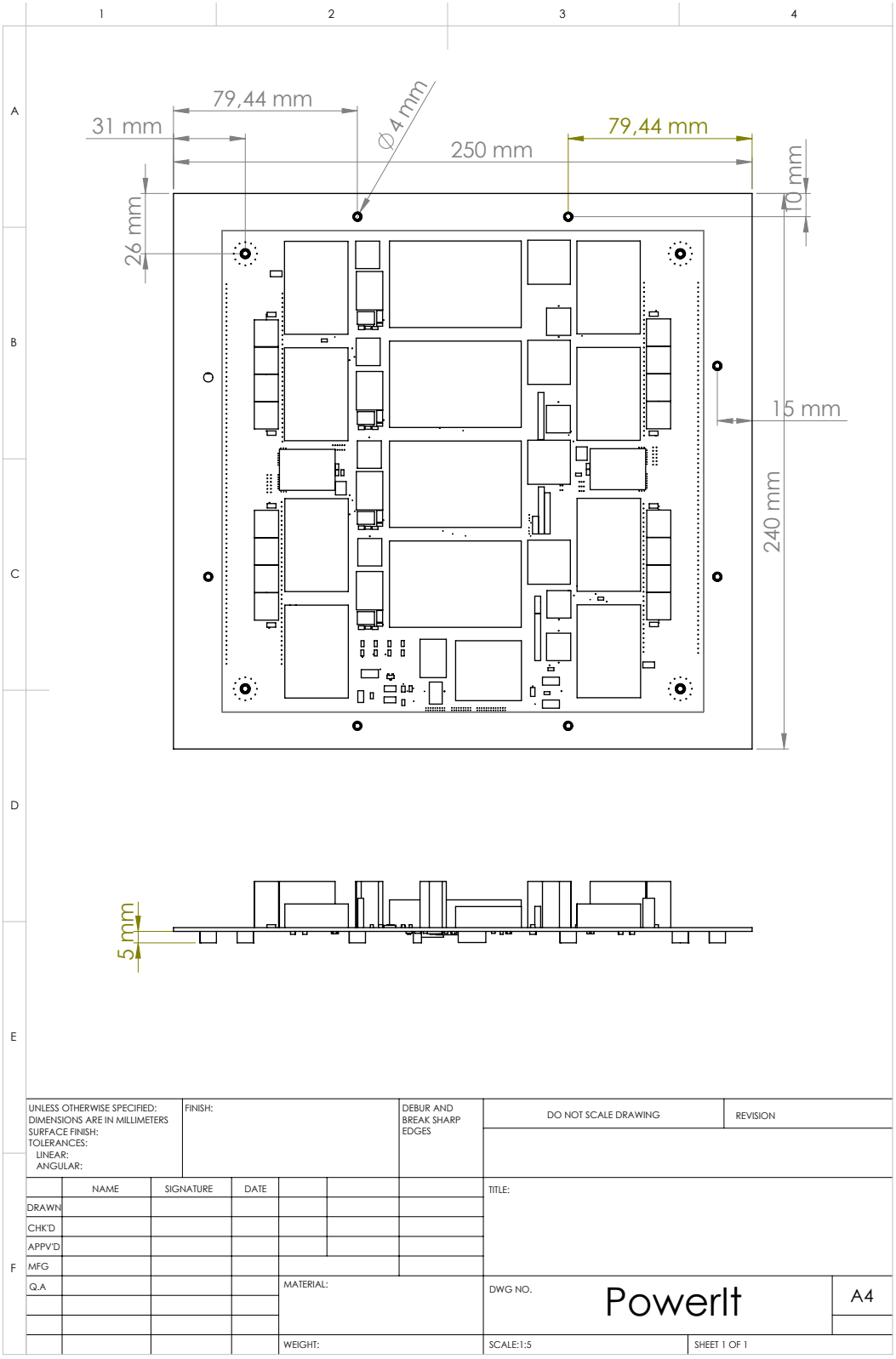
WIOH
Horizontal Wafer I/O PCB. 56, 124, 148

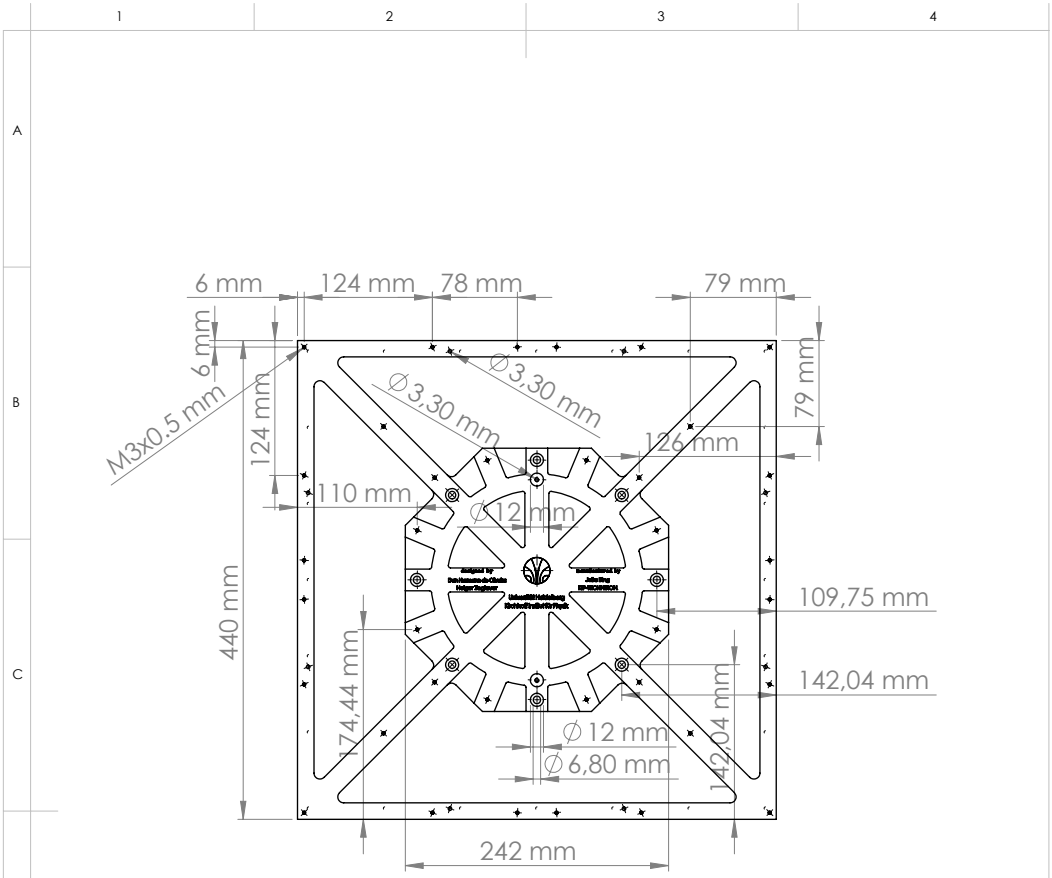
WIOV
Vertical Wafer I/O PCB. 56, 124, 148, 149

A Technical drawings of Wafer Module components

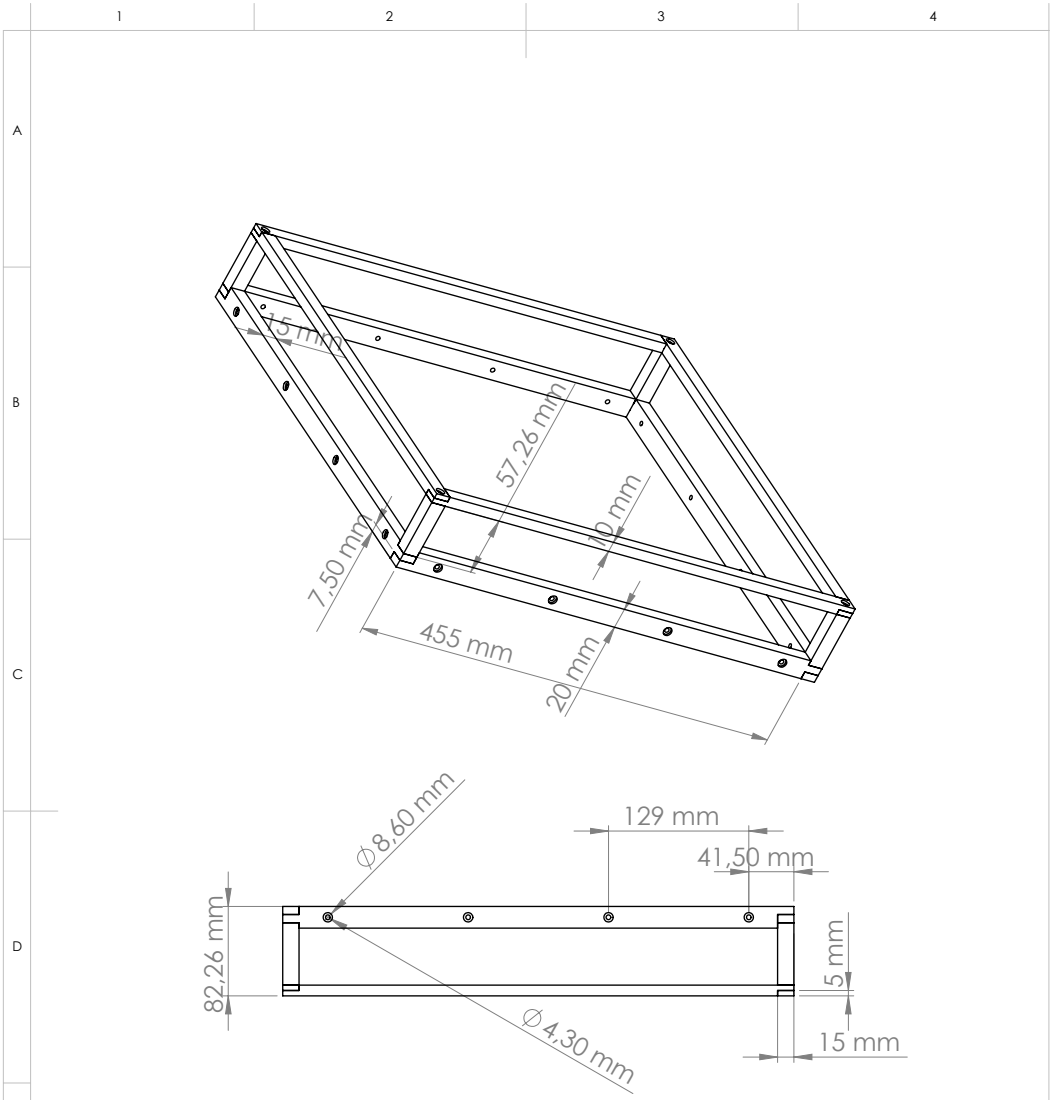

1
2
3
4
A
B
C
D
E
F
∅8 mm
∅6,50 mm
∅205 mm
R110,25 mm
R100,10 mm
242 mm
100,24 mm
R65 mm
∅211 mm
∅236 mm
∅230 mm
∅4,30 mm
M3x0.5 mm
C
C
∅5,60 mm
6 mm
2,50 mm
C-C (1 : 2)
2,20 mm
5 mm
7,10 mm
1,33 mm
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
TITLE:
MATERIAL:
DWG NO.
waferbracket
A4
WEIGHT:
SCALE:1:5
SHEET 1 OF 2


mpcb2cure
mpcb2cure
mpcb2auxpwr
mpcb2powerit vdda
mpcb2abpB master
mpcb2abpA master
mpcb2powerit vdd
MainPCB top view
70 mm
70 mm
5 mm
98 mm
Ø 6,60 mm
30,27 mm
86,81 mm
138 mm
105 mm
276 mm
60 mm
135°
8,33 mm
39,14 mm
430 mm
137,12 mm
138 mm
132,80 mm
14,25 mm
430 mm
137,11 mm
1
2
3
4
A
B
C
D
E
F
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
TITLE:
MATERIAL:
DWG NO.
MainPCB
A4
WEIGHT:
SCALE:1:1
SHEET 1 OF 2

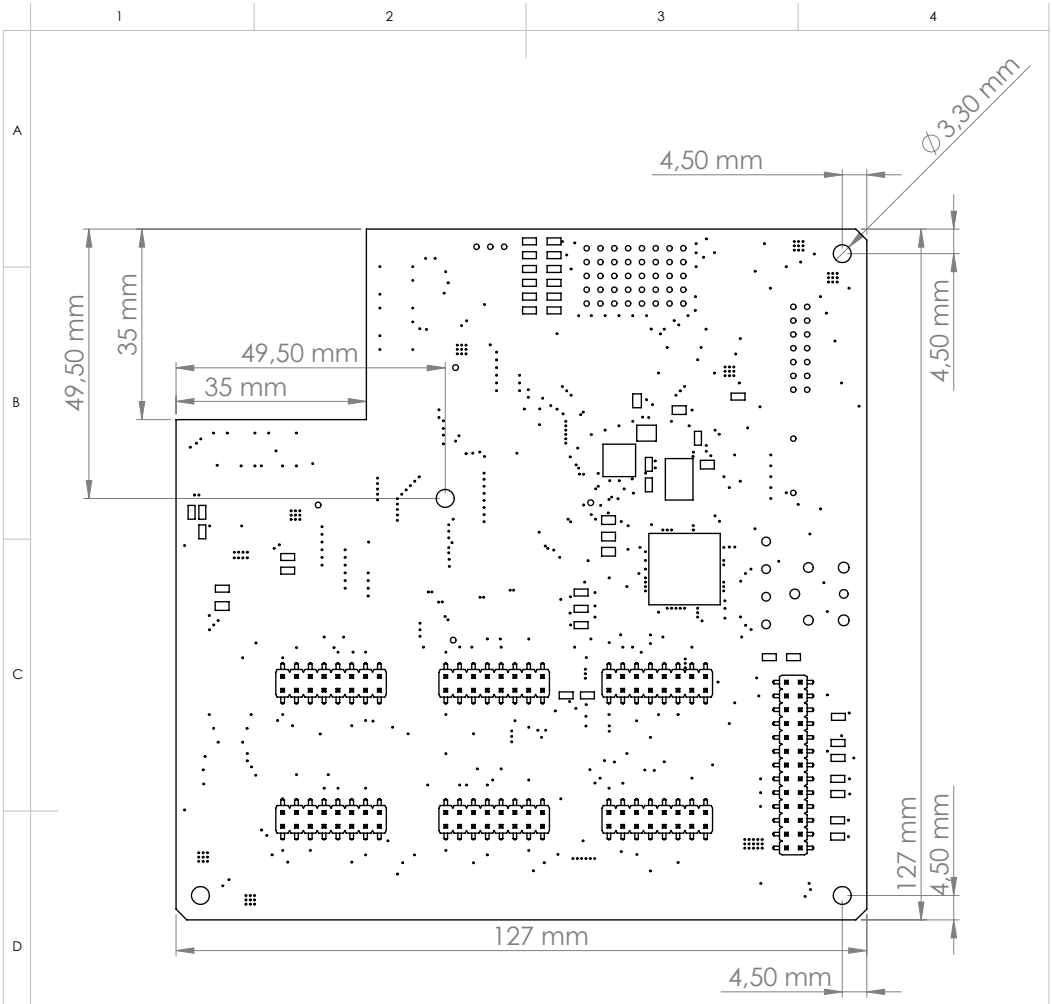

1
2
3
4
A
B
C
D
E
F
79,44 mm
31 mm
Ø4 mm
250 mm
79,44 mm
10 mm
26 mm
15 mm
240 mm
5 mm
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
TITLE:
MATERIAL:
DWG NO.
PowerIt
A4
WEIGHT:
SCALE:1:5
SHEET 1 OF 1



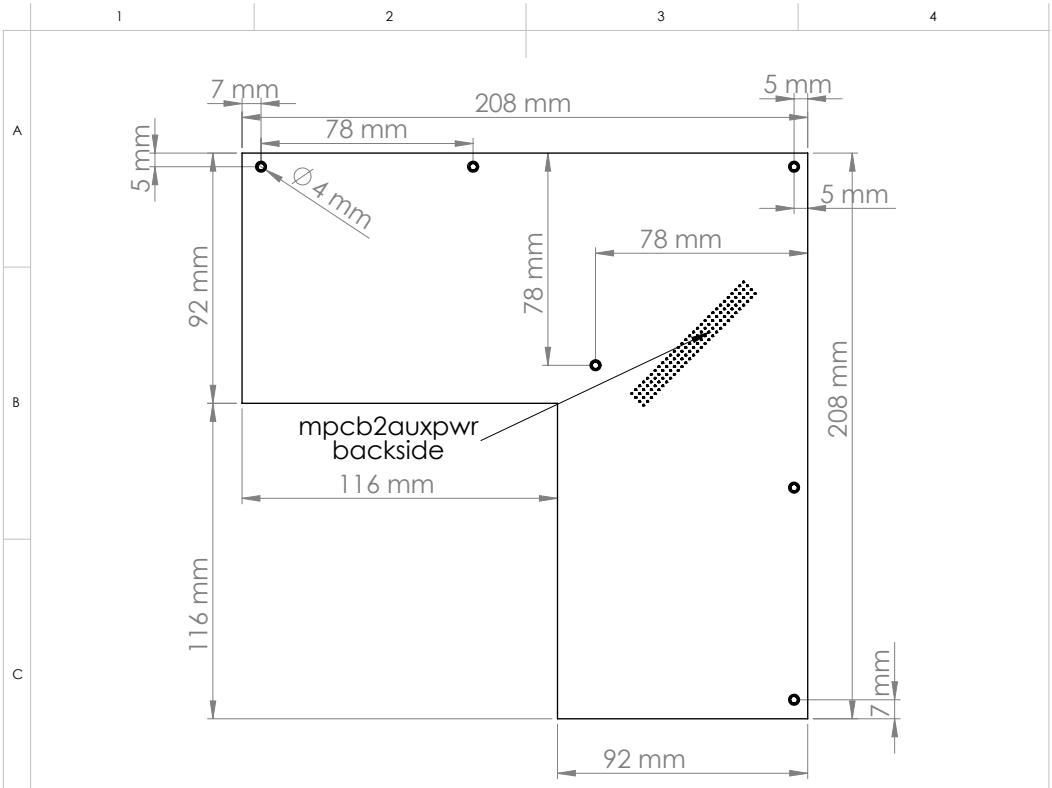
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBUR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN							
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO. TopCover	A4
				WEIGHT:		SCALE:1:10	SHEET 1 OF 3



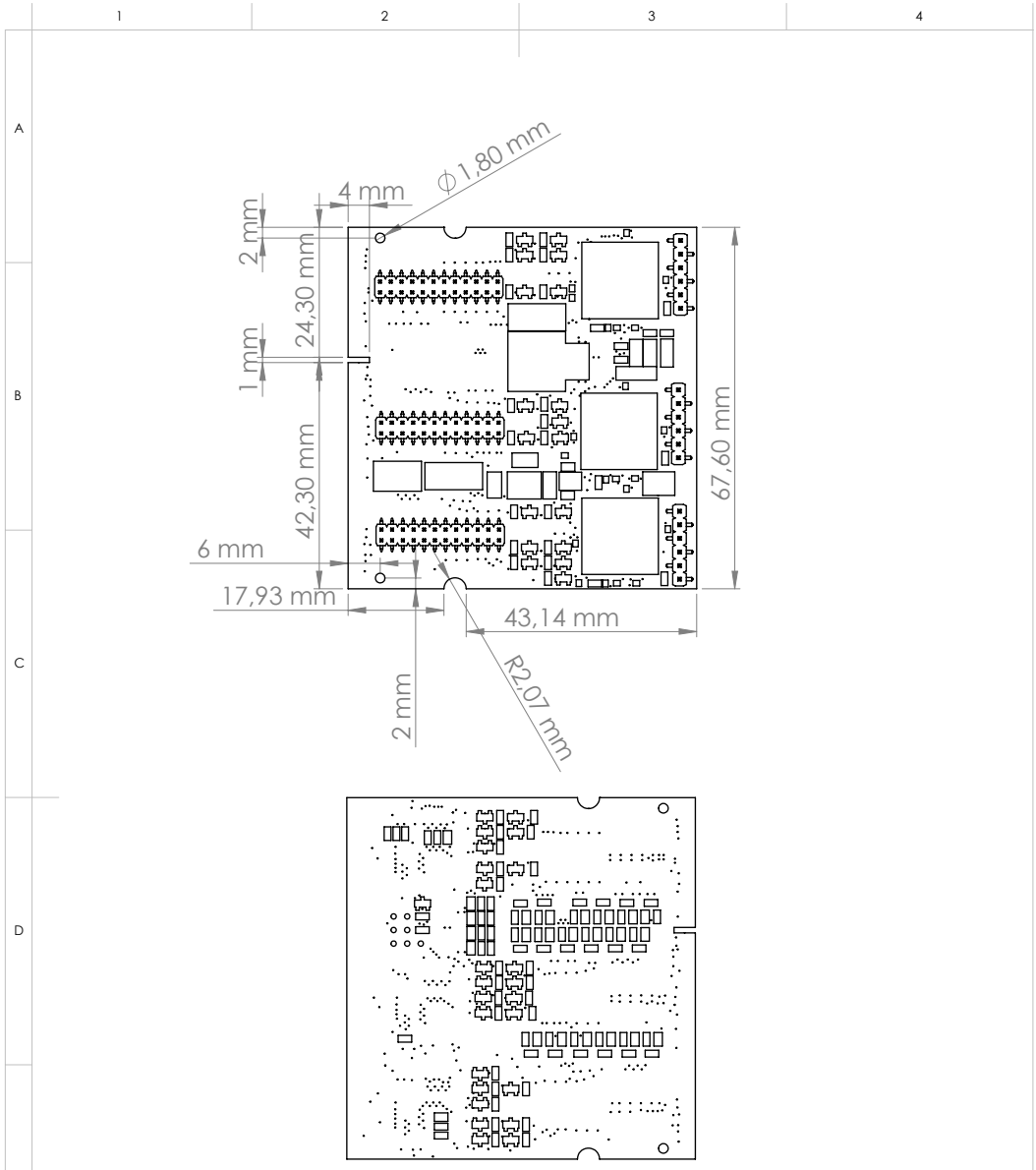
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE			
DRAWN				TITLE:		
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. infra	A4
				WEIGHT:	SCALE:1:10	SHEET 1 OF 1

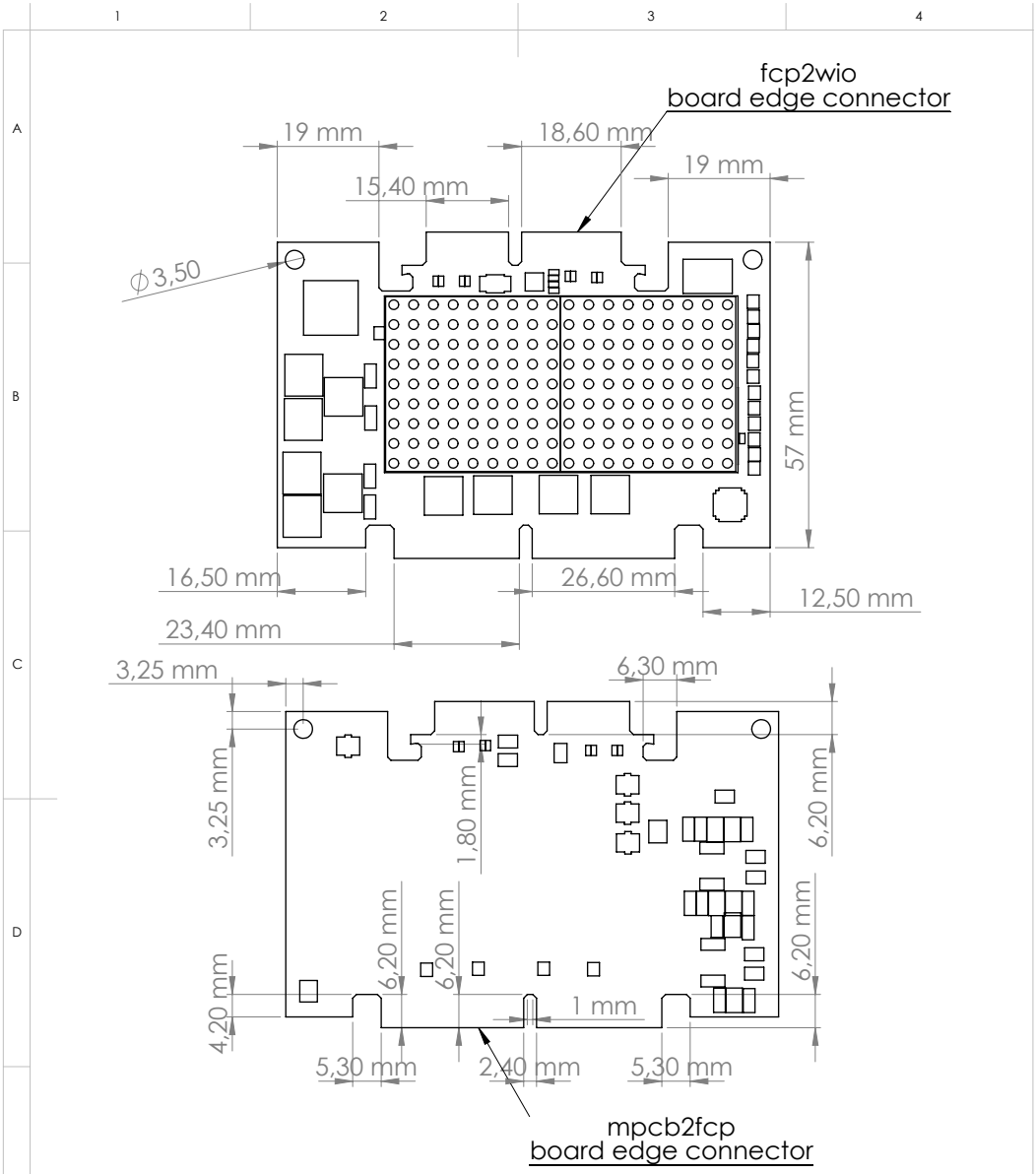


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE			
DRAWN				TITLE:		
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. AnaB	A4
				WEIGHT:	SCALE:1:1	SHEET 1 OF 2

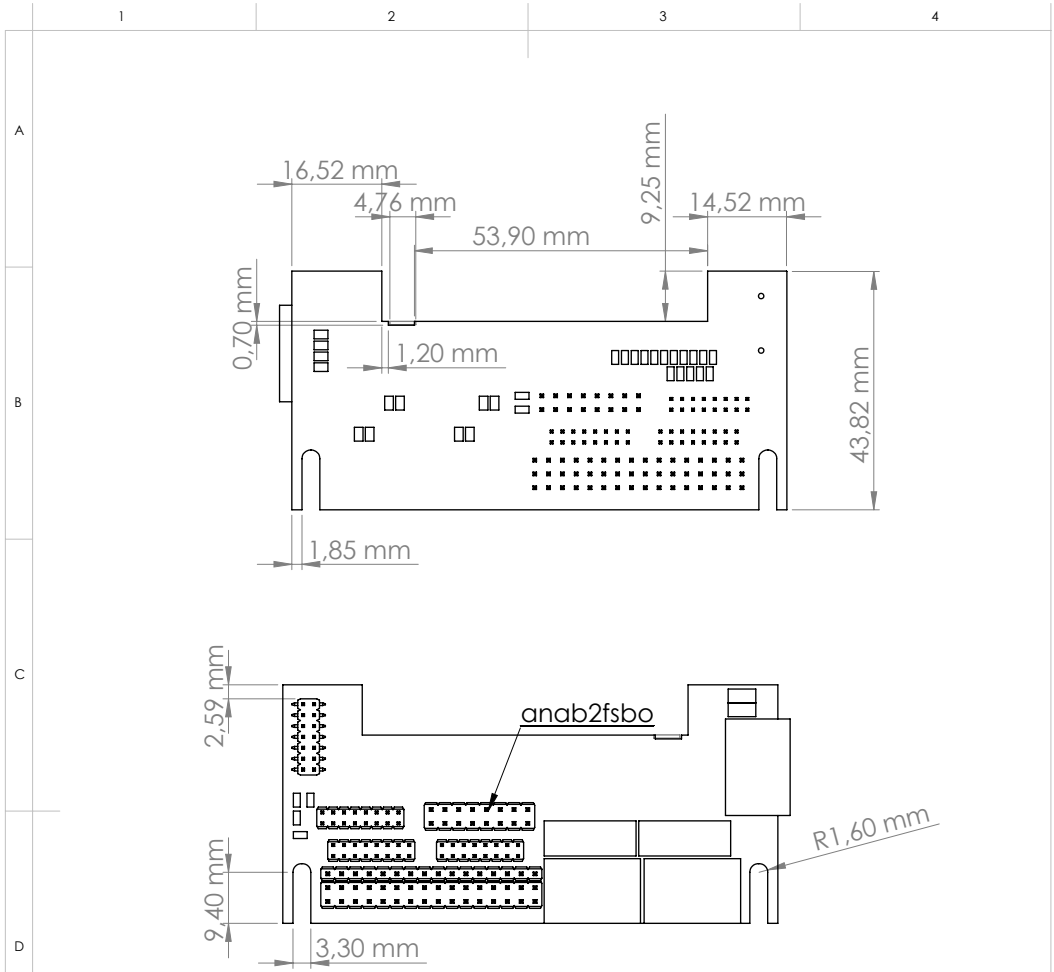


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			
DRAWN					TITLE:	
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. AuxPwr	A4
				WEIGHT:	SCALE:1:2	SHEET 1 OF 1

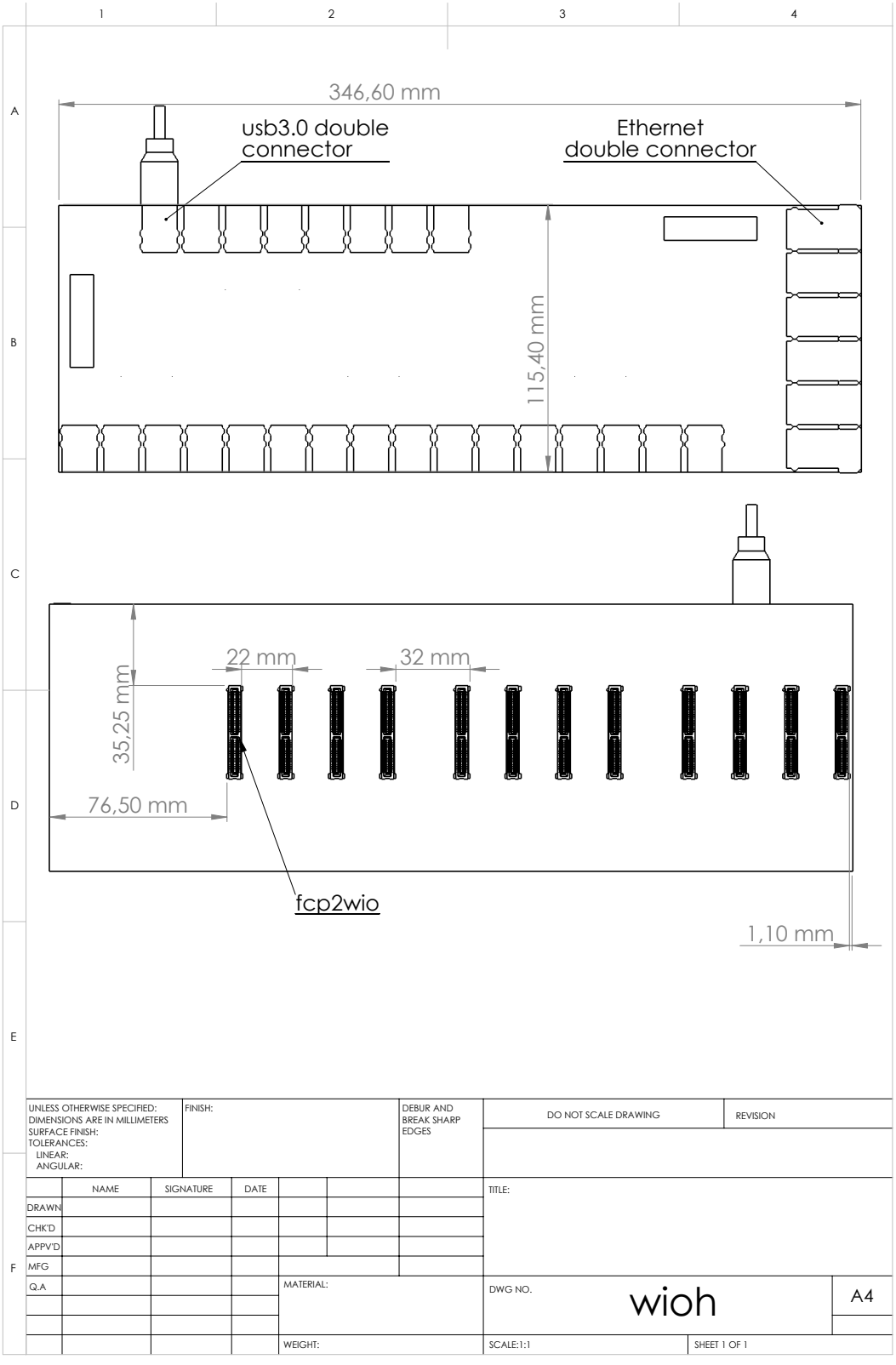

1
2
3
4
A
B
C
D
E
F
4 mm
ϕ1,80 mm
2 mm
24,30 mm
1 mm
42,30 mm
67,60 mm
6 mm
17,93 mm
43,14 mm
2 mm
R2,07 mm
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
TITLE:
MATERIAL:
DWG NO.
cure
A4
WEIGHT:
SCALE:1:1
SHEET 1 OF 1

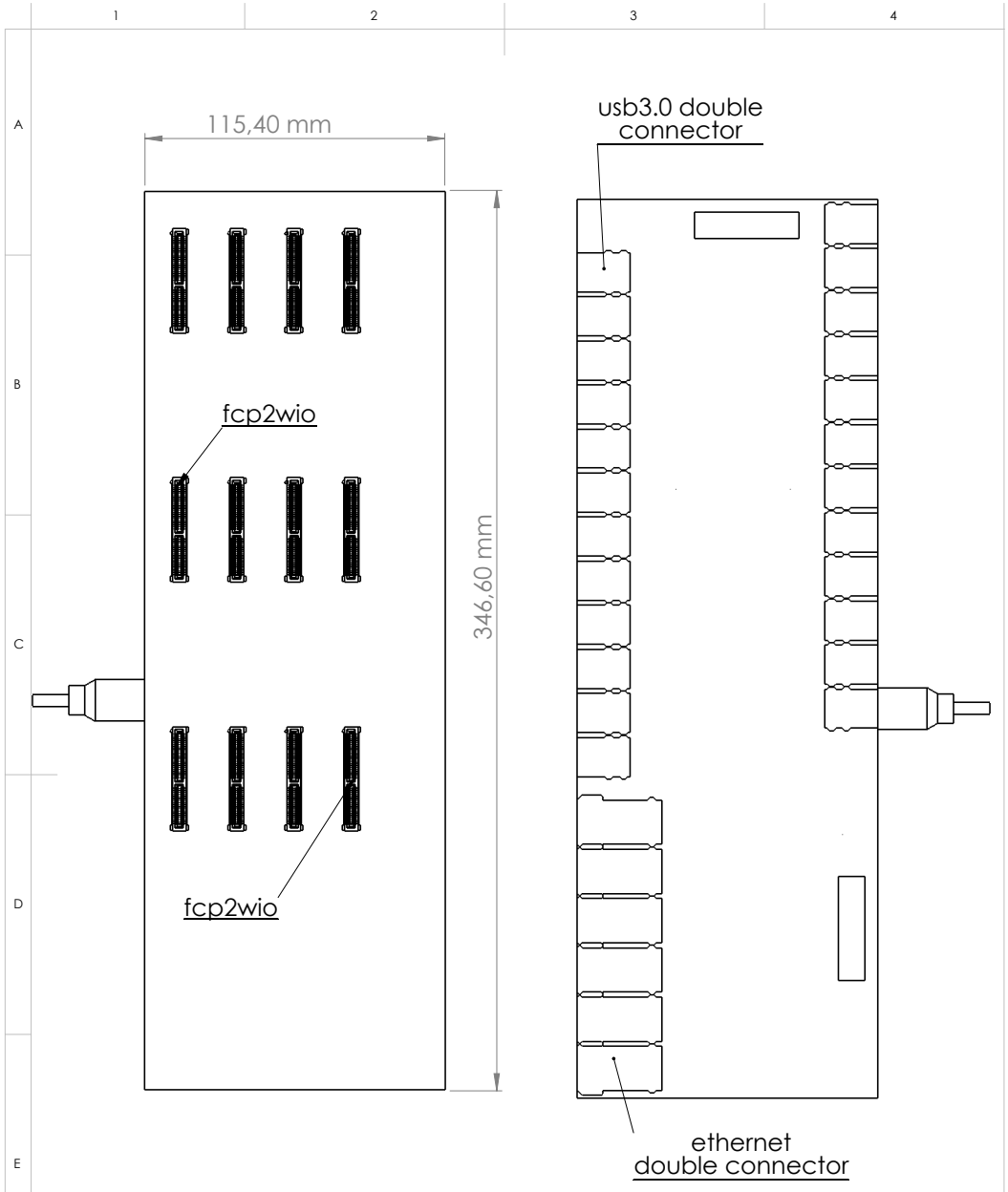


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:	DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION		
	NAME	SIGNATURE	DATE			TITLE:
DRAWN						
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. fcp	A4
				WEIGHT:	SCALE:1:1	SHEET 1 OF 1



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE		TITLE:	
DRAWN						
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. FsBo	A4
				WEIGHT:	SCALE:1:1	SHEET 1 OF 1


1
2
3
4
A
B
C
D
E
F
346,60 mm
usb3.0 double connector
Ethernet double connector
115,40 mm
22 mm
32 mm
35,25 mm
76,50 mm
fcp2wio
1,10 mm
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
TITLE:
MATERIAL:
DWG NO.
wioh
A4
WEIGHT:
SCALE:1:1
SHEET 1 OF 1


1
2
3
4
A
B
C
D
E
F
115,40 mm
346,60 mm
fcp2wio
fcp2wio
usb3.0 double connector
ethernet double connector
UNLESS OTHERWISE SPECIFIED:
DIMENSIONS ARE IN MILLIMETERS
SURFACE FINISH:
TOLERANCES:
LINEAR:
ANGULAR:
FINISH:
DEBUR AND BREAK SHARP EDGES
DO NOT SCALE DRAWING
REVISION
NAME
SIGNATURE
DATE
DRAWN
CHK'D
APPV'D
MFG
Q.A
MATERIAL:
WEIGHT:
TITLE:
DWG NO.
wiov
A4
SCALE:1:1
SHEET 1 OF 1