Human Brain Project

Co-funded by the European Commission

Project Number:	604102	Project Title:	Human Brain Project

Document Title:	Neuromorphic Platform Specification — public version
Document Filename:	HBP_SP9_D9.7.1_NeuromorphicPlatformSpec_ fbf0f70 from 05 October 2022
Deliverable Number:	D 9.7.1
Deliverable Type:	Platform Specification
Sub Project:	SP9
Planned Delivery Date:	M6 - 31 March 2014
Actual Delivery Date:	M7
Dissemination level:	public
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

Human Brain Project

Co-funded by the European Commission

Neuromorphic Platform Specification — public version

05 October 2022 (git fbf0f70 — public)

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.

Please note: this document is the technical specification / documentation. For user-access to the platform consult the HBP Neuromorphic Computing Platform Guidebook (also available as downloadable .pdf version.

Contents

The Neuromorphic Computing Platform 11
What is Neuromorphic Computing? . . . 11
What are the key features of the HBP Neuromorphic Computing Platform? . 12
How will the NM Platform be used? . . . 16
Integration of the NM Platform into the HBP Platform Ecosystem . . . 16
The purpose of this document . . . 17

1 User interface to the Neuromorphic Computing Platform 19

1.1 Overall goals 21

1.2 Use cases 23
1.2.1 A single run of a simple network model . . . 23
1.2.2 A scripted run of a complex network model with input data and parameter files 24
1.2.3 Using the Neuromorphic Computing Platform through the Unified Portal and Brain Simulation Platform . . . 26
1.2.4 Parameter sweeps . . . 27
1.2.5 Closed-loop experiment involving a virtual environment . . . 27

1.3 Functional requirements 29
1.3.1 Model and experiment descriptions . . . 29
1.3.2 Job control interface . . . 29
1.3.2.1 Batch mode . . . 30
1.3.3 Data handling . . . 30
1.3.4 Closed-loop experiments . . . 31

1.4 Non-functional requirements 33
1.4.1 Sharing . . . 33
1.4.2 Authentication and Authorization . . . 33
1.4.3 Security . . . 33
1.4.4 Accounting . . . 34
1.4.5 Efficiency and user volumes . . . 34
1.4.6 Reliability . . . 34

1.5 Architectural overview 35

1.5.1 Job submission API . . . 35
1.5.1.1 Overview . . . 35
1.5.1.2 Endpoints . . . 36
1.5.1.3 Resource descriptions . . . 36
1.5.1.4 Serializations and allowed document types . . . 37
1.5.1.5 Physical architecture . . . 37
1.5.2 Python client for REST API . . . 37
1.5.3 Model/experiment verification . . . 37
1.5.4 Resource management software in Heidelberg and Manchester . . . 38
1.5.5 Tools for exporting Brain Builder model descriptions as PyNN descriptions . . . 38

1.6 Interfaces to other platforms **39**
1.6.1 Services required from other Platforms . . . 39
1.6.2 Services provided to other Platforms . . . 39

1.7 Key performance indicators and Function blocks **41**

2 Neuromorphic Computing with Physical Emulation of Brain Models **47**

2.1 Physical Model Platform: NM-PM (BrainScaleS-1) **49**
2.1.1 Neuromorphic Physical Model . . . 49
2.1.2 Constitutent Parts of the Neuromorphic Physical Model version 1 (NM-PM1) 50

2.2 Users view of the NM-PM system **55**
2.2.1 Usage of the NM-PM as a modeling back-end . . . 55
2.2.2 Low-level user access . . . 56
2.2.3 Real-time interaction with the NM-PM . . . 57
2.2.4 Evaluation Workflow . . . 58

2.3 Physical Model Platform: BrainScaleS-2 **61**
2.3.1 Omnibus . . . 61
2.3.1.1 Bus_if_split . . . 61
2.3.1.2 Bus_if_arb . . . 61
2.3.1.3 Bus_delay . . . 63
2.3.1.4 Bus_reg_target . . . 63
2.3.1.5 m4 macro . . . 63
2.3.2 Routing . . . 64
2.3.2.1 Crossbar (Layer-1) . . . 64
2.3.2.2 External events (Layer-2) . . . 64
2.3.2.3 PADI-Bus . . . 65
2.3.2.4 Synapse driver . . . 65
2.3.2.5 Synapse . . . 66
2.3.2.6 Neuron . . . 66

3 Neuromorphic Computing with Many-core Emulation of Brain Models 69

3.1 Multi-core Platform: NM-MC 71
3.1.1 Physical Architecture . . . 72
3.1.2 Software . . . 75

3.2 SpiNNaker Chip Datasheet 79
3.2.1 Chip Organization . . . 82
3.2.1.1 Block Diagram . . . 82
3.2.1.2 System-on-Chip hierarchy . . . 83
3.2.1.3 Register description convention . . . 84
3.2.2 System architecture . . . 85
3.2.2.1 Routing . . . 86
3.2.2.2 Time references . . . 87
3.2.2.3 System-level address spaces . . . 87
3.2.3 ARM968 processing subsystem . . . 88
3.2.3.1 Features . . . 88
3.2.3.2 ARM968 subsystem organisation . . . 89
3.2.3.3 Memory Map . . . 89
3.2.4 ARM 968 . . . 92
3.2.4.1 Features . . . 92
3.2.4.2 Organization . . . 92
3.2.4.3 Fault-tolerance . . . 92
3.2.5 Vectored interrupt controller . . . 93
3.2.5.1 Features . . . 93
3.2.5.2 Register summary . . . 93
3.2.5.3 Register details . . . 94
3.2.5.4 Interrupt sources . . . 97
3.2.5.5 Fault-tolerance . . . 98
3.2.6 Counter/timer . . . 100
3.2.6.1 Features . . . 100
3.2.6.2 Register summary . . . 100
3.2.6.3 Register details . . . 101
3.2.6.4 Fault-tolerance . . . 103
3.2.7 DMA controller . . . 104
3.2.7.1 Features . . . 104
3.2.7.2 Using the DMA controller . . . 104
3.2.7.3 Register summary . . . 105
3.2.7.4 Register details . . . 106
3.2.7.5 Fault-tolerance . . . 111
3.2.8 Communications controller . . . 113
3.2.8.1 Features . . . 113
3.2.8.2 Packet formats . . . 113
3.2.8.3 Control byte summary . . . 115
3.2.8.4 Debug access to neighbouring devices . . . 116

3.2.8.5 Register summary . . . 117
3.2.8.6 Register details . . . 117
3.2.8.7 Fault-tolerance . . . 120
3.2.9 Communications NoC . . . 121
3.2.9.1 Features . . . 121
3.2.9.2 Input structure . . . 121
3.2.9.3 Output structure . . . 121
3.2.10 Router . . . 122
3.2.10.1 Features . . . 122
3.2.10.2 Description . . . 122
3.2.10.3 Internal organization . . . 124
3.2.10.4 Multicast (MC) router . . . 125
3.2.10.5 The point-to-point (P2P) router . . . 126
3.2.10.6 The nearest-neighbour (NN) router . . . 127
3.2.10.7 Time phase handling . . . 127
3.2.10.8 Packet error handler . . . 128
3.2.10.9 Emergency routing . . . 128
3.2.10.10 Register summary . . . 128
3.2.10.11 Register details . . . 129
3.2.10.12 Fault-tolerance . . . 137
3.2.10.13 Test . . . 138
3.2.11 Inter-chip transmit and receive interfaces . . . 139
3.2.11.1 Features . . . 139
3.2.11.2 Programmer view . . . 139
3.2.11.3 Fault-tolerance . . . 139
3.2.12 System NoC . . . 141
3.2.12.1 Features . . . 141
3.2.12.2 Organisation . . . 142
3.2.13 SDRAM interface . . . 143
3.2.13.1 Features . . . 143
3.2.13.2 Register summary . . . 143
3.2.13.3 Register details . . . 145
3.2.13.4 The delay-locked loop (DLL) . . . 151
3.2.13.5 Fault-tolerance . . . 153
3.2.14 System Controller . . . 154
3.2.14.1 Features . . . 154
3.2.14.2 Register summary . . . 154
3.2.14.3 Register details . . . 155
3.2.15 Ethernet MII interface . . . 168
3.2.15.1 Features . . . 168
3.2.15.2 Using the Ethernet MII interface . . . 168
3.2.15.3 Register summary . . . 168
3.2.15.4 Register details . . . 169
3.2.15.5 Fault-tolerance . . . 173

3.2.16 Watchdog timer . . . 174
3.2.16.1 Features . . . 174
3.2.16.2 Register summary . . . 174
3.2.16.3 Register details . . . 175
3.2.17 System RAM . . . 177
3.2.17.1 Features . . . 177
3.2.17.2 Address location . . . 177
3.2.17.3 Fault-tolerance . . . 177
3.2.17.4 Test . . . 178
3.2.18 Boot ROM . . . 179
3.2.18.1 Features . . . 179
3.2.18.2 Address location . . . 179
3.2.18.3 Fault-tolerance . . . 179
3.2.19 JTAG . . . 180
3.2.19.1 Features . . . 180
3.2.19.2 Organisation . . . 180
3.2.19.3 Operation . . . 180
3.2.20 Input and Output signals . . . 181
3.2.20.1 Key . . . 181
3.2.20.2 SDRAM interface . . . 181
3.2.20.3 JTAG . . . 181
3.2.20.4 Ethernet MII . . . 182
3.2.20.5 Communication links . . . 182
3.2.20.6 Miscellaneous . . . 183
3.2.20.7 Internal SDRAM interface . . . 184
3.2.20.8 Internal SDRAM power & ground . . . 184
3.2.21 Packaging . . . 185
3.2.22 Application notes . . . 186
3.2.22.1 Firefly synchronization . . . 186
3.2.22.2 Neuron address space . . . 186

3.3 SpiNNaker Software Datasheet 187
3.3.1 Run-time software . . . 188
3.3.1.1 Run-time software stack . . . 189
3.3.1.2 Inter-processor communication . . . 189
3.3.1.3 Runtime memory map . . . 192
3.3.2 Application programming interface (API) . . . 193
3.3.2.1 Event-driven programming model . . . 193
3.3.2.2 Programming interface . . . 194
3.3.3 Neural net simulation frameworks . . . 215
3.3.3.1 Spiking Neural net simulation framework . . . 215
3.3.3.2 MLP simulation framework . . . 218
3.3.4 Neural net simulation development route . . . 220
3.3.4.1 pyNN.spiNNaker . . . 221
3.3.4.2 PyNN API functions list . . . 224

3.3.4.3 Simulation setup and control . . . 225
3.3.4.4 Object-oriented interface for creating and recording networks . . . 225
3.3.4.5 PopulationView . . . 225
3.3.4.6 Assembly . . . 225
3.3.4.7 Object-oriented interface for connecting populations of neurons . . . 226
3.3.4.8 Procedural interface for creating, connecting and recording networks 227
3.3.4.9 Neural Models . . . 227
3.3.4.10 Specification of synaptic plasticity . . . 227
3.3.4.11 Current Injection . . . 228
3.3.5 Damson development route . . . 232
3.3.5.1 Damson program compilation . . . 232
3.3.5.2 Damson code components . . . 232
3.3.5.3 Mapping code to SpiNNaker processors . . . 233
3.3.5.4 Runtime system . . . 233
3.3.5.5 Damson development flow . . . 233
3.3.6 PACMAN: partition and configuration manager . . . 233
3.3.6.1 Introduction . . . 233
3.3.6.2 Splitting . . . 236
3.3.6.3 Grouping . . . 240
3.3.6.4 Mapper . . . 241
3.3.6.5 Object File Generator . . . 243
3.3.6.6 Neural Data Structure generation . . . 245
3.3.6.7 Automatic Run Script generation . . . 246
3.3.6.8 MLP PACMAN . . . 247
3.3.7 Coding guidelines . . . 257
3.3.7.1 All languages . . . 257
3.3.7.2 C . . . 257
3.3.7.3 ARM assembly . . . 258
3.3.7.4 Python . . . 259
3.3.8 Documentation guidelines . . . 259
3.3.8.1 C / C++ . . . 259
3.3.8.2 Assembly language . . . 260
3.3.8.3 Robodoc configuration file . . . 262

4 Benchmarks 265

4.1 Overall goals 267

4.2 Quality criteria for neuromorphic benchmark tests 269
4.2.1 What units should be benchmarked? . . . 269

4.3 Use cases 271
4.3.1 Tracking the performance of a neuromorphic computing system over time . . . 271

4.3.2 Determining whether the Neuromorphic Computing Platform is suitable for a specific task . . . 271

4.4 Functional requirements 273

4.5 Architectural overview 275

4.6 Implementation 277
4.6.1 Defining models and tasks . . . 277
4.6.2 Returning numerical measures . . . 278
4.6.3 Registering benchmarks . . . 279
4.6.4 Running benchmarks . . . 279

5 Following the platform building: Key Performance Indicators and time plans 281

5.1 KPIs and time plans 283
5.1.1 KPIs of the NMPM . . . 283
5.1.1.1 Wafer Production . . . 283
5.1.1.2 Printed Circuit Board Production . . . 283
5.1.1.3 Wafer Module Production . . . 285
5.1.1.4 Software and Hardware Usage KPIs . . . 285
5.1.2 KPIs of the NMMC . . . 286
5.1.2.1 Cabinet Assembly . . . 286
5.1.2.2 Sub-rack assembly . . . 286
5.1.2.3 Network . . . 287
5.1.2.4 Fan Tray Assembly . . . 288
5.1.2.5 Power Supply Assembly . . . 288
5.1.3 KPIs of the common software part . . . 288
5.1.4 KPIs of the benchmark part . . . 289

Bibliography 291

Glossary 293

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 [9] 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 [3] to reach a power consumption of 20-30 MW for such system.

Massive parallelism also affects the speed of brain simulations on neuromorphic systems [9]. 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 51)

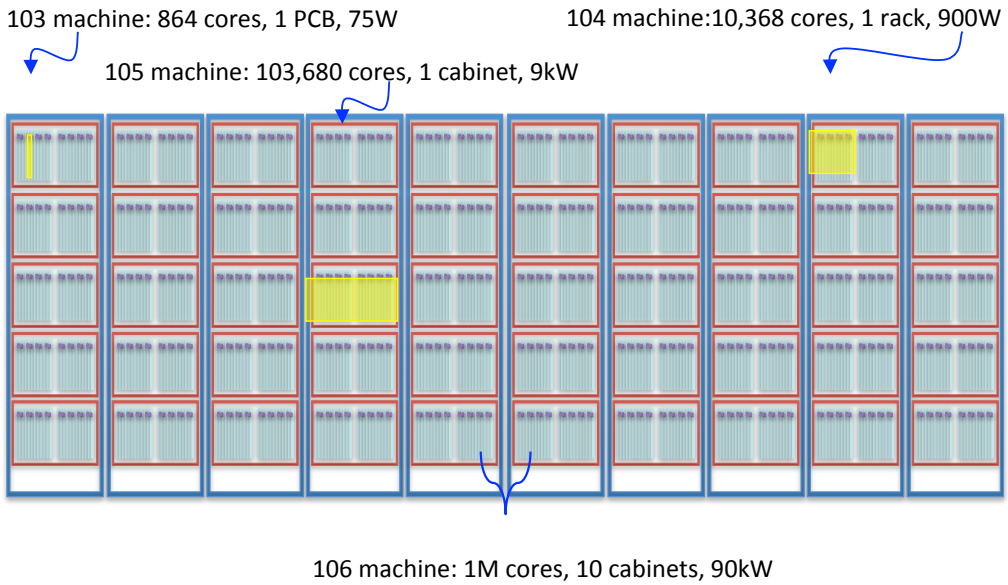


Figure .2: Concept view of the NM-MC1 system

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 enables access by non-expert users. A typical example are neuroscientists running experiments implementing reverse

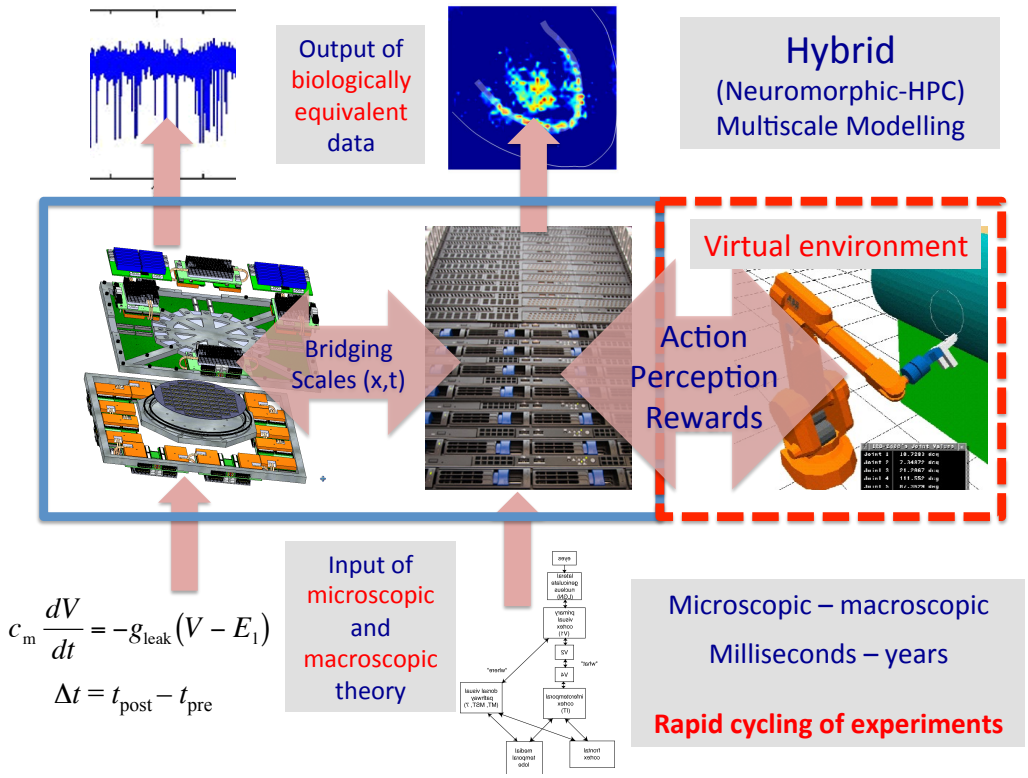


Figure .3: Hybrid operation

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.

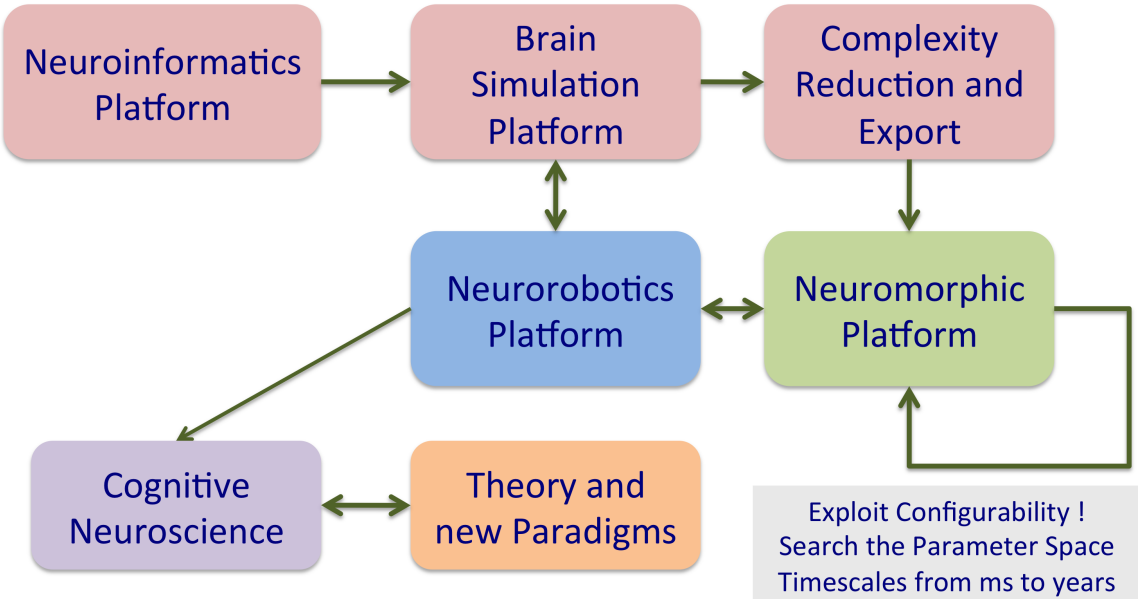


Figure .4: Integration of the Neuromorphic Computing Platform into the HBP Platform Ecosystem

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 Unified Portal to perform his simulations.

Preconditions Dennis' model is available in the Unified Portal.

Success scenario Using the Unified 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 Unified 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 Unified 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 Unified 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 – Unified Portal
3) Provenance tracking – Unified 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 Unified 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		

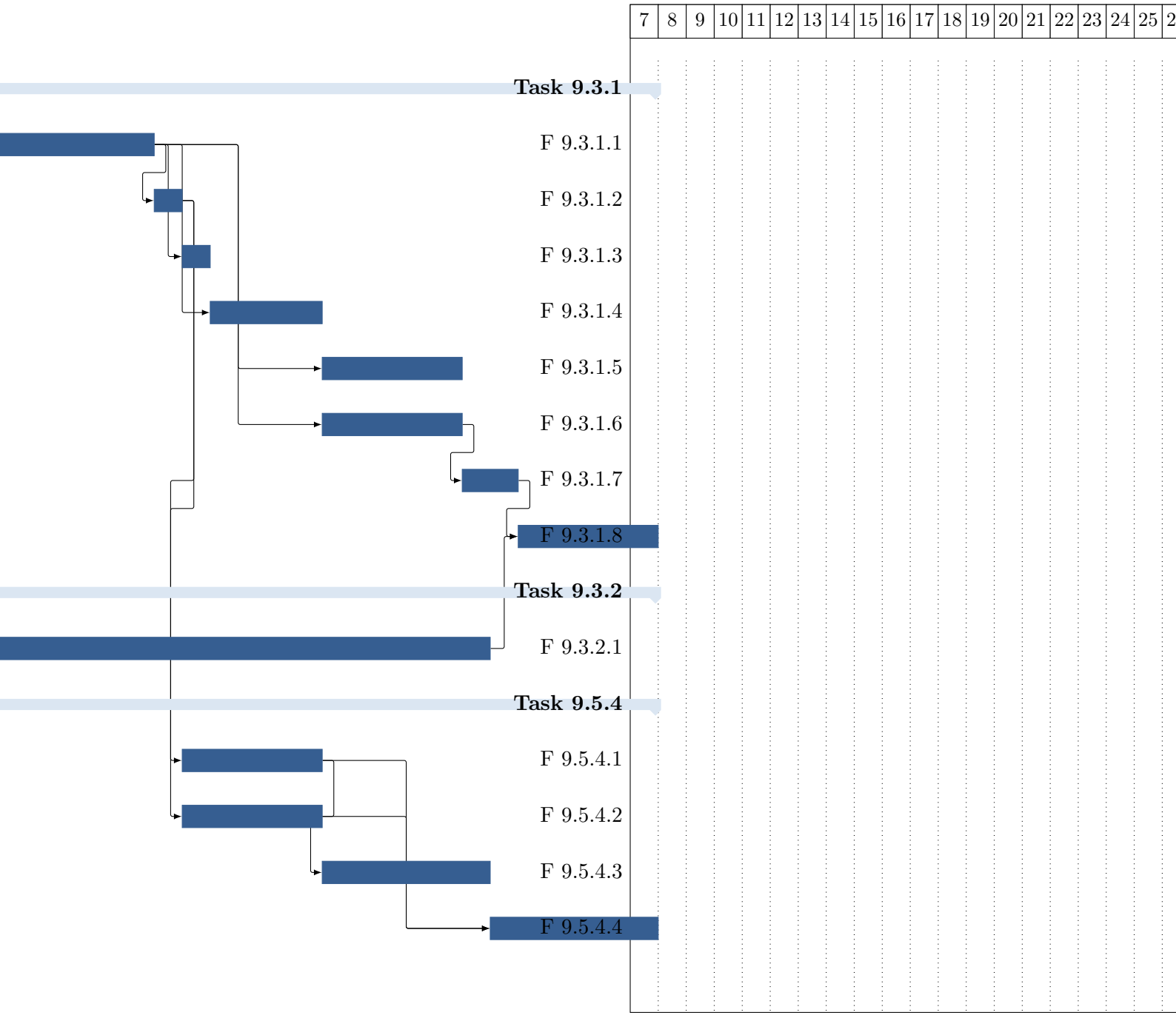


Figure 1.7.1: Scheduling of Functions to be implemented in building the user interface to the Neuromorphic Computing Platform. The numbers in the top row refer to project months. Month 7 is April 2014, Month 30 is March 2016.

Part 2

Neuromorphic Computing with Physical Emulation of Brain Models

2.1 Physical Model Platform: NM-PM (BrainScaleS-1)

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. (In this 'public' version of the document these detailed chapters are not included).

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

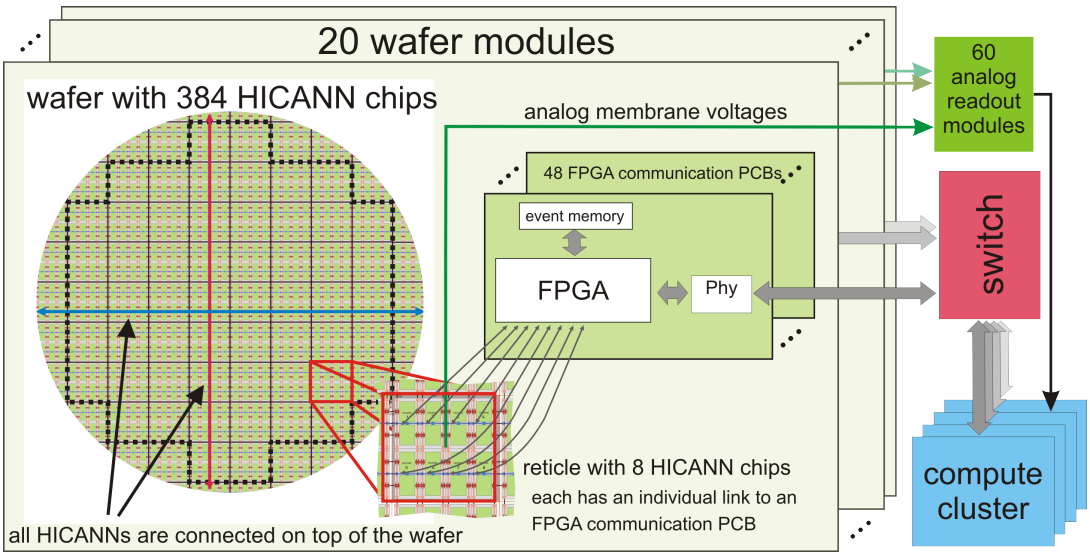


Figure 2.1.1: Simplified overview of the Neuromorphic Physical Model version 1 (NM-PM1) system.

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 Top-of-Rack (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 [2] 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)[1] 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) , ([5, 7]). 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, which computes a hardware configuration which represents the network topology given by the user, and the calibration , which translates the user-defined neuron and synapse parameters to hardware-specific settings for the analog components. The mapping uses the Hardware Abstraction Layeras the interface to the hardware system as well as to the hardware Executable System Specification (ESS).

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 ([4], `EIF_cond_exp_isfa_ista` in PyNN notation). The ranges for neuron, synapse and connectivity parameters are limited by the hardware implementation.

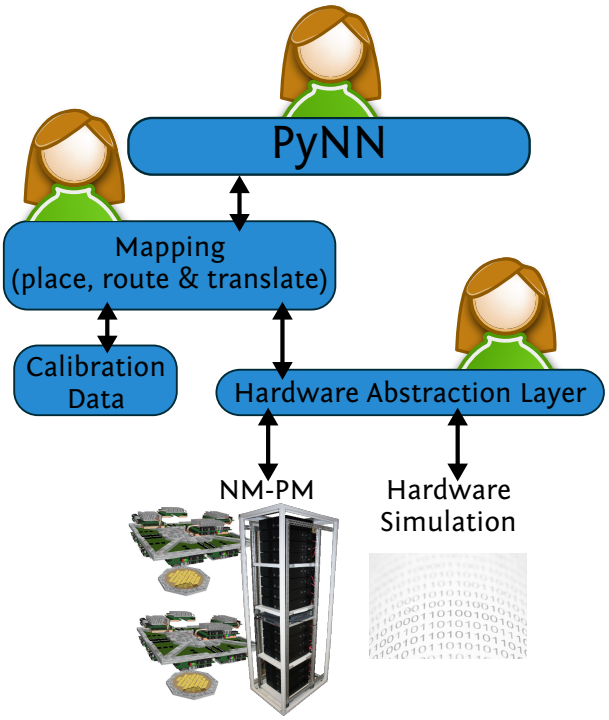


Figure 2.2.1: PyNN is the main user interface for the NM-PM. It hides the mapping and calibration steps from the end-user. Expert users can still access the hardware abstraction layers directly.

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

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, which handles the topological translation, and the Calibration 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 HALbe (Hardware Abstraction Layer Backend) and StHAL (Stateful Hardware Abstraction Layer), 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. 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 Hardware Abstraction Layer Backend (HALbe).

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 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

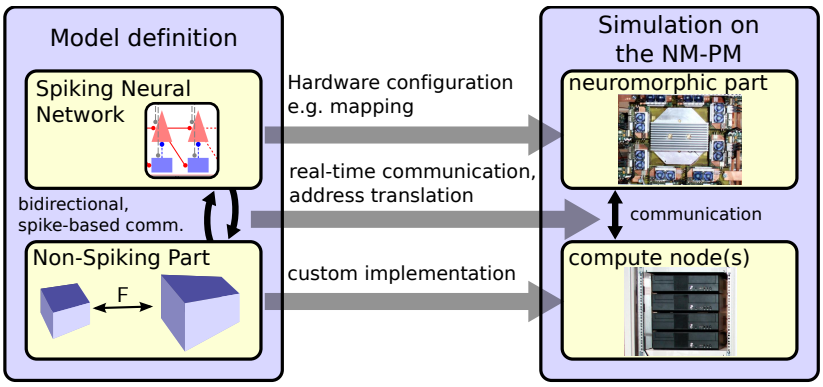


Figure 2.2.2: Hybrid simulation on the NM-PM. A system that consists of a neural network and a non-spiking part (left) is being simulated on the hybrid neuromorphic-classical device (right). The neural network is emulated on the neuromorphic part while the remaining part of the system is simulated in 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.

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 [8, 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.

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

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 [6].

...

(Info: the rest of the NM-PM1 description is 'consortium internal' and therefore not included in this 'public' version of the document.)

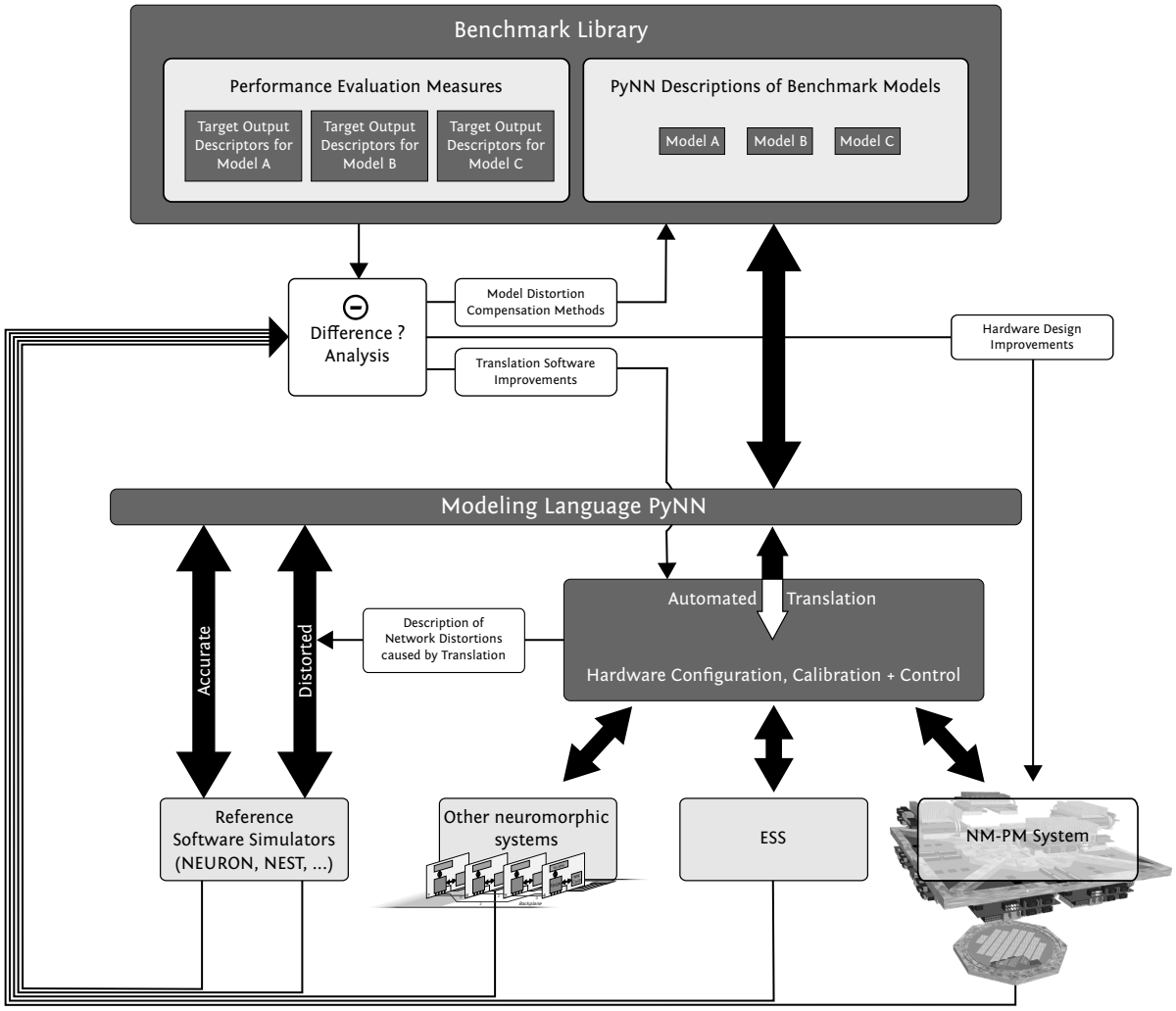


Figure 2.2.3: Hardware and Model evaluation workflow. Description in section 2.2.4. This figure has been adapted from [6].

2.3 Physical Model Platform: BrainScaleS-2

2.3.1 Omnibus

The omnibus implements a minimal subset of the Open Core Protocol (OCP) bus standard. The basis is a request and response path with each its own handshake as illustrated in 2.3.1.

A notable supported simple extension is the request phase byte enables (`MByteEn`). Some of the implemented modules are detailed below.

! A large part of the implemented omnibus modules only support master commands `IDLE`, `WR` and `RD`, as well as slave responses `NULL` and `DVA`.

2.3.1.1 Bus_if_split

The module Bus_if_split implements a simple one-hot encoded bus splitter from a single slave interface to two master interfaces. The address bit that determines the routing is set via `SELECT_BIT`. If `MAddr[SELECT_BIT]` is set, the incoming command is forwarded to the `out_1` interface, else to `out_0`.

An internal buffer can store the target of the master commands in flight to allow proper arbitration of the slave response accept handshakes. Thus, the depth of this buffer limits the maximum number of transactions in flight and can be set via the `NUM_IN_FLIGHT` parameter.

This module supports request phase byte enables.

2.3.1.2 Bus_if_arb

The module Bus_if_arb implements a priority arbiter between two omnibus slave interfaces and one master interface. The first slave interface `in_0` always wins on simultaneous commands. The reset signal source can be selected which helps to prevent circular dependencies in more complex tree structures. If either `RESET_BY_0` *or* `RESET_BY_1` is set *only* the respective bus slave interface can reset the master interface. With both of the other parameter combinations the master interface is reset when either slave interface reset is pulled. The reset signal can be selected to be synchronous or asynchronous via the `RESET_SYNCHRONOUSLY` parameter.

An internal buffer can store the source of the master commands in flight to allow proper arbitration on the slave response return path. Thus, the depth of this buffer limits the maximum

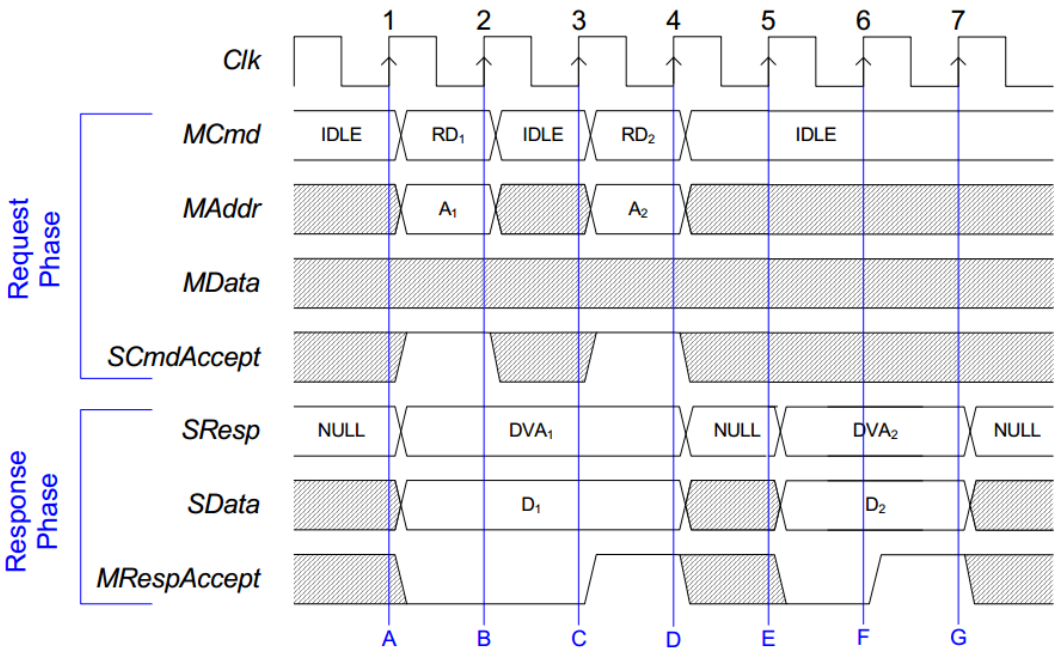


Figure 2.3.1: OCP protocol handshakes from the Open Core Protocol Specification. The command phase does not have to wait for the completion of the response phase to allow transactions in flight.

number of transactions in flight and can be set via the `NUM_IN_FLIGHT` parameter.

This module supports request phase byte enables.

! Due to the priority arbiter implementation the first slave interface `in_0` can block the second interface for arbitrarily long times if it continuously provides back-to-back transactions.

2.3.1.3 Bus_delay

The module Bus_delay implements a register stage. It allows to split the combinatorial paths to relax timing constraints in a larger and more complex bus, but increases the latency in both the command and response path. The reset signal can be selected to be synchronous or asynchronous via the `RESET_SYNCHRONOUSLY` parameter.

This module supports request phase byte enables.

2.3.1.4 Bus_reg_target

The module Bus_reg_target implements an omnibus slave for simple register read and write accesses. Its omnibus interface port is called `bus`. It provides `NUM_REGS` registers of type `logic[REG_WIDTH]`. For each of these registers, it also provides a signal `reading` and a signal called `writing`, indicating whether a register is currently read from or written to, respectively.

The module implements a state machine on the registers `regs_i`, `bus.SResp` and `bus.SData`. These internal registers are only modified if the base address matches, i.e., if `bus.MAddr & (BASE_MASK & ~OFFSET_MASK) == BASE_ADDR` and if a read or write request arrives. In case of a write access, the internal register `regs_i[bus.MAddr]` is set to `bus.MData`. In case of a read access, `bus.SData` is set to `regs_in[bus.MAddr]`.

Access to the registers is determined based on the following parameters:

ADDR_WIDTH determines the total width of the field `MAddr` of the omnibus slave interface.

OFFSET_MASK determines the address space used for accessing the individual registers. The register number is selected based on `MAddr & OFFSET_MASK`.

BASE_MASK determines the bits for comparison with the BASE_ADDR together with the OFFSET_MASK.

BASE_ADDR gates all updates to the internal state as stated above.

WRITEABLE is of type `logic [0:NUM_REGS-1]` and contains one bit for each register that determines whether a register is writeable.

This module supports request phase byte enables.

2.3.1.5 m4 macro

The omnibus m4 macro defined in `hicann-dls/units/omnibus/m4/bus.m4` allows for simple and easily readable construction of bus trees. All implicit connections within the tree are handled by the macro and do not clutter the source code.

The bus description is contained between `bus_begin` and `bus_end` within systemverilog code. Any `master` port has to be matched by name to an existing systemverilog `Bus_if.slave` interface. For each `slave` port a systemverilog `Bus_connect` module has to be instantiated with the `in` port matching by name to `<M4_BUS_PREFIX>_slave_<M4_SLAVE_NAME>.slave`. While processing the bus description from top to bottom all internal output connections of modules are generated with a linear increasing index suffix. Input connections of bus modules are connected to the lowest available indices at each individual level. To understand the functionality in detail it is advised to take a look at the macro processed versions of existing bus top level files.

2.3.2 Routing

This document is supposed to serve as an introduction to the event routing facilities on the HICANN-X neuromorphic ASIC. It shall serve as an entry point for users wishing to develop their own routing. Therefore its aim is to be *logically* correct while not concerning with implementation details. At the current stage congestion effects of multiple events sharing a resource at the same time are disregarded.

2.3.2.1 Crossbar (Layer-1)

The crossbar is the central logic for distribution of events on the HICANN-X. It's also called layer-1 in contrast to layer-2 for chip-external events. It has a set of inputs and a set of outputs of the same event type. The event type in the crossbar is 14-bit wide. All bits in an event are treated equally within the crossbar. Its values are never altered within the crossbar. Figure 2.3.2 shows the crossbar. Inputs from the left are broadcasted horizontally, outputs on the top merge incoming events vertically. Crossbar nodes connect a horizontal input line with a vertical output line and are described in detail in section 2.3.2.1.

Node

A crossbar node connects a horizontal input line to a vertical output line. It conditionally forwards events based on the rule

$$\text{event} \;\&\; \text{mask} \overset{!}{=} \text{target}, \tag{2.3.1}$$

where event is the event value and the mask and the target are configurable 14-bit wide values. Therefore it allows selection of forwarded events based on their content.

2.3.2.2 External events (Layer-2)

Chip-external events are used for communication with the outside world, e.g. for feeding-in external spike sources or recording a neuron's spikes. External events are also called layer-2 (L2) events in contrast to the layer-1 events within the routing crossbar. The chip has one external event input and one external event output channel. The external event type is 16-bit wide. The external events are connected exclusively to the four input and output channels of the crossbar from and to the L2 via value-based split of the one channel to four channels, cf. section 2.3.2.2.

Connection to the crossbar

The crossbar has four input and four output channels for external events. The single external event input channel is forwarded to the crossbar by the following rule:

$$\text{crossbar input channel} = (\text{event} \,\&\, (0\text{x}3 \ll 14)) \gg 14, \tag{2.3.2}$$

where the crossbar input channel to be selected is calculated by taking the value of the highest two bits in the external event's value. The crossbar event forwarded consists of the lower 14 bits of the external event's value:

$$\text{crossbar input event} = \text{event} \,\&\, 0\text{x}3\text{fff}. \tag{2.3.3}$$

Connection from the crossbar

The four crossbar external event output channels are merged to the single external event output channel by the following rule:

$$\text{event} = \text{crossbar output event} \mid (\text{crossbar output channel} \ll 14), \tag{2.3.4}$$

where the external event is calculated by taking the crossbar output event and extending its 14 bit value by the crossbar output channel index placed in the two highmost bits of the 16 bit external event's value.

2.3.2.3 PADI-Bus

The PADI-Bus connects the synapse driver crossbar output channels to the synapse drivers. There are four PADI-buses per hemisphere of the chip. There is a one-to-one relation between synapse driver crossbar output channels and PADI-buses. Its event type is 11-bit wide. A PADI-event is formed from a crossbar event by discarding the highmost three bits:

$$\text{PADI event} = \text{crossbar output event} \,\&\, 0\text{x}7\text{ff}. \tag{2.3.5}$$

2.3.2.4 Synapse driver

Each hemisphere of the chip features a block of 128 synapse drivers. Synapse driver input events are PADI-events. Each synapse driver is connected to one of the four PADI-buses of its hemisphere. Figure 2.3.3 shows the synapse drivers' static connection to their PADI-buses. The synapse driver is connected to one PADI-bus by the following rule:

$$\text{PADI-bus} = \text{synapse driver on block} \,\%\, 4. \tag{2.3.6}$$

Each PADI-bus therefore is connected to 32 synapse drivers.

PADI-event filtering

A synapse driver filters incoming events based on a static and a configurable entity. The static entity is the index of the synapse driver on its PADI-bus (right in fig. 2.3.3). The configurable

entity is a 5-bit mask. The highmost five bits of incoming PADI-events are processed by the following rule:

$$\text{event high-bits \& mask} \stackrel{!}{=} \text{index on PADI-bus \& mask.} \tag{2.3.7}$$

For all bits enabled in the mask the event's high-bits have to match the synapse driver's static index on the PADI-bus. Receiving of PADI-events can be enabled/disabled by the `enable_receiver` switch.

Forwarding to synapse rows

If the incoming PADI-event filter forwards the event, the lower six bits may be forwarded as synaptic event to the two synapse rows connecting to one driver. Forwarding of the lower five bits of these six bits can be enabled/disabled via the `enable_address_out` switch, the highmost bit is always sent. For each row driving the excitatory and the inhibitory line can be enabled via `enable_excitatory` and `enable_inhibitory`. Events are broadcasted to all synapses of the synapse driver's two synapse rows.

2.3.2.5 Synapse

A synapse is located within one synapse row, of which there are 256 per chip hemisphere (and two per synapse driver). A synapse is vertically connected to exactly one neuron. Each synapse locally compares its configurable 6-bit label value to the incoming event's value and elicits an event on match:

$$\text{event} \stackrel{!}{=} \text{local label} \tag{2.3.8}$$

For each synapse column (equivalent to a neuron index), connectivity of the excitatory and inhibitory lines can be enabled in the `ColumnCurrentSwitch` object.

2.3.2.6 Neuron

The neurons are located in one row per hemisphere with 256 neurons per row. Each neuron is connected statically to exactly one neuron output channel into the routing crossbar. Figure 2.3.4 shows the mapping of neurons to neuron output channels. Upon eliciting a spike, the neuron generates a crossbar event (14-bit wide), where the lower eight bits are configurable per neuron and the highmost six bits are unconnected (clamped to 0).

...

(Info: the rest of the NM-PM1 description is 'consortium internal' and therefore not included in this 'public' version of the document.)

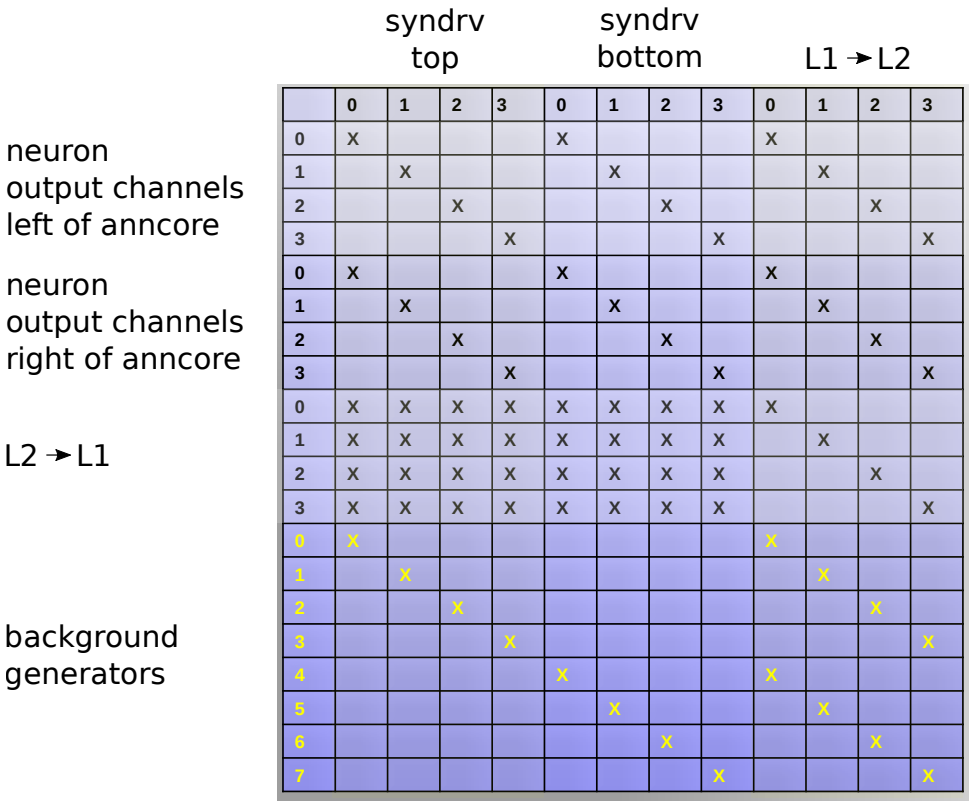


Figure 2.3.2: Schematic of the routing crossbar (taken from J. Schemmel). Inputs coming from the left are broadcasted horizontally, outputs merge incoming events vertically. Each X describes a location of a crossbar node, which connects a horizontal input line with a vertical output line. Their location is static, intersections without an X can't be connected. The input channels can be divided into neuron output channels in groups of four for the left and the right half of the analog neural network core, cf. section 2.3.2.6, four external input channels (from the L2) into the crossbar, cf. section 2.3.2.2 and eight on-chip background event generators. The output channels can be divided into four synapse driver input channels per hemisphere, cf. section 2.3.2.3, and four external output channels from the crossbar (to the L2), cf. section 2.3.2.2.

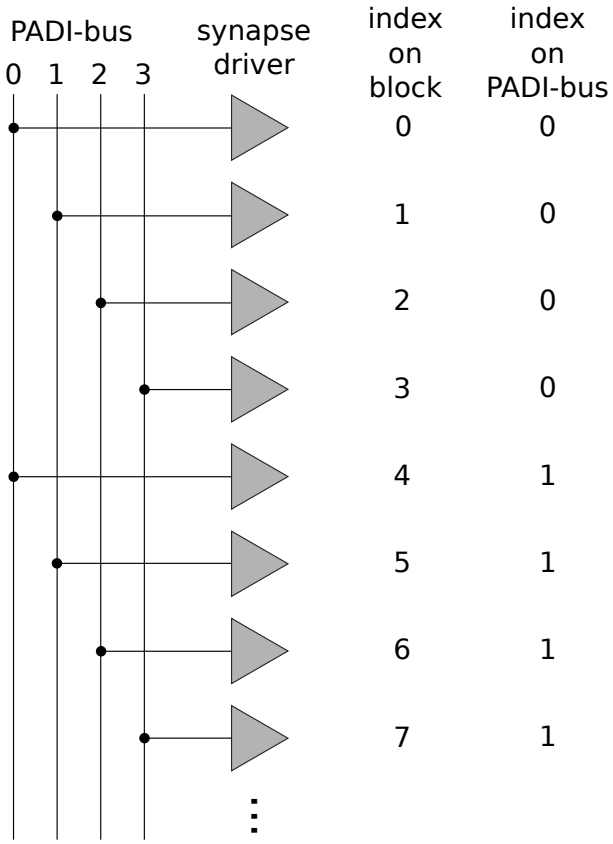


Figure 2.3.3: Schematic of the synapse driver to PADI-bus connectivity. Each synapse driver is connected to one PADI-bus. Adjacent synapse drivers are connected to different buses.

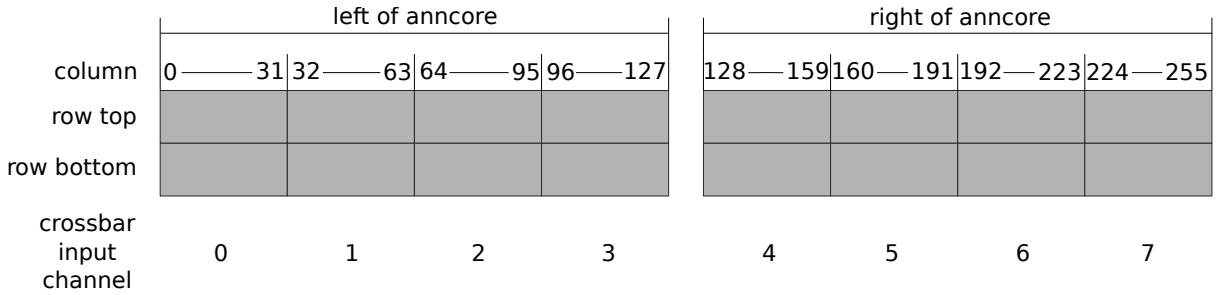


Figure 2.3.4: Schematic of the connectivity of the neurons. Neurons in horizontal blocks of 32 per row are connected to one neuron output channel into the crossbar. Vertically adjacent neurons are connected to the same output channel.

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

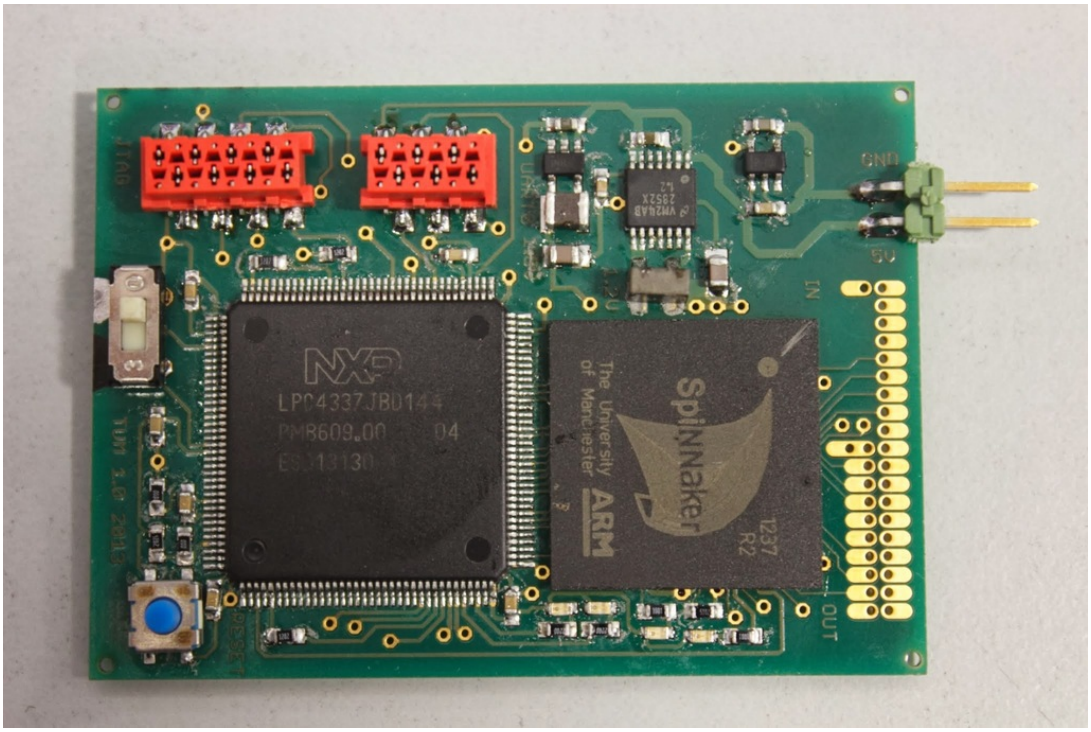
Physical machines used to deploy Platform, locations, etc.

The NM-MC1 system uses the SpiNNaker chip, which is now in production.

Various configurations are possible:

Single node board

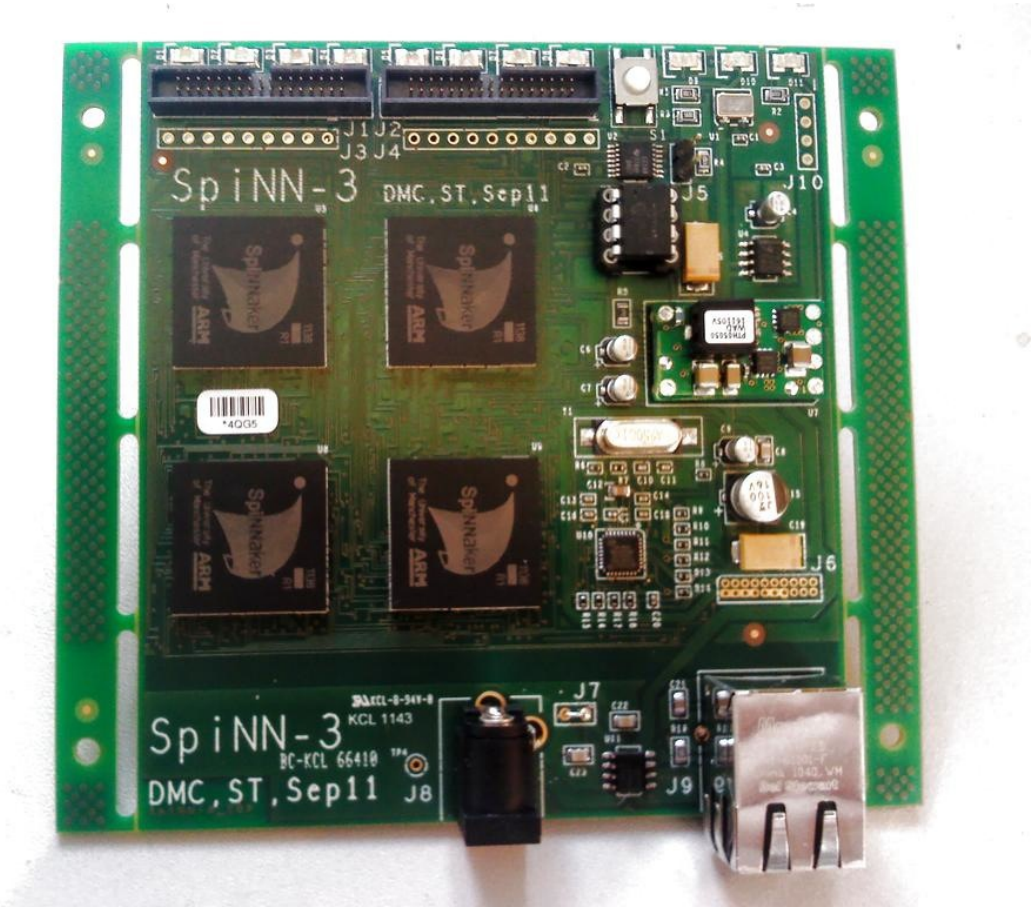
Although this has not been developed in Manchester, it is an interesting Neurorobotics platform, developed by Jörg Conradt at TU Munchen:



Four node board

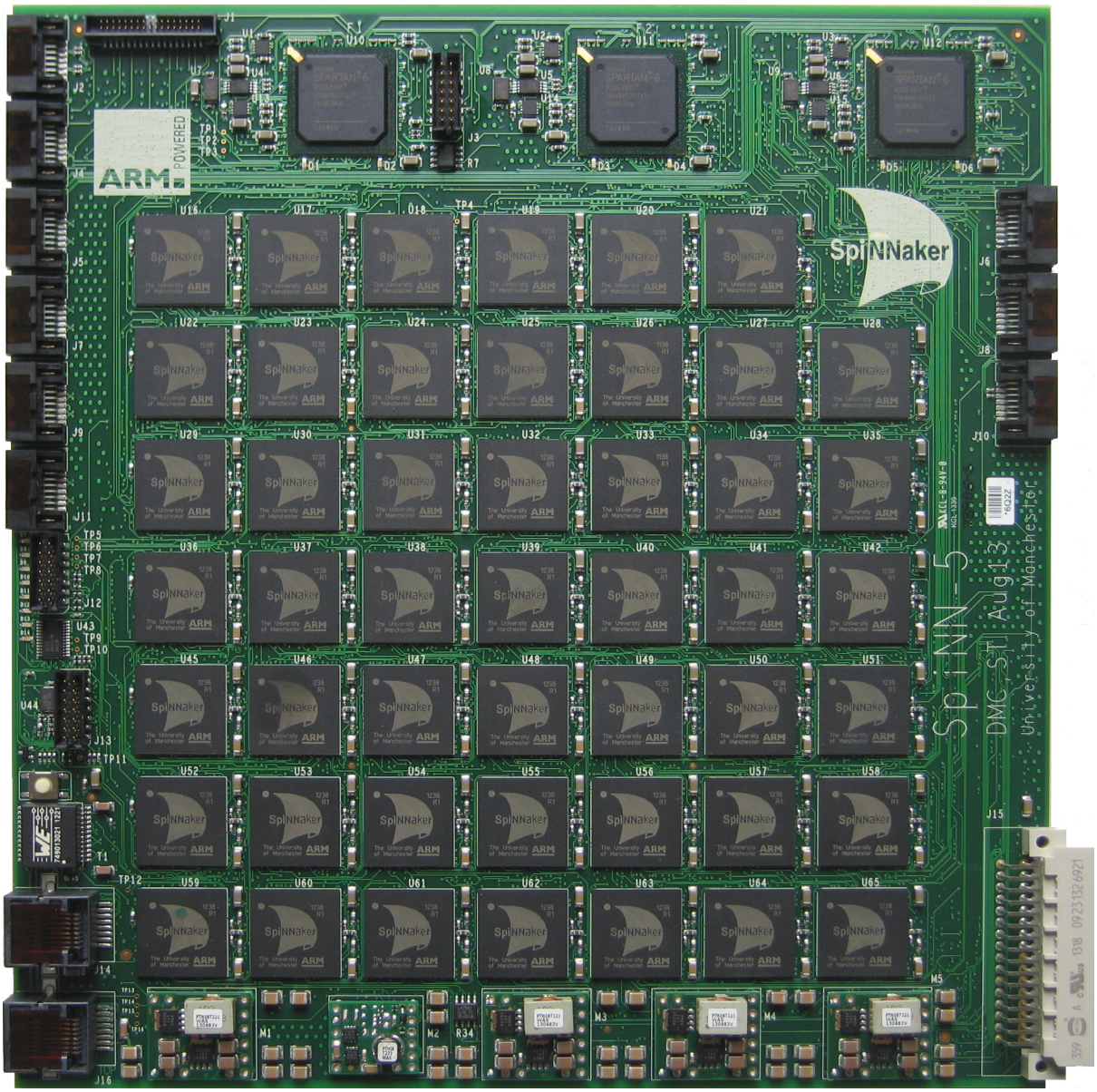
This consists of a four SpiNNaker chips and has been developed by in Manchester for use in collaboration with the Robotics group at Plymouth University. It can be used in an iCUB robot.

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 subproject, 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.

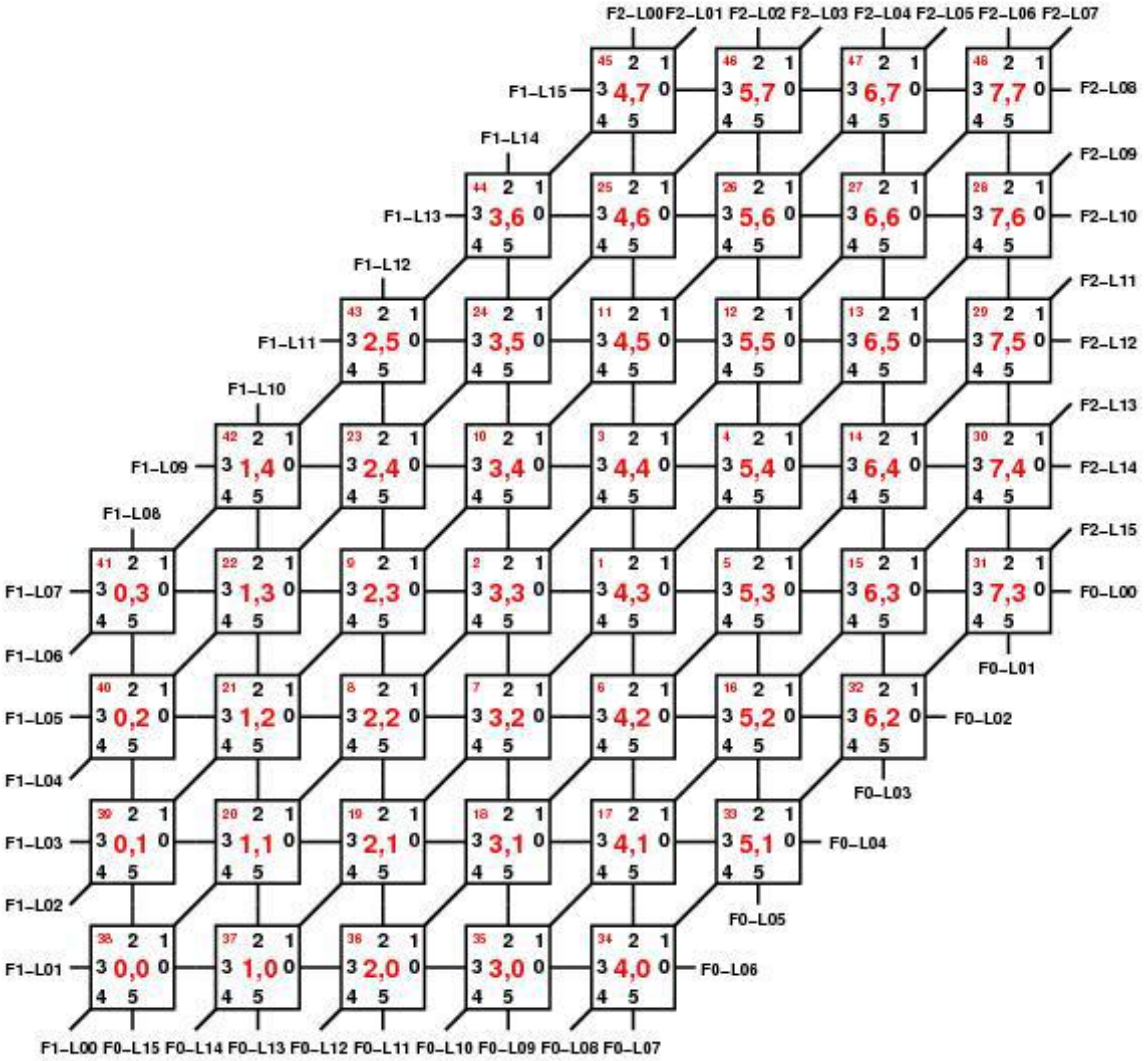
Expected Delivery: 2nd quarter 2014.

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
F2-L08
F1-L14
F2-L09
F1-L13
F2-L10
F1-L12
F2-L11
F1-L11
F2-L12
F1-L10
F2-L13
F1-L09
F2-L14
F1-L08
F2-L15
F1-L07
F0-L00
F1-L06
F0-L01
F1-L05
F0-L02
F1-L04
F0-L03
F1-L03
F0-L04
F1-L02
F0-L05
F1-L01
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.

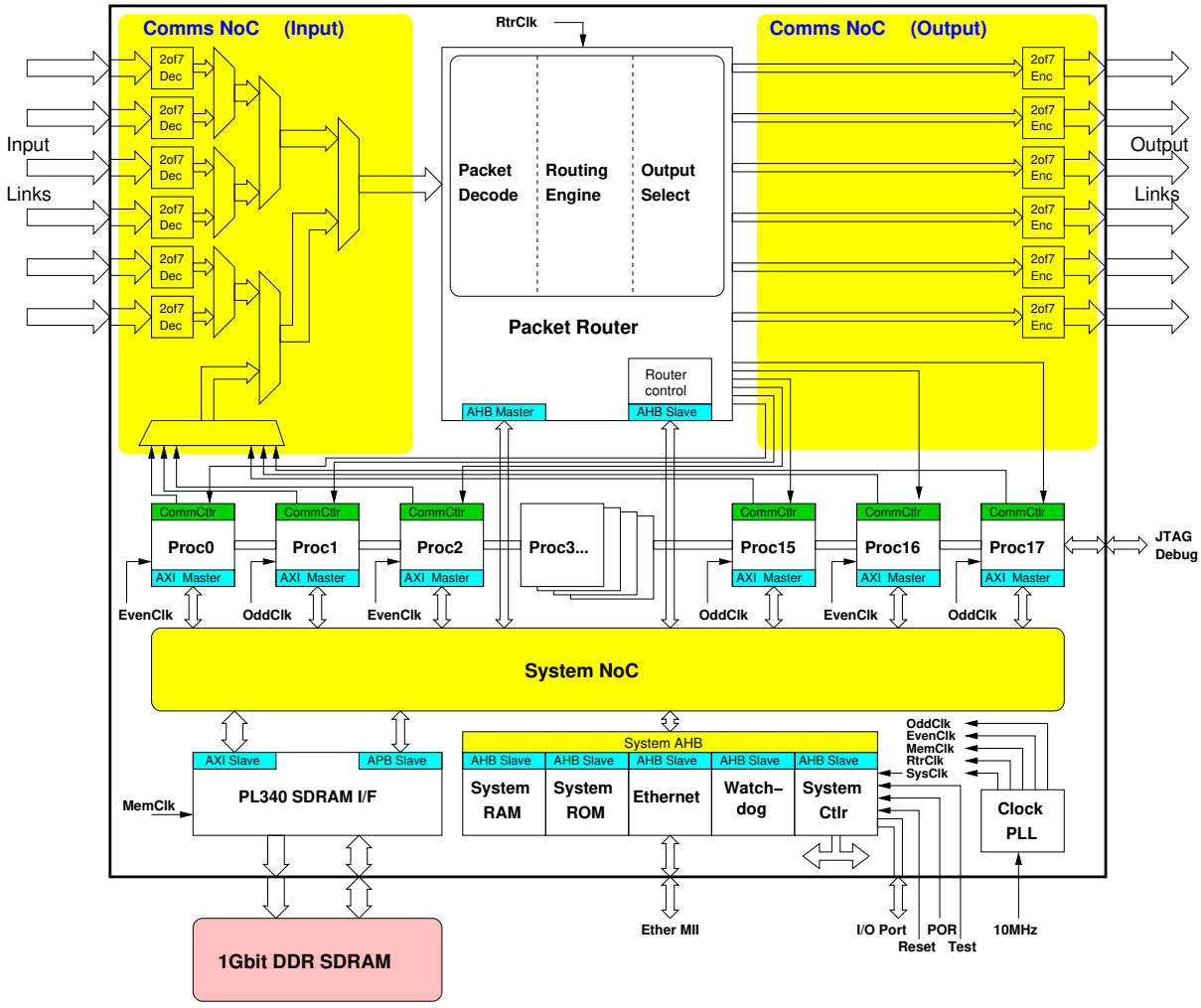
Change history

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.

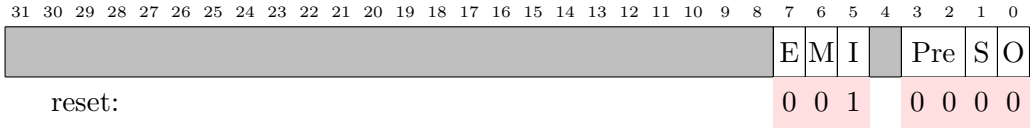
3.2.1.2 System-on-Chip hierarchy

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

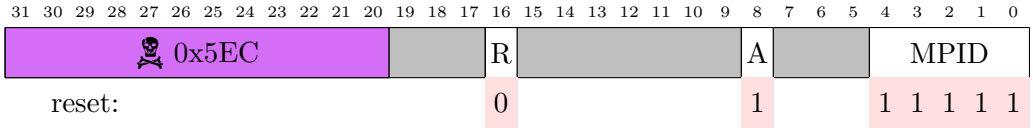
3.2.1.3 Register description convention

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.

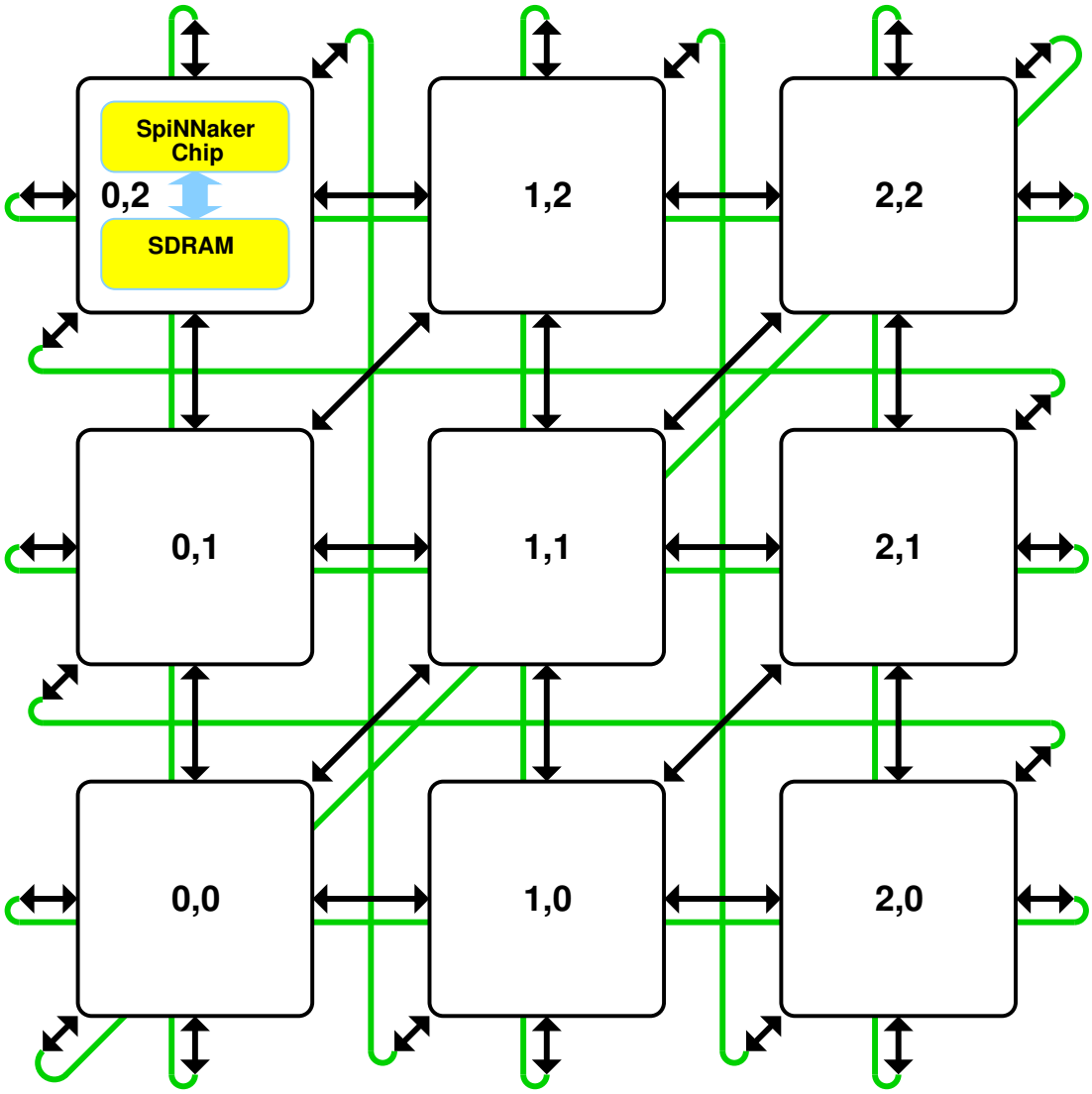
Certain registers in the System Controller have protection against corruption by errant code:



- 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 $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$, 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.

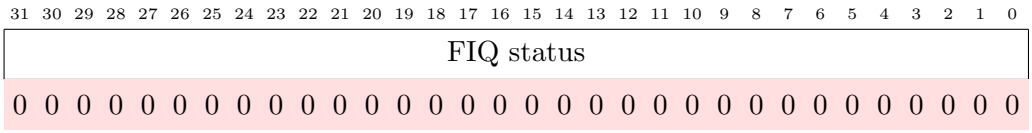
3.2.5.3 Register details

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

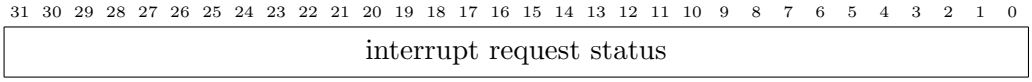
This read-only register yields the set of active IRQ requests (after masking).

r1: FIQ status



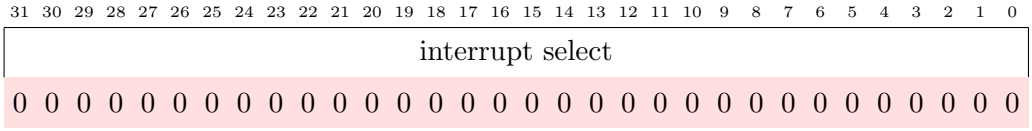
This read-only register yields the set of active FIQ requests (after masking).

r2: raw interrupt status



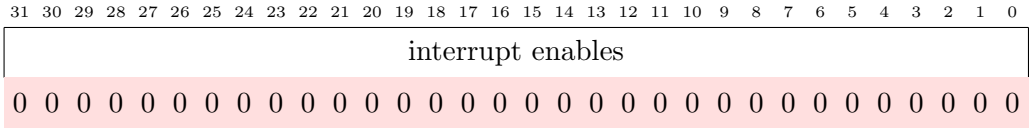
This read-only register yields the set of active input interrupt requests (before any masking).

r3: interrupt select



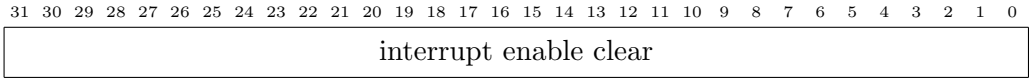
This register selects for each of the 32 interrupt inputs whether it gets sent to IRQ (0) or FIQ (1). The reset state is not specified (though is probably '0'?); all interrupts are disabled by r4 at reset.

r4: interrupt enable register



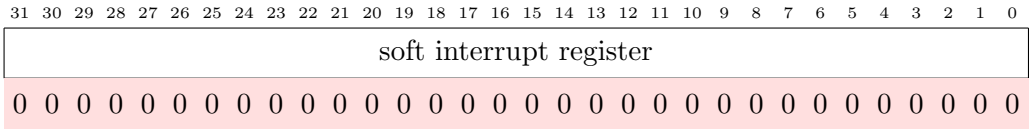
This register disables (0) or enables (1) each of the 32 interrupt inputs. Writing a '1' sets the corresponding bit in r4; writing a '0' has no effect. Interrupts are all disabled at reset.

r5: interrupt enable clear



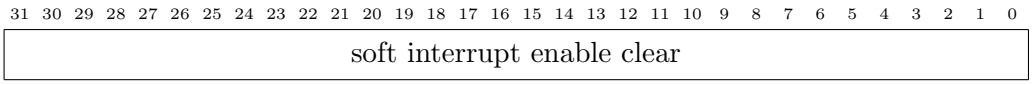
This write-only register selectively clears interrupt enable bits in r4. A '1' clears the corresponding bit in r4; a '0' has no effect.

r6: soft interrupt register



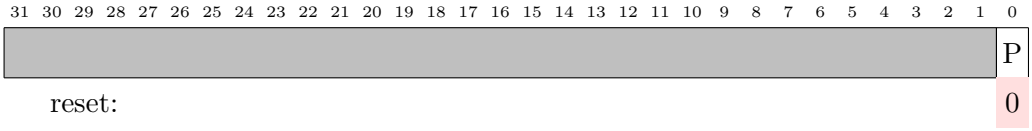
This register enables software to force interrupt inputs to appear high (before masking). A '1' written to any bit location will force the corresponding interrupt input to be active; writing a '0' has no effect. The reset state for these bits is unspecified, though probably '0'?

r7: soft interrupt register clear



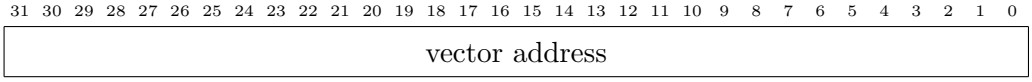
This write-only register selectively clears soft interrupt bits in r6. A '1' clears the corresponding bit in r6; a '0' has no effect.

r8: protection



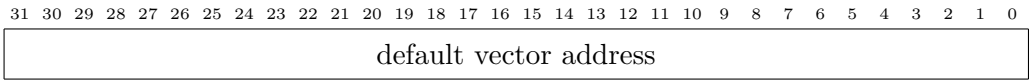
If the P bit is set VIC registers can only be accessed in a privileged mode; if it is clear then User- mode code can access the registers.

r9: vector address



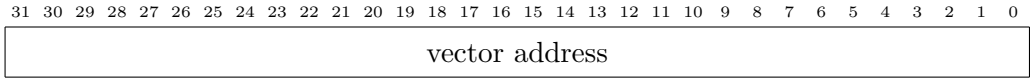
This register contains the address of the currently active interrupt service routine (ISR). It must be read at the start of the ISR, and written at the end of the ISR to signal that the priority logic should update to the next priority interrupt. Its state following reset is undefined.

r10: default vector address



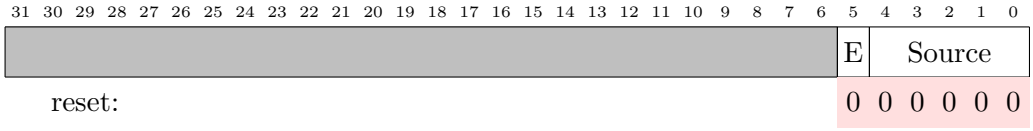
The default vector address is used by the 16 interrupts that are not vectored. Its state following reset is undefined.

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.

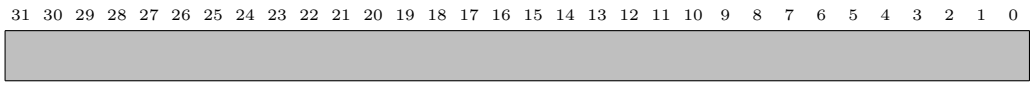
r2/10: Timer X control

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:

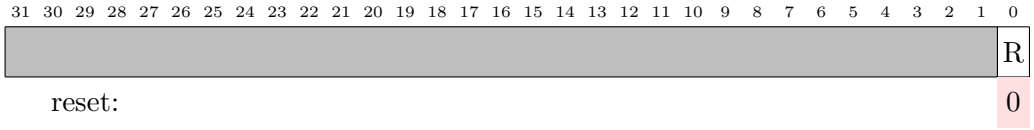
Name	bits	R/W	Function
E: Enable	7	R/W	enable counter/timer (1 = enabled)
M: Mode	6	R/W	0 = free-running; 1 = periodic
I: Int enable	5	R/W	enable interrupt (1 = enabled)
Pre: TimerPre	3:2	R/W	divide input clock by 1 (00), 16 (01), 256 (10)
S: Timer size	1	R/W	0 = 16 bit, 1 = 32 bit
O: One shot	0	R/W	0 = wrapping mode; 1 = one shot

r3/11: Timer X interrupt clear



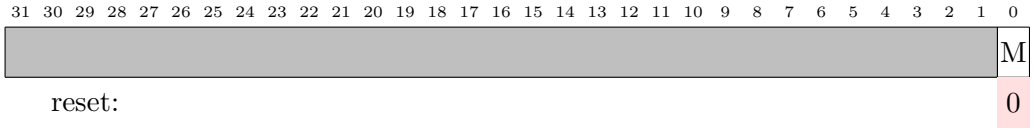
Any write to this address will clear the interrupt request.

rr4/12: Timer X raw interrupt status



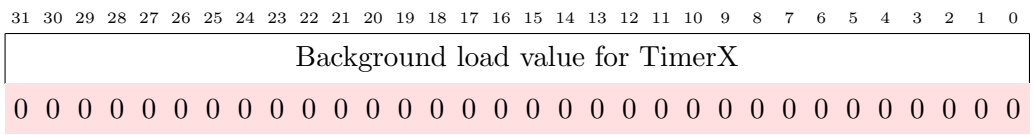
Bit zero yields the raw (unmasked) interrupt request status of this counter/timer.

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

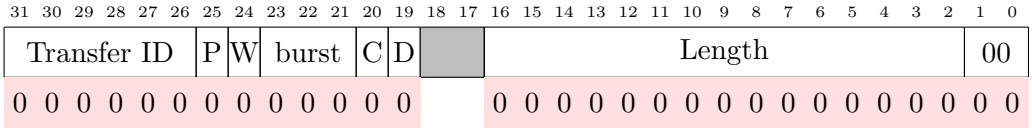
The 32-bit start byte address on the system interface. Note that a read is considered a data movement from a source on the system bus to a destination on the TCM bus. DMA transfers are word-aligned, so bits[1:0] are fixed at zero.

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

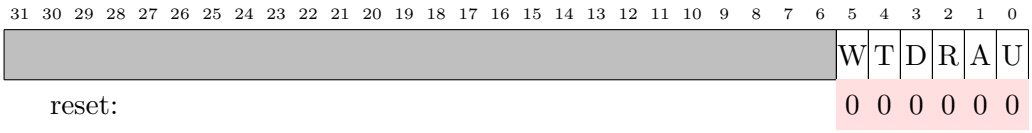
The TCM as currently implemented has a size of 64Kbytes (for the data TCM). A DMA transfer must of necessity either take as a source or a destination the TCM, justifying this restriction. DMA transfers are word-aligned, so bits[1:0] are fixed at zero.

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

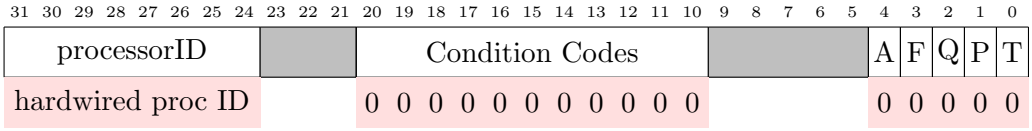


The functions of these fields are described in the table below:

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

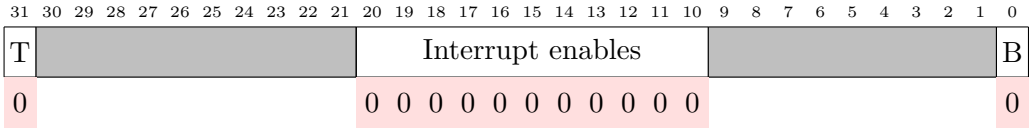
clear the error codes [16:13] and restart DMA operation. The terminated transfer must be restarted explicitly by software if this is required.

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

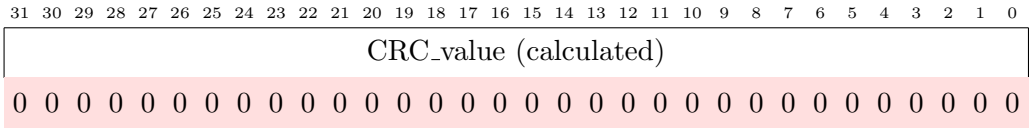
The DMA controller passes four interrupt request lines to the VIC:

- 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.

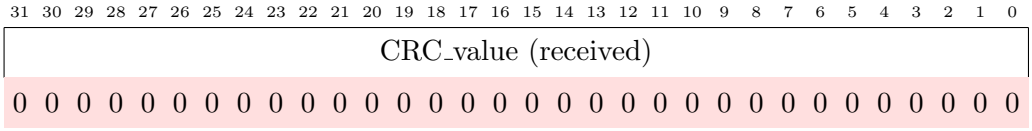
The system-wide slow timer is a clock signal that sets bit[31] on every rising edge, thereby raising an interrupt request to the VIC, and is cleared by writing a 0 to bit[31]. Writing a 1 to bit[31] has no effect.

r7: CRCC - Calculated CRC



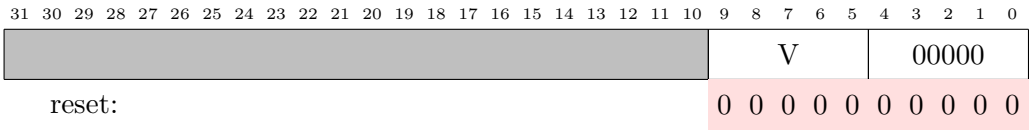
This is the 32-bit CRC value calculated by the DMA CRC unit.

r8: CRCC - Received CRC



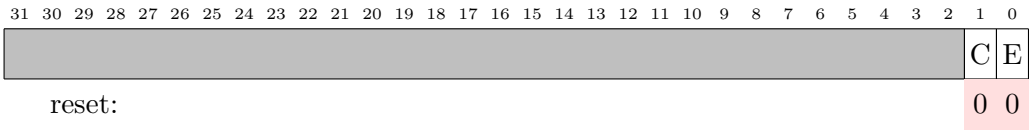
This is the 32-bit CRC value read in the block of data loaded by a DMA transfer.

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.

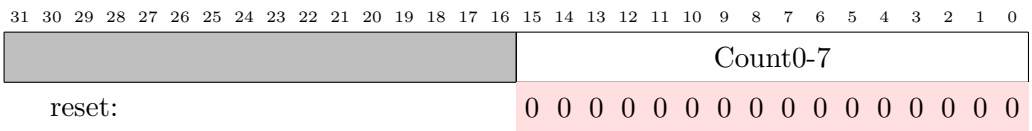
r10: StatsCtl - Statistics counters control



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



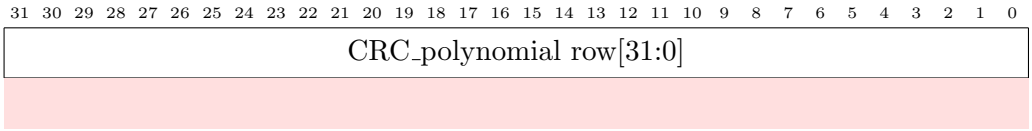
These eight 16-bit counter registers record statistics relating to the latency of DMA transactions across the System NoC. Count0 records the number of transactions that complete in 0-127 clock cycles, Count1 128-255 clock cycles, and so on up to Count7 which counts transactions that complete in 896+ 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.

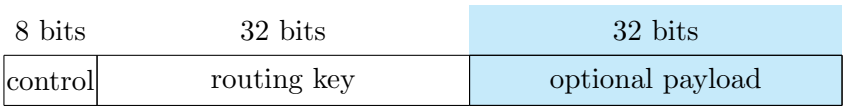
3.2.8.1 Features

- 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).

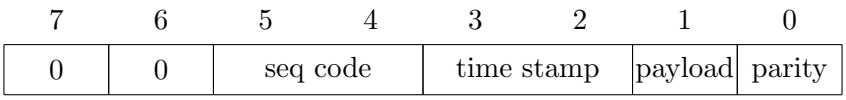
3.2.8.2 Packet formats

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 payloadless response packet is returned which will indicate the error status.

3.2.8.5 Register summary

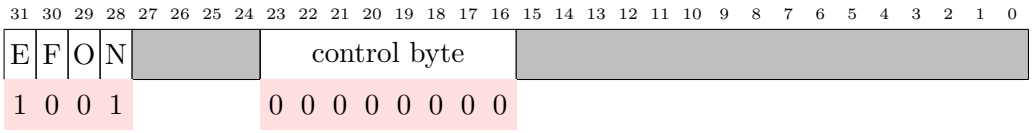
Base address: 0x20000000 (buffered write), 0x10000000 (unbuffered write).

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:

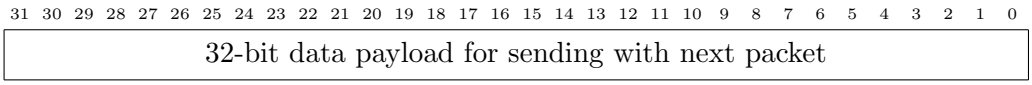
Name	bits	R/W	Function
E: empty	31	R	Tx buffer empty
F: full	30	R/W	Tx buffer full (sticky)
O: overrun	29	R/W	Tx buffer overrun (sticky)
N: not full	28	R	Tx buffer not full, so it is safe to send a packet
control byte	23:16	W	control byte of next sent packet

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.

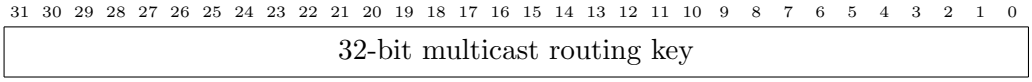
r1: TDR - transmit data payload



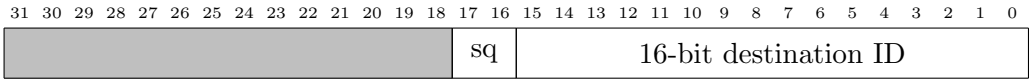
If data is written into TDR before a send key or dest ID is written into TKR, the packet initiated by writing to TKR will include the contents of TDR as its data payload. If no data is written into TDR before a send key or dest ID is written into TKR the packet will carry no data payload.

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).



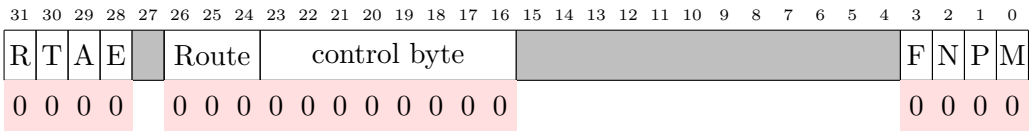
If bits[23:22] of the control register are 01 the Communication Controller is set to send point-to- point packets and the value written into TKR should include the 16-bit address of the destination chip in bits[15:0] and a sequence code in bits[17:16]. (See ‘seq code’ on page 32.)



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

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 received data payload																															

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

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
					Route											p2p source ID															
reset:					1	1	1									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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

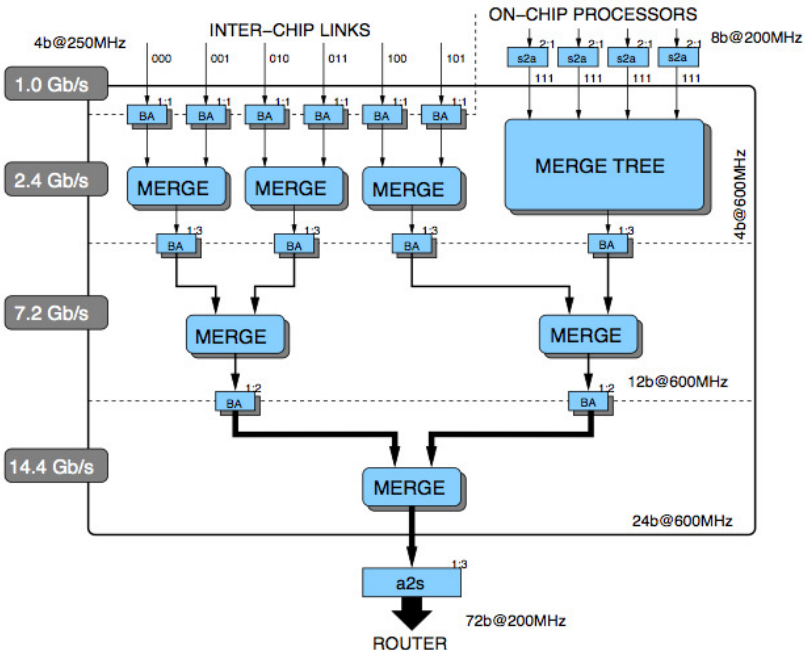
The Communications NoC carries packets between the processors on the same or different chips. It plays a central role in the system architecture. Its connectivity to the other components is shown the the chip block diagram in 'Chip organization' on page 5.

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

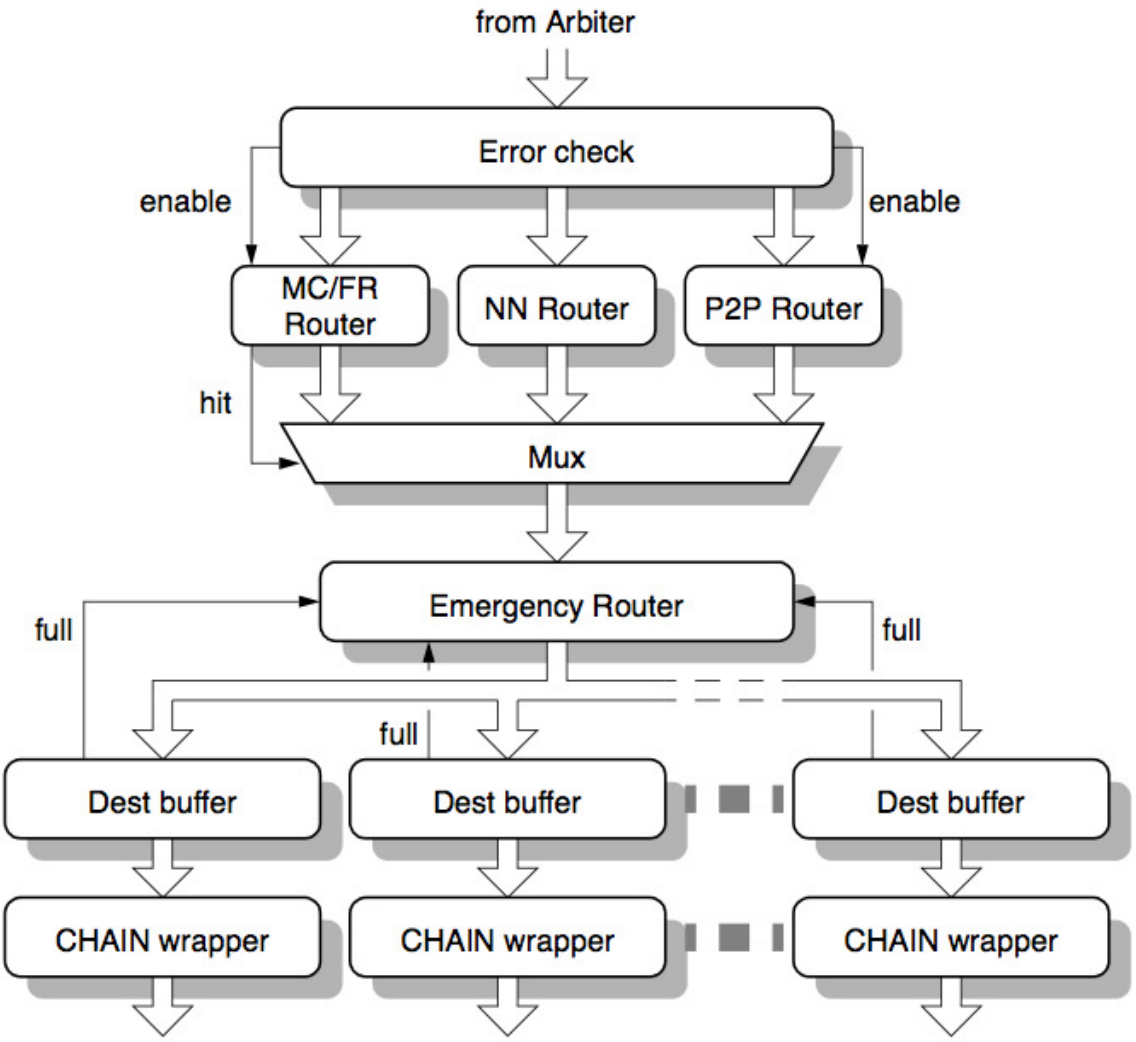
Packets arrive from other nodes via the link receiver interfaces and from internal processor nodes and are presented to the router one-at-a-time. The Arbiter is responsible for determining the order of presentation of the packets, but as each packet is handled independently the order is unimportant (though 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.

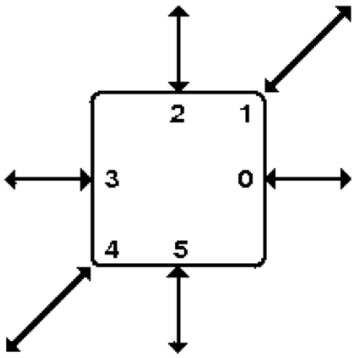
Note that the MC Router CAM is not initialised at reset. Before the Router is enabled all CAM entries must be initialised by software. Unused mask[] entries should be initialised to 0000000, and unused key[] entries should be initialised to FFFFFFFF. This invalidates every bit in the word, ensuring that 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.

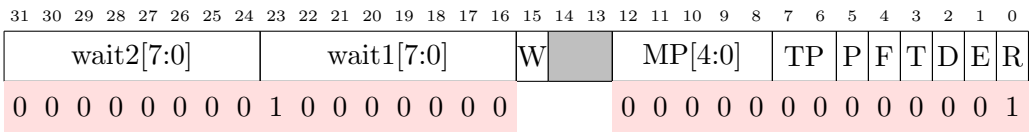
3.2.10.10 Register summary

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.

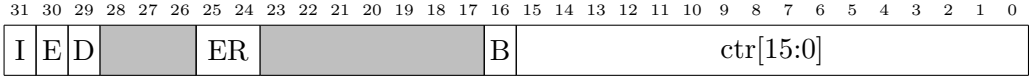
- wait2[7:0] is the number of cycles during which emergency routing will be attempted before the packet is dumped; wait2[] = 0 therefore effectively disables emergency routing.

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.



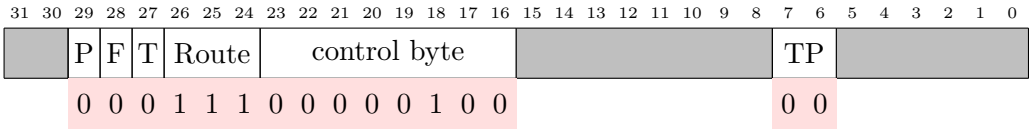
The functions of these fields are described in the table below:

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

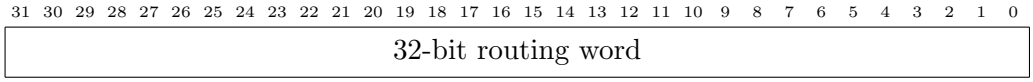
A packet which contains an error is copied to r2-5. Once a packet has been copied (indicated by bit[31] of r5 being set) any further error packet is ignored, except that it can update the sticky bits in r5 (and errors of the types specified in r0 are counted in r5).



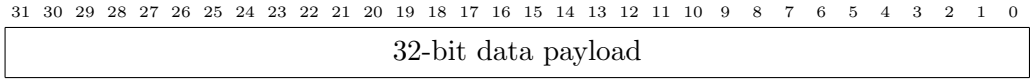
The functions of these fields are described in the table below:

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

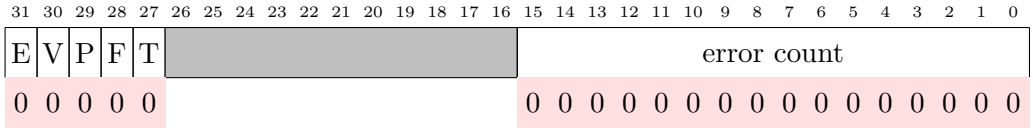


r4: error data payload



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.

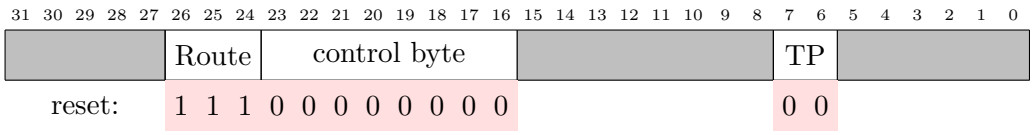


The functions of these fields are described in the table below:

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

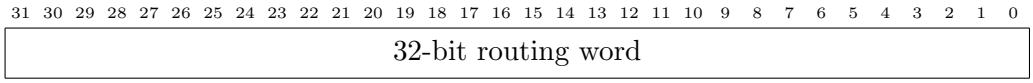
A packet which is dumped because it cannot be routed to its destination is copied to r6-10. Once a packet has been dumped (indicated by bit[31] of r10 being set) any further packet that is dumped is ignored, except that it can update the sticky bits in r10 (and can be counted by a diagnostic counter).



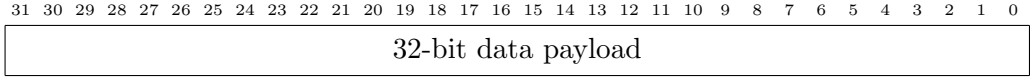
The functions of these fields are described in the table below:

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

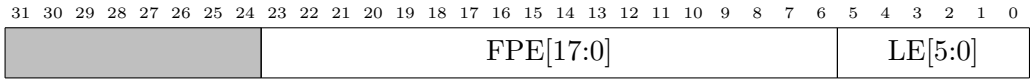
r7: dump routing word



r8: dump data payload



r9: dump outputs

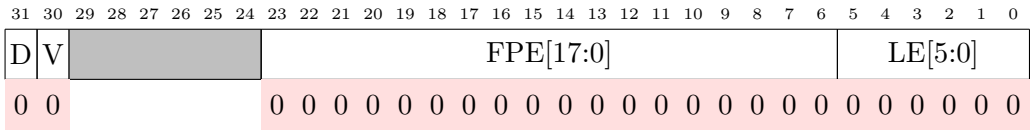


The functions of these fields are described in the table below:

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.

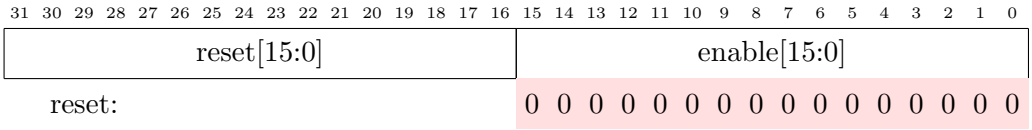


The functions of these fields are described in the table below:

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.



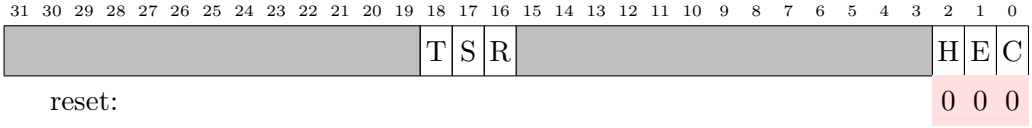
The functions of these fields are described in the table below:

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.

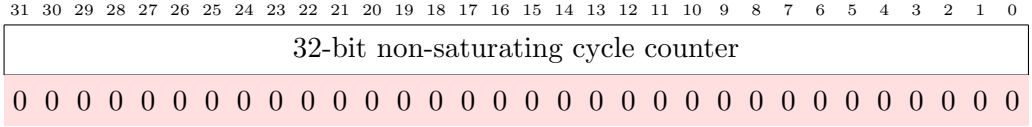


The functions of these fields are described in the table below:

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

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 non-saturating emergency router active cycle counter																															
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

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

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 non-saturating unblocked packet counter																															
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

r15, when enabled by r12, counts the number of packets which pass through undelayed by congested output links.

r16-31: packet delay histogram

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 non-saturating packet delay counter																															
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

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.

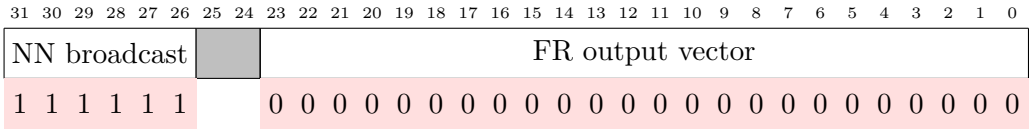
r32: diversion

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.

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
																				L5		L4		L4		L2		L1		L0	
reset:																				0	0	0	0	0	0	0	0	0	0	0	0

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



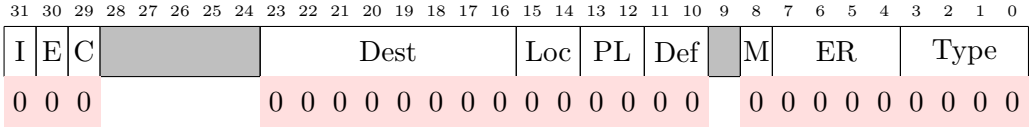
r33 routes fixed-route (type 3) packets to off-chip links and local processors in exactly the same way, with the same bit allocation, as an MC output vector as described in section 10.4 on page 39.

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

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 count value																															
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

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.

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
																					entry										M

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

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 key																															
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 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.

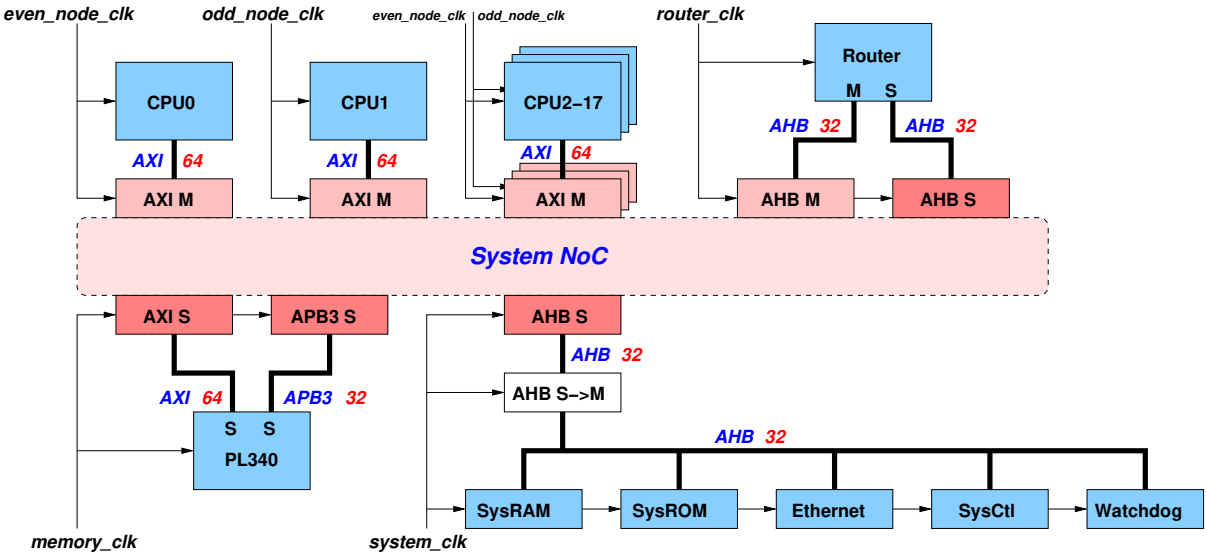
3.2.12 System NoC

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.

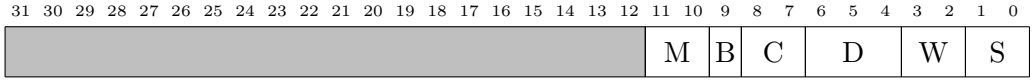
Restrictions on when registers may be modified

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

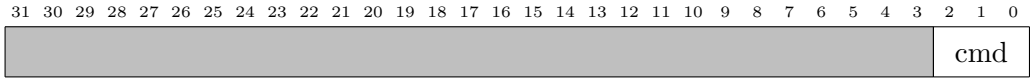
r0: memory controller status



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

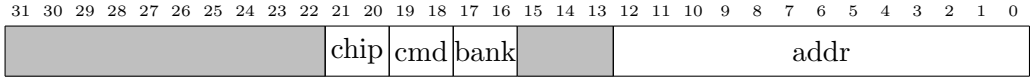
r1: memory controller command



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

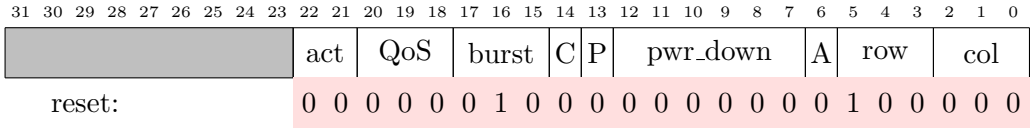
r2: direct command



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

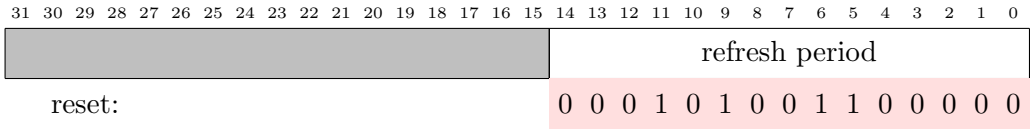
r3: memory configuration



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)

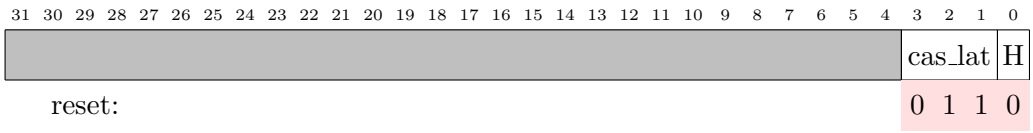
r4: refresh period



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

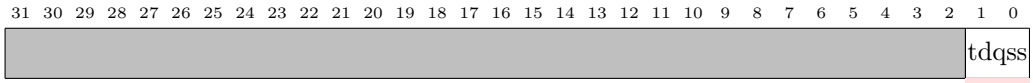
r5: CAS latency



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

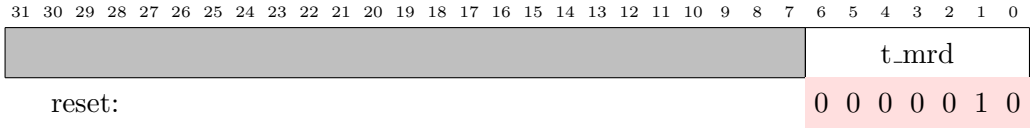
r6: t_dqss



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

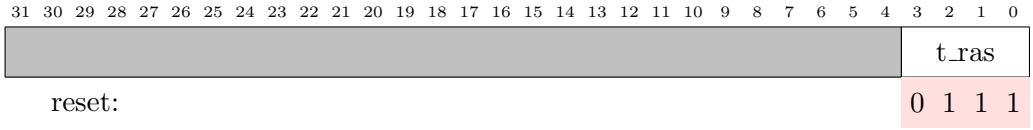
r7: t_mrd



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

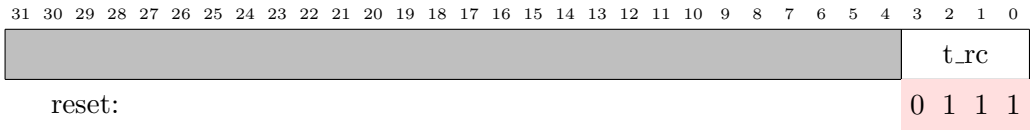
r8: t_ras



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

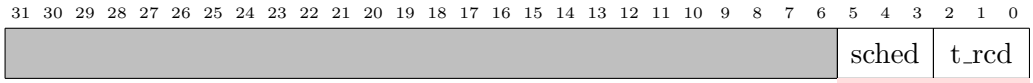
r9: t_rc



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

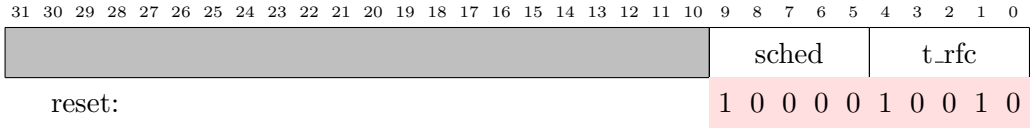
r10: t_rcd



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

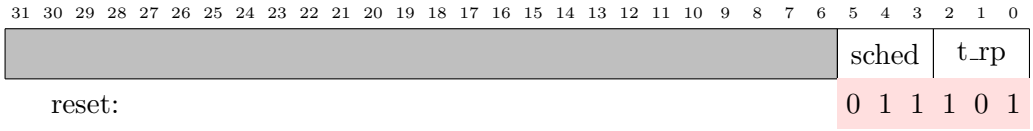
r11: t_rfc



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

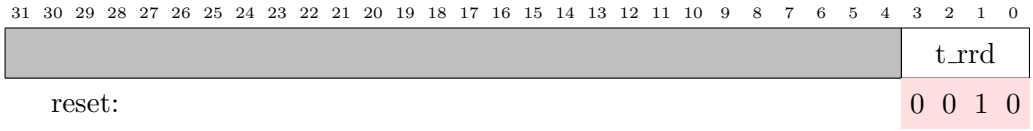
r12: t_rp



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

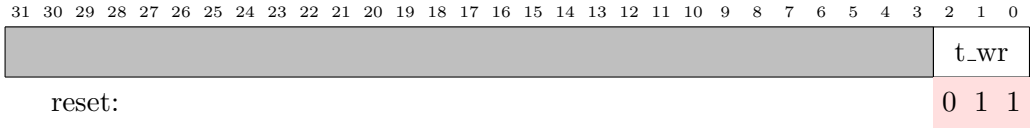
r13: t_rrd



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

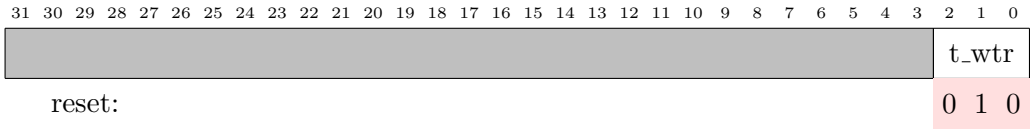
r14: t_wr



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

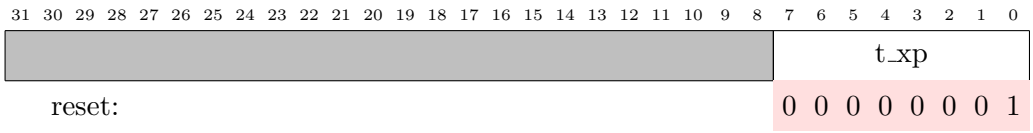
r15: t_wtr



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

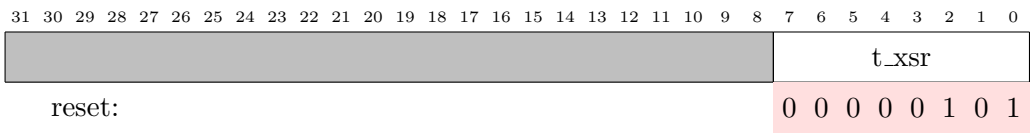
r16: t_xp



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

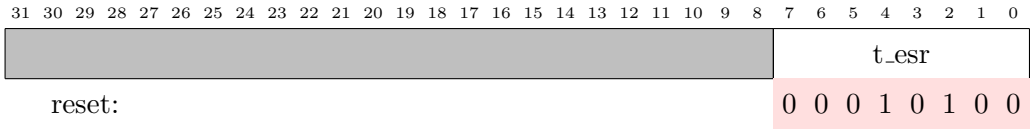
r17: t_xsr



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

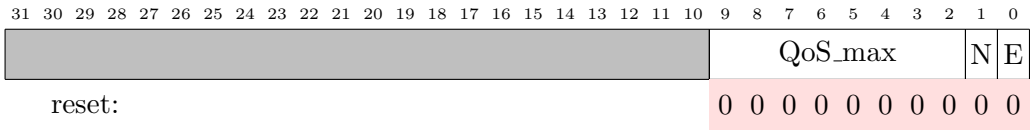
r18: t_esr



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

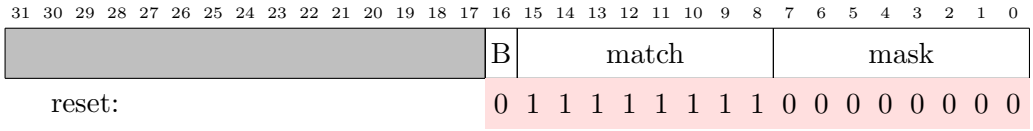
id_n_cfg



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



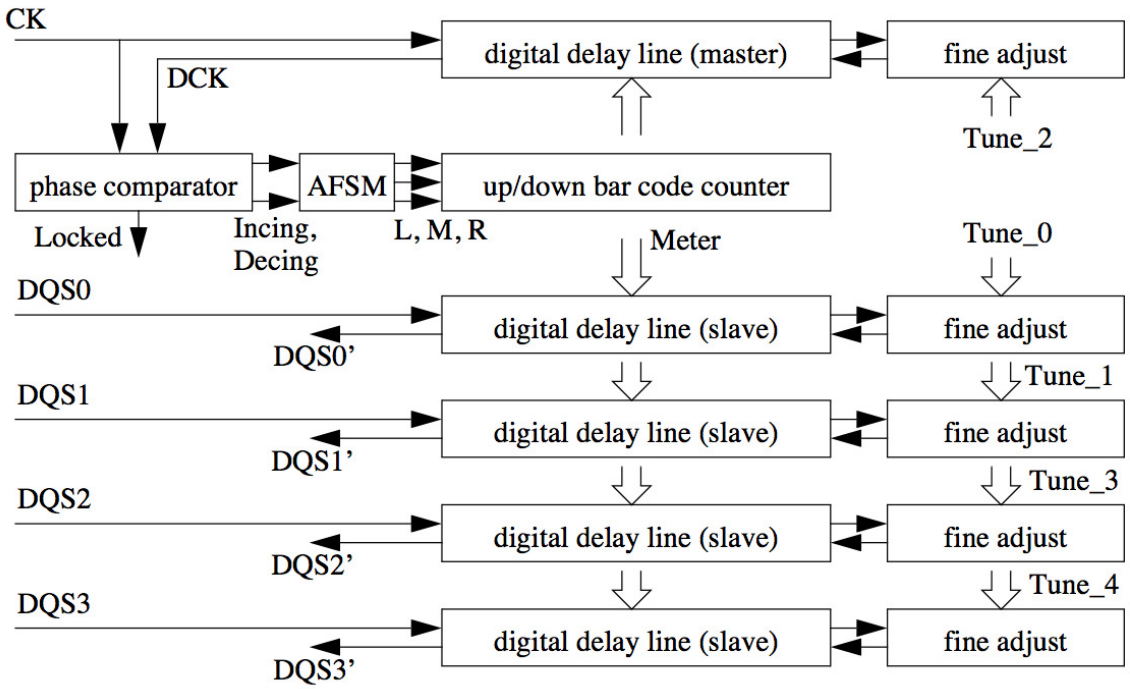
There is one of these registers for each external chip that is supported. The functions of these fields are described in the table below:

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

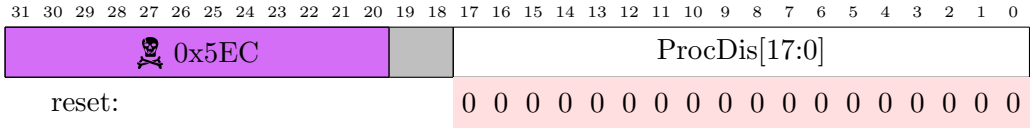
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
device												version				Year								# CPUs							
0	1	0	1	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0

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.

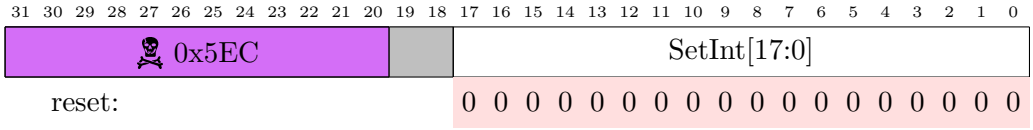
r1: CPU disable



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.

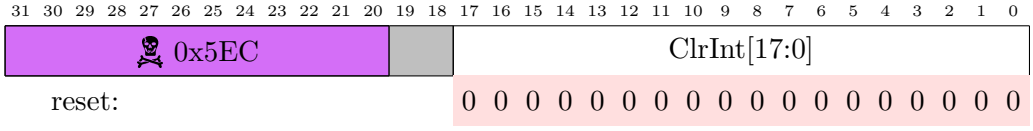
To ensure the processor is disabled in a low-power state it should be disabled and then reset via r9.Reading from this register returns the current status of all of the processor disable lines.

r2: Set CPU interrupt request



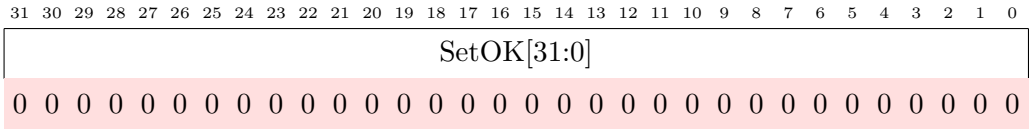
Writing a 1 to bit[n] (n = 0...17) will set an interrupt request to processor[n], which can be enabled/ disabled and routed to IRQ or FIQ by that processor's local Vectored Interrupt Controller (VIC - see page 12). Writing a 0 has no effect. For a write to be effective it must include a security code 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.

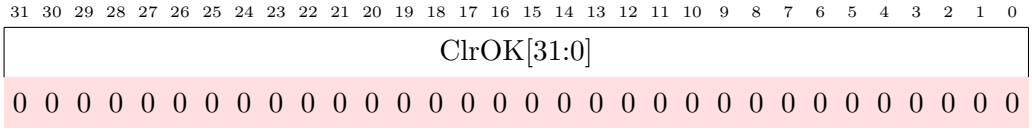
r4: Set CPU OK



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.

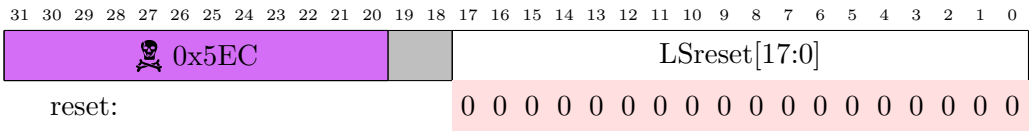
In normal use a processor will set its own bit after performing some functional self-testing. The Monitor Processor will read the register after the start-up phase to establish which processors are functional, and assign them tasks accordingly. The MP may attempt to restart faulty processors by resetting 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.

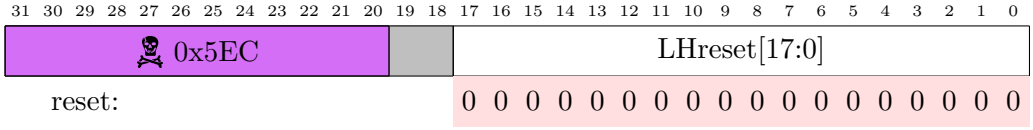
r6: CPU node soft reset - level



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.

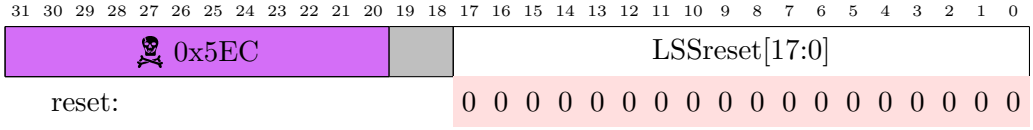
r7: CPU node hard reset - level



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

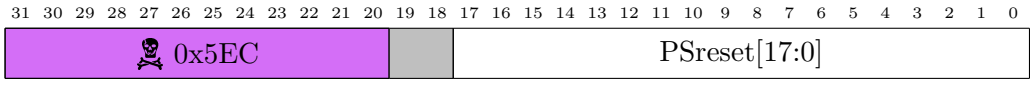


Writing a 1 to bit[n] (n = 0...17) will set a level on the reset input of a subsystem. For a write to be effective it must include a security code in bits [31:20]: 0x5ECXXXXX. Reading from this register returns the current status of this register, that is the level before the OR with the pulse reset 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

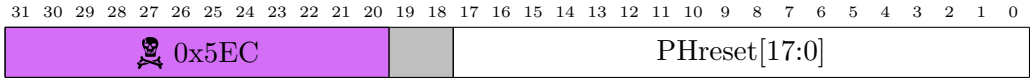
r9: CPU node soft reset - pulse



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.

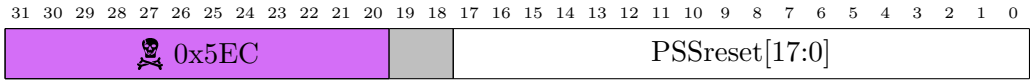
r10: CPU node hard reset - pulse



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.

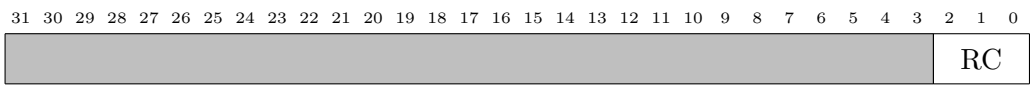
r11: Subsystem reset - pulse



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.

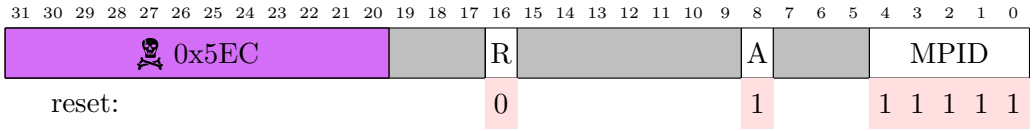
The Watchdog interrupt RC[2:0] = 100 only soft resets the Monitor Processor (with a 256 cycle pulse), and then only if this is enabled in r13.

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

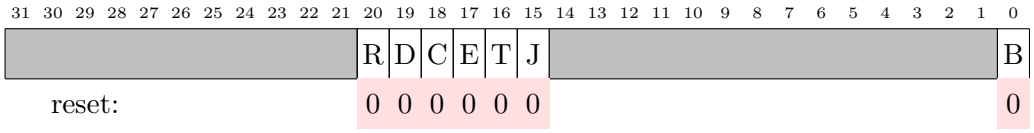
The ‘R’ bit causes the Watchdog interrupt signal to cause a soft reset of processor[MPID], which will override any interrupt masking by the Monitor Processor. In any case, this interrupt is available at all processor VICs and can therefore be enabled locally as an IRQ or FIQ source.

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:

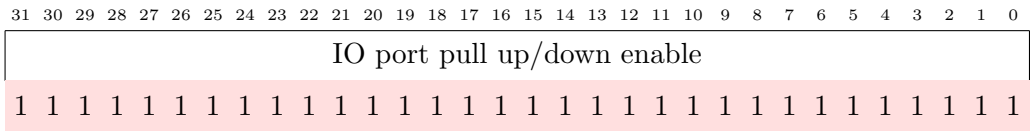
Name	bits	R/W	Function
R	20	R	read value on JTAG_RTCK pin
D	19	R	read value on JTAG_TDO pin
C	18	R	read value on Clk32 pin
E	17	R	read value on Ethermux pin
T	16	R	read value on Test pin
J	15	R/W	select on-chip (1) or off-chip (0) control of JTAG pins
B	0	R/W	map System ROM (0) or RAM (1) to Boot area

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

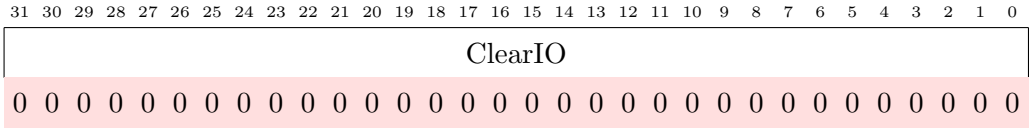
This register determines whether each IO port bit is an input (1) or an output (0). Setting a bit to an input does not invalidate the corresponding bit in r16 - that value will be held in r16 until explicitly changed by a write to r16. When read, this register returns the value last written.

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

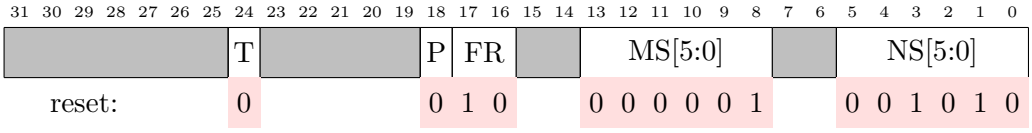
Writing a 1 sets the corresponding bit in r16. Writing a 0 has no effect.
Reading this register returns the values on the IO pins (if present).

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

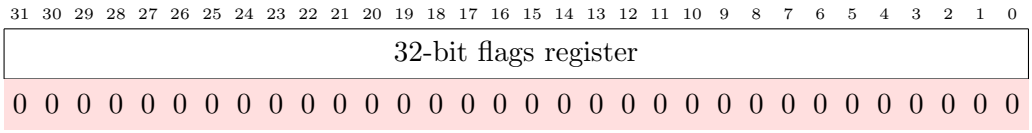


The function of these fields is described in the table below:

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 $10 \times \mathrm{NS/MS}$. 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.

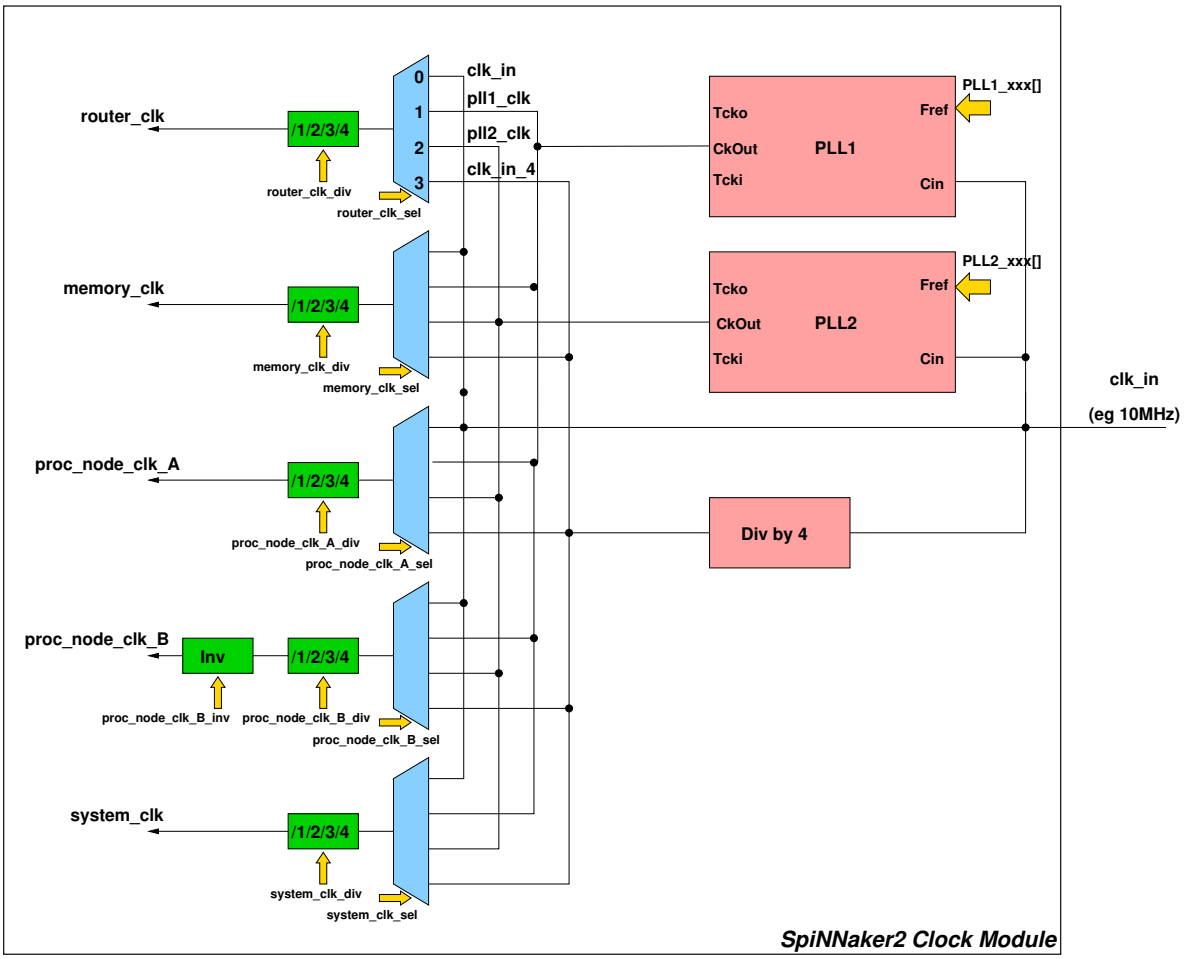
r23: Reset flags

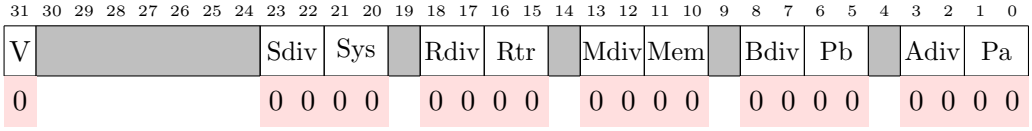
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
32-bit flags register																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

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:

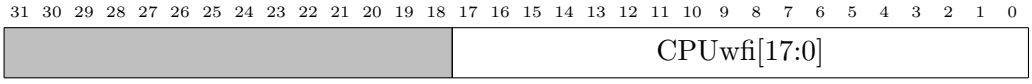
Name	bits	R/W	Function
V	31	R/W	invert CPU clock B
Sdiv[1:0]	23:22	R/W	divide System AHB clock by Sdiv + 1(= 1 − 4)
Sys[1:0]	21:20	R/W	clock selector for System AHB components
Rdiv[1:0]	18:17	R/W	divide Router clock by Rdiv + 1(= 1 − 4)
Rtr[1:0]	16:15	R/W	clock selector for Router
Mdiv[1:0]	13:12	R/W	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

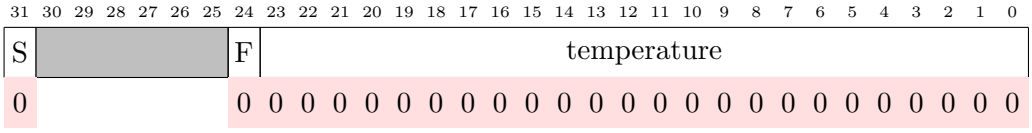


Each bit in this register indicates the state of the respective ARM968 STANDBYWFI (stand-by wait for interrupt) signal, which is active when the CPU is in its low-power sleep mode.

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 tem-

perature to be corrected for process and voltage variations.

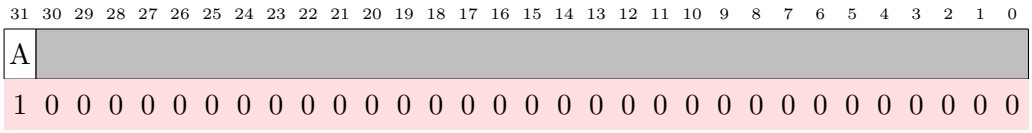


The functions of these fields are described in the table below:

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

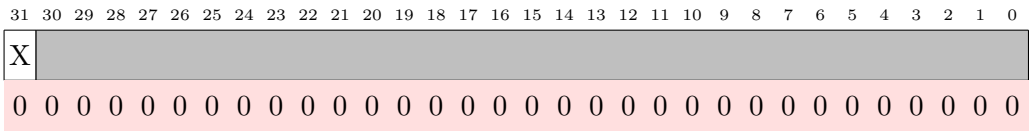


The same single-bit value 'A' appears in all registers r32 to r63.

'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.

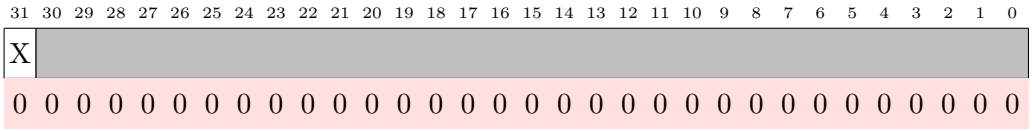
If A is clear when the read takes place, or N equals the current value in r13, then the value 0x00000000 is returned and A and r13 are unchanged.

r64-95: Test and Set



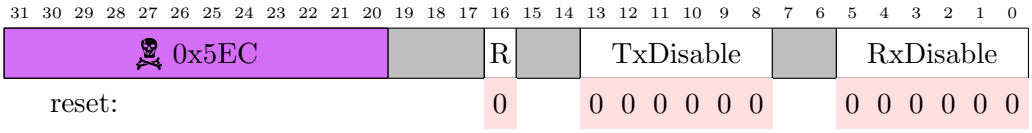
A unique single-bit value 'X' appears in each register r64 to r95. Reading each register returns 0x00000000 or 0x80000000 depending on whether its respective bit was clear or set prior to the read, and as a side-effect the bit is set by the read. Together with r96 to r127, these registers provide support 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

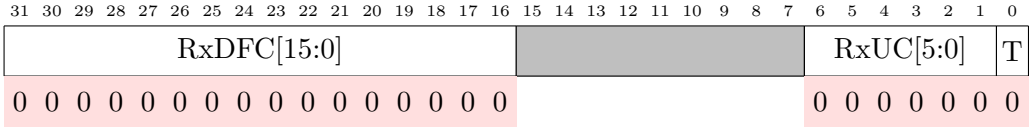
r0: General command register

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:

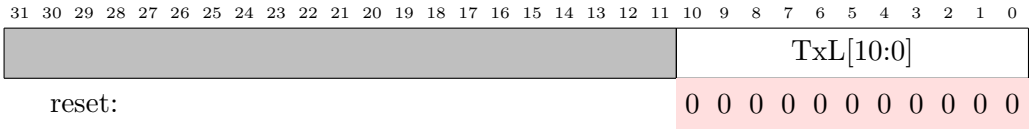
Name	bits	R/W	Function
H	10	R/W	Disable hardware byte reordering
D	9	R/W	Reset receive dropped frame count (in r1)
V	8	R/W	Receive VLAN enable
P	7	R/W	Receive promiscuous packets enable
B	6	R/W	Receive broadcast packets enable
M	5	R/W	Receive multicast packets enable
U	4	R/W	Receive unicast packets enable
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



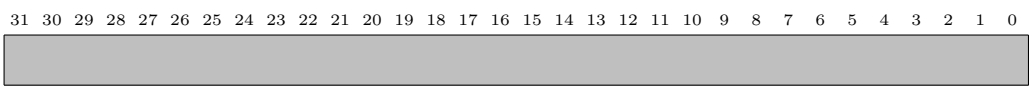
Name	bits	R/W	Function
RxDFC[15:0]	31:16	R	Receive dropped frame count
RxUC[5:0]	7:1	R	Received unread frame count
T	0	R	Transmit MII interface active

r2: Transmit frame length



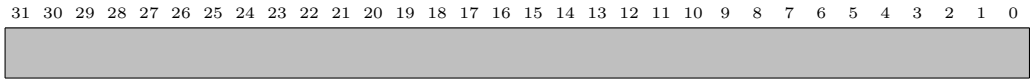
Name	bits	R/W	Function
TxL[10:0]	10:0	R/W	Length of transmit frame (60 - 1514 bytes)

r3: Transmit command register



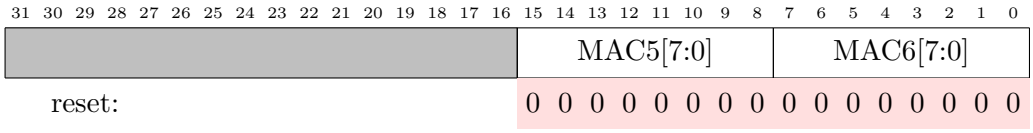
Any write to register 3 causes the transmission of a frame.

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.

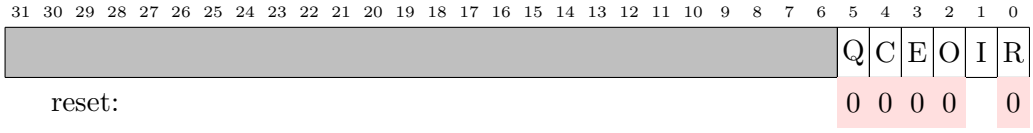
r6: MAC address high bytes



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

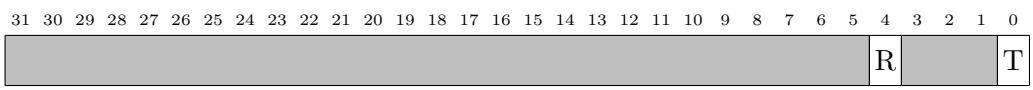
r7: PHY control



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

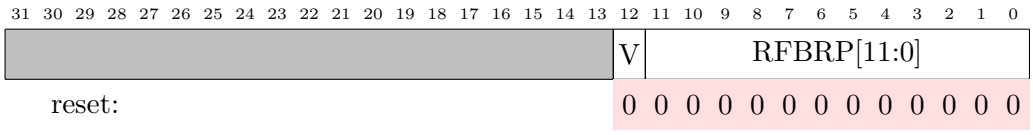


The functions of these fields are described in the table below:

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.

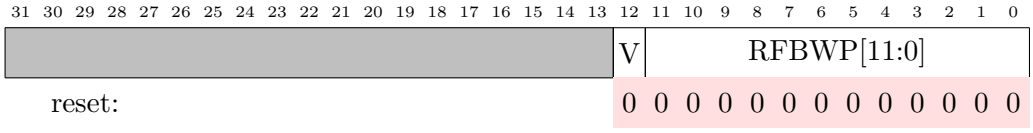
r9: Receive frame buffer read pointer



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

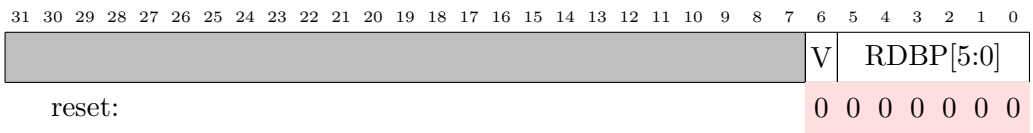
r10: Receive frame buffer write pointer



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

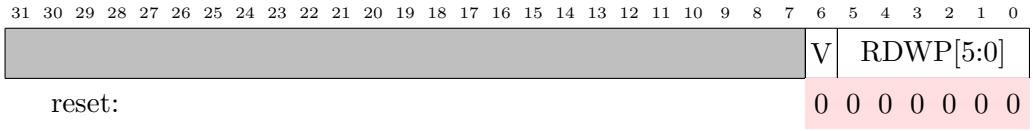
r11: Receive descriptor read pointer



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

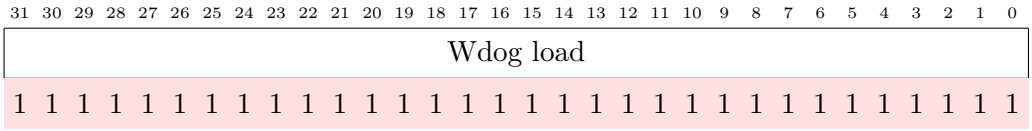
In addition, there are test and ID registers that will not normally be of interest to the programmer:

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.

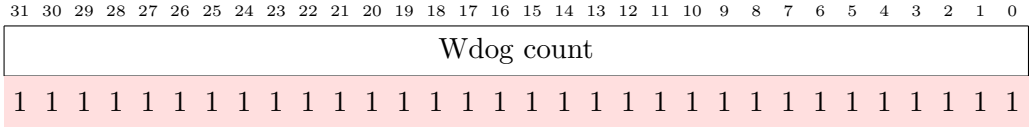
3.2.16.3 Register details

r0: Load



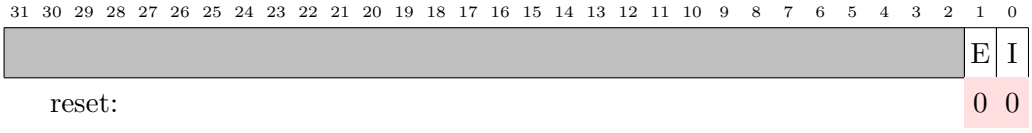
This read-write register contains the value the from which the counter is to decrement. When this register is written to, the count immediately restarts from the new value. The minimum value is 1.

r1: Count



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

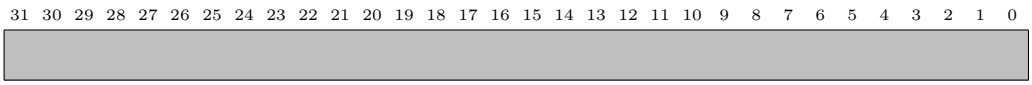


The functions of these fields are described in the table below:

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)

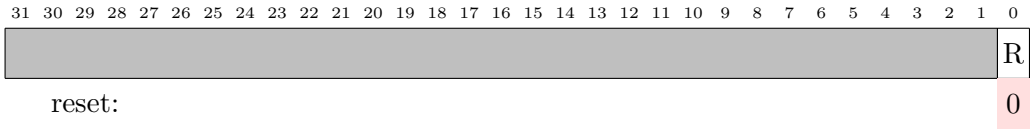
Once the Watchdog has been initialised both enables should be set to '1' for normal watchdog operation.

r3: Interrupt clear



A write of any value to this register clears the watchdog interrupt and reloads the counter from r1.

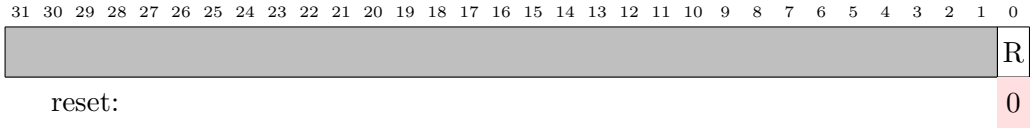
r4: Raw interrupt status



The function of this field is described in the table below:

Name	Offset	R/W	Function
R	0	R	Raw (unmasked) watchdog interrupt

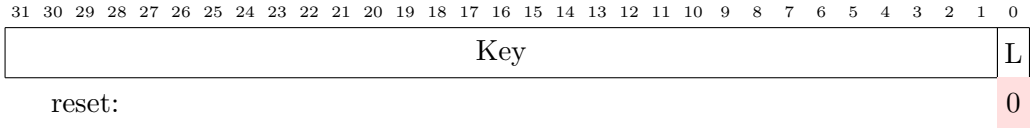
r5: Masked interrupt status



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

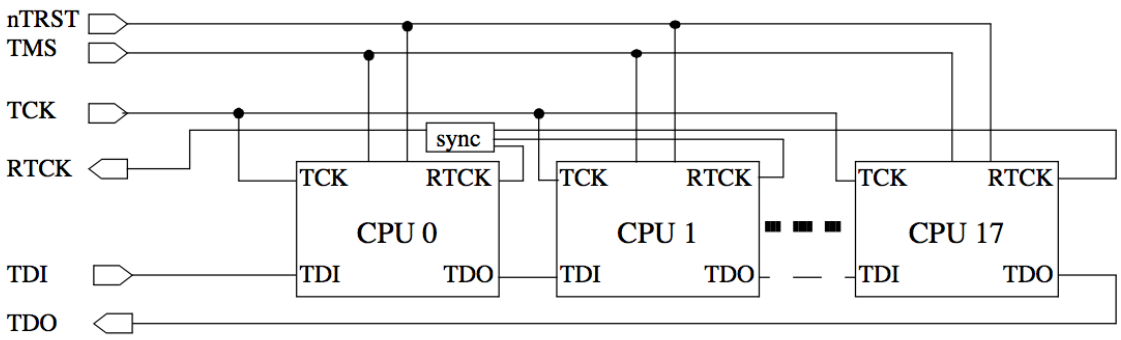
The JTAG IEEE 1149.1 system on the SpiNNaker chip provides access only to the ARM968 processors for software debug purposes. There is no provision for scan access to the SpiNNaker pins or other on-chip features.

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	NmA	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:

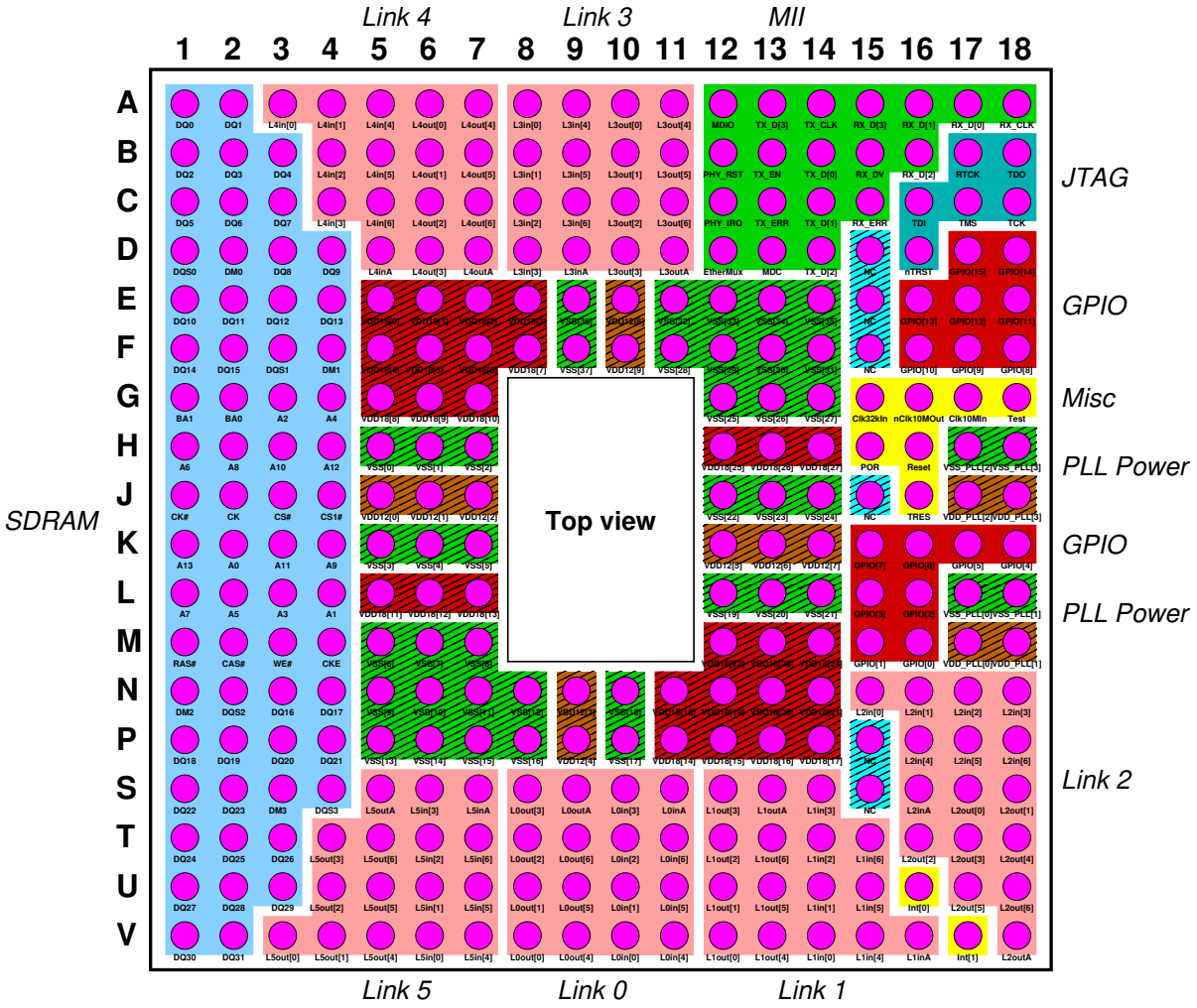


Figure 3.2.1: SpiNNaker 300LBGA Packaging

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.

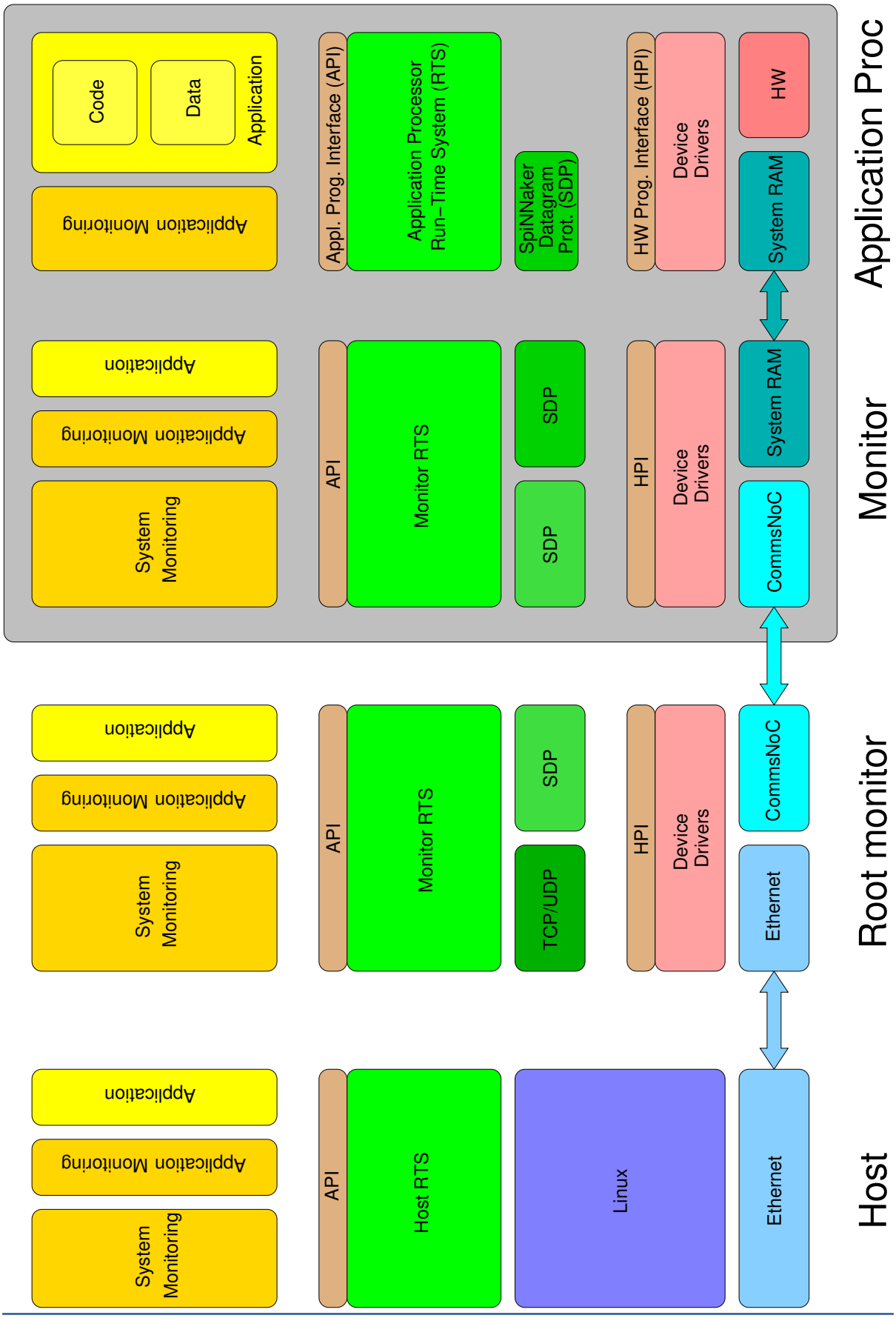


Figure 3.3.1: SpiNNaker run-time software stack.

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.

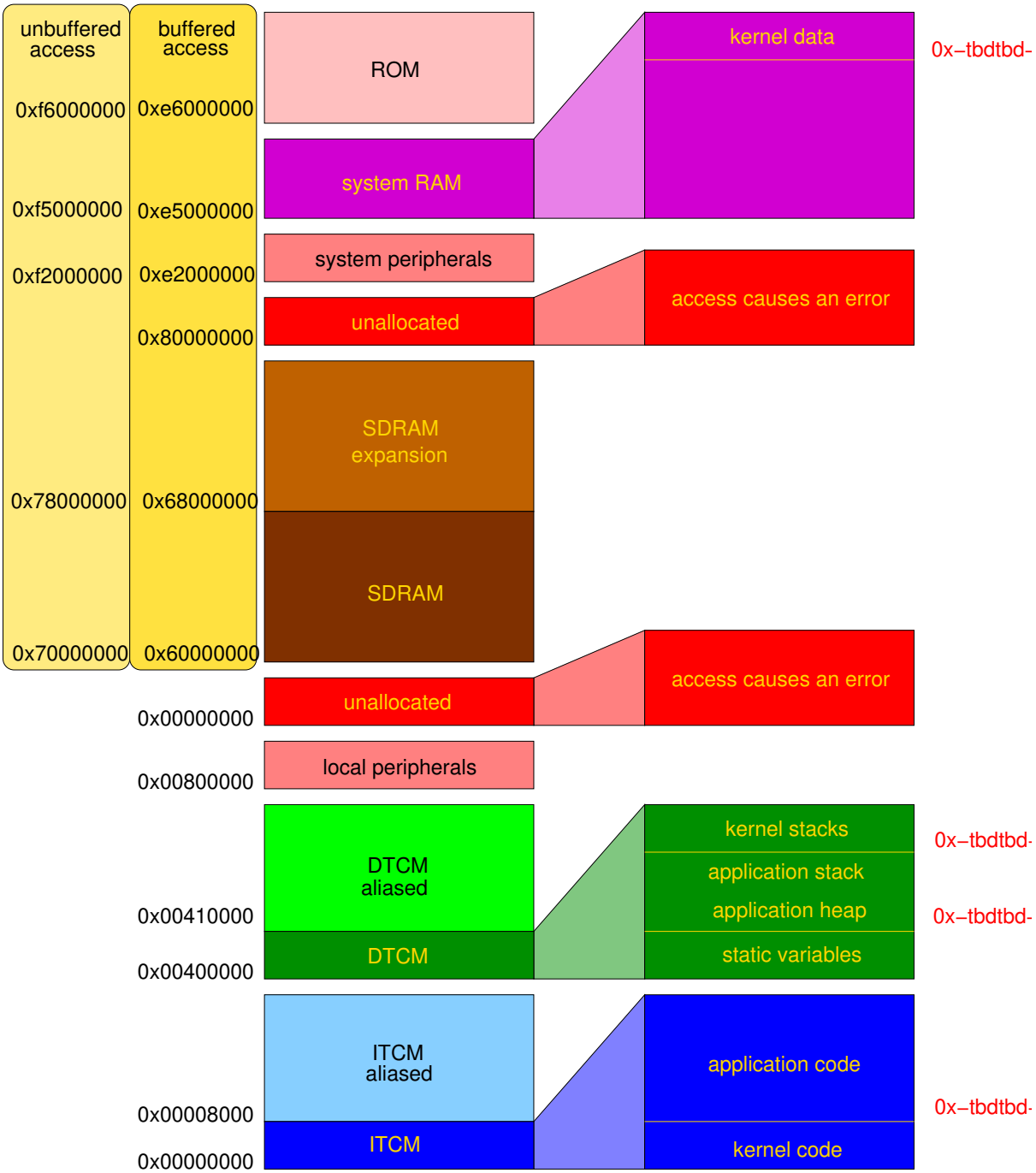


Figure 3.3.2: SpiNNaker run-time memory map.

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.

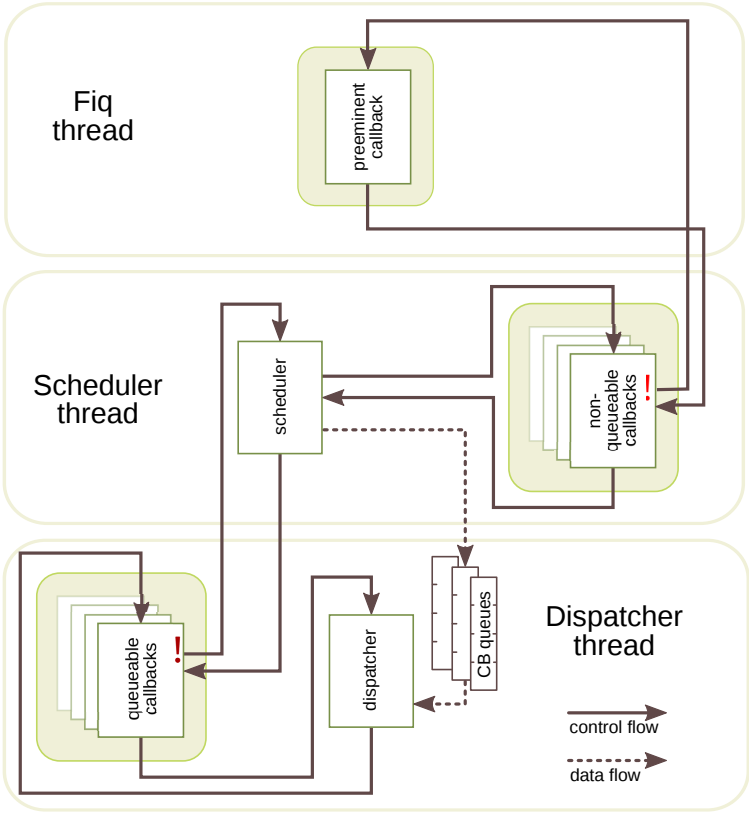


Figure 3.3.3: SpiNNaker event-driven programming framework.

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,        O,        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,
  O,        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,        O},
  {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},
  {O,        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.

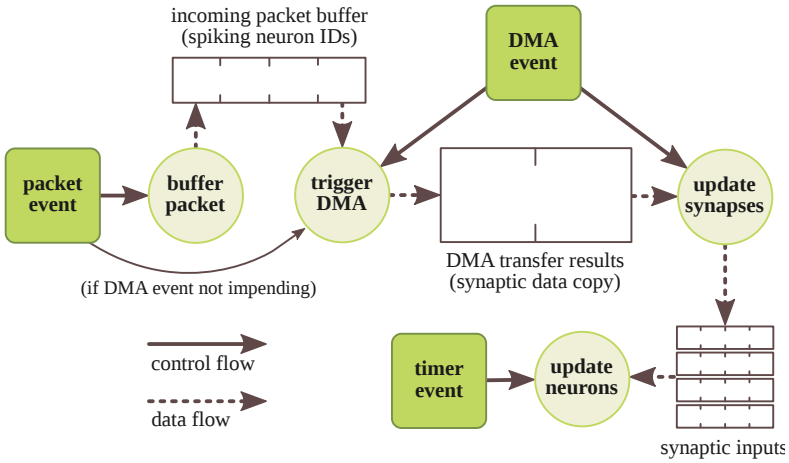


Figure 3.3.4: Events and corresponding tasks in a typical neural simulation.

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.

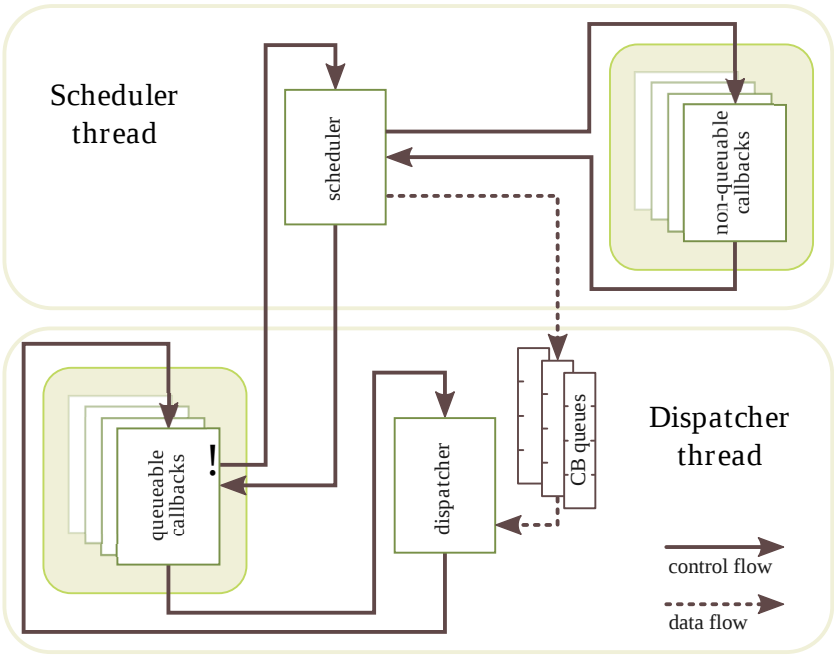


Figure 3.3.5: Control and data flow between the scheduler and dispatcher threads.

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:

$$\Delta w_{ij} = \eta \delta_j O_i \tag{3.3.1}$$

$$\delta_j = \begin{cases} (C_j - O_j)\frac{dT_j}{dS_j} & \text{if j is an output layer} \\ \frac{dT_j}{dS_j}\sum_k \delta_j w_{jk} & \text{if j is not an output layer} \end{cases} \tag{3.3.2}$$

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: $M_{I_x J_y} = m_{ij} \mid_{i_{nx}:i_{n(x+1)};j_{ny}:j_{n(y+1)}}$. 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:

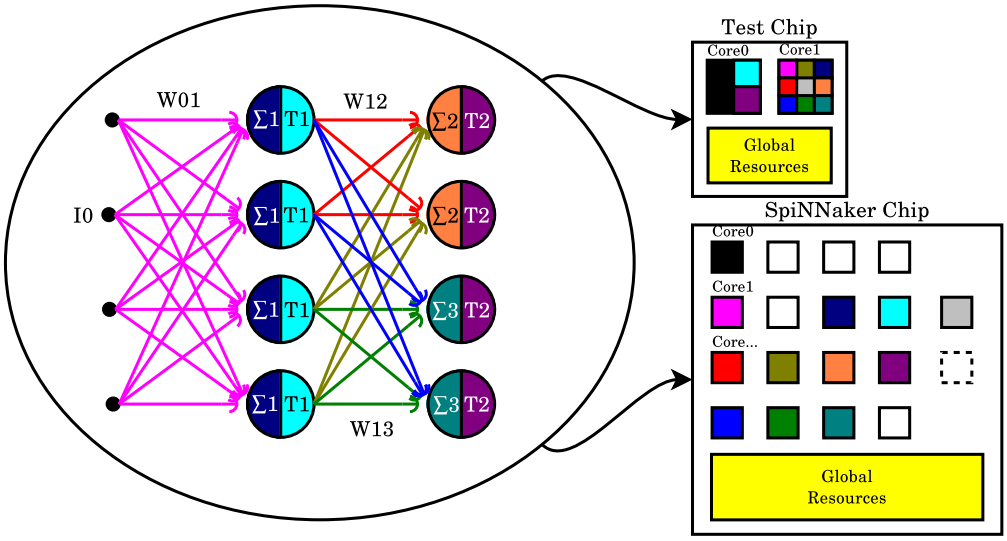


Figure 3.3.6: MLP network mapping.

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

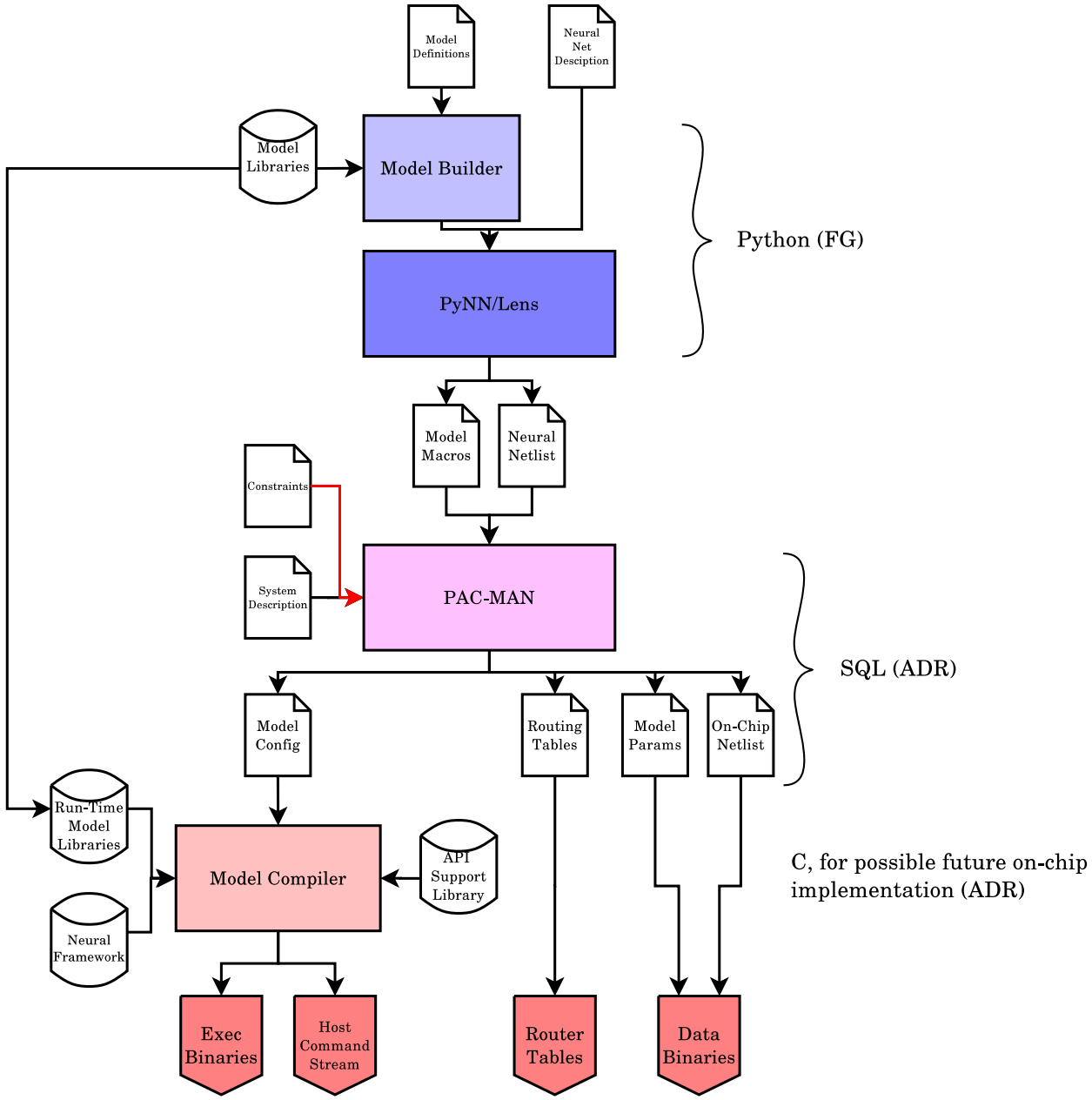


Figure 3.3.7: SpiNNaker neural net simulation development route.

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]’

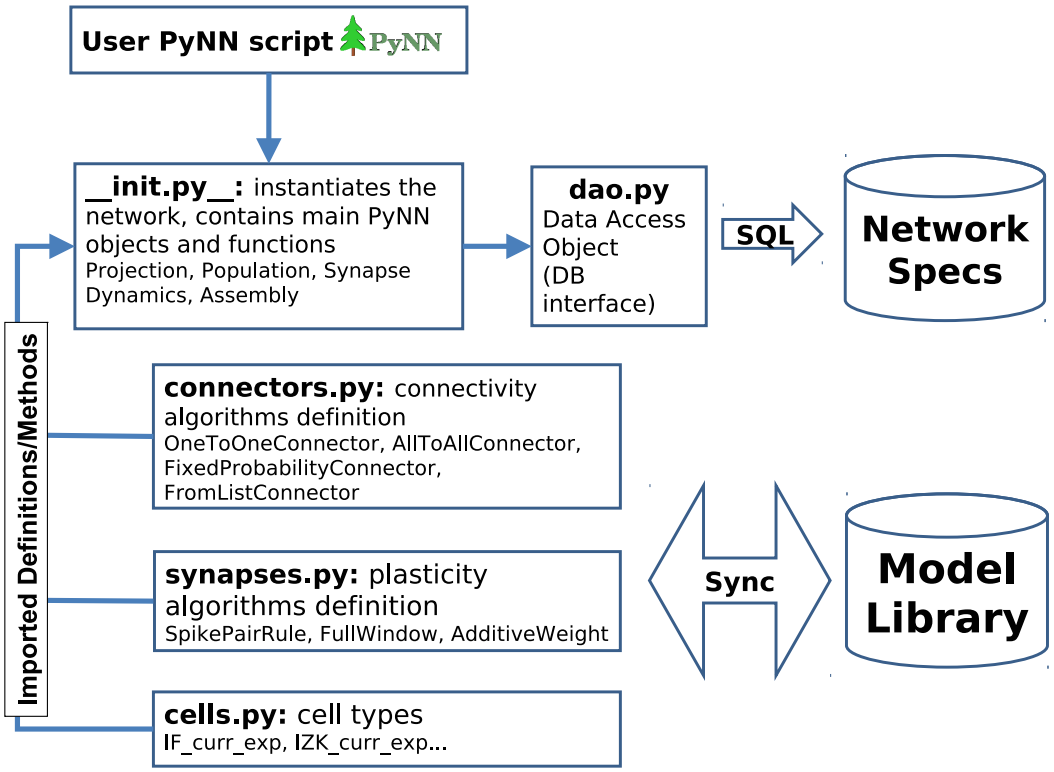


Figure 3.3.8: PyNN/SpiNNaker interface structure.

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.)

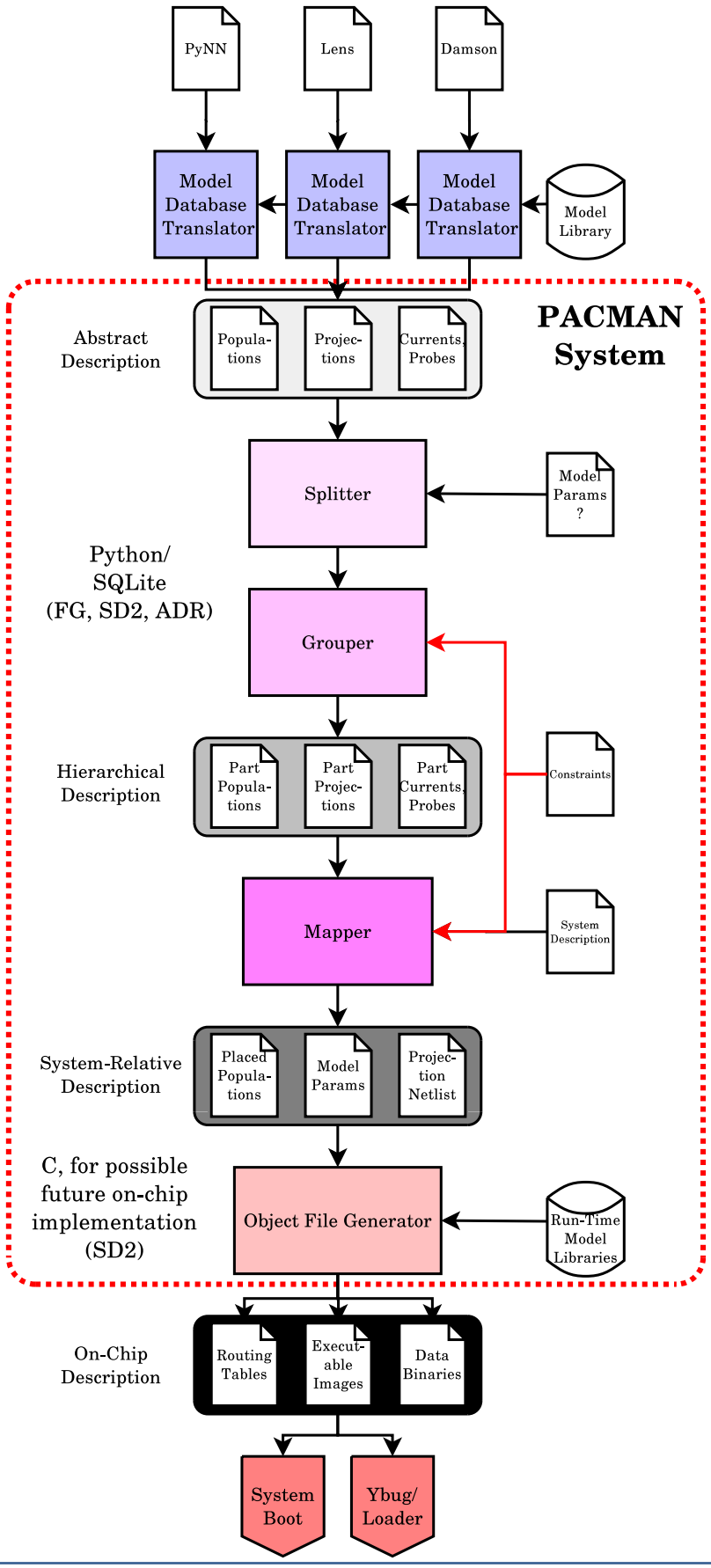


Figure 3.3.9: PACMAN internal structure

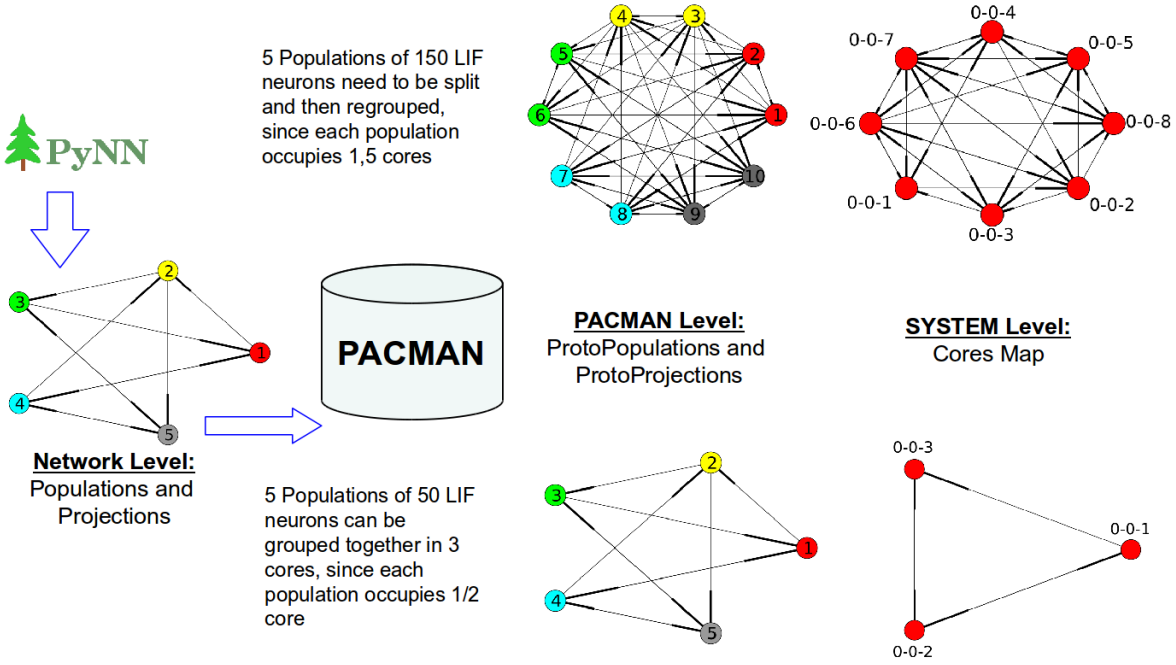


Figure 3.3.10: Example of network representation in PACMAN

- **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:

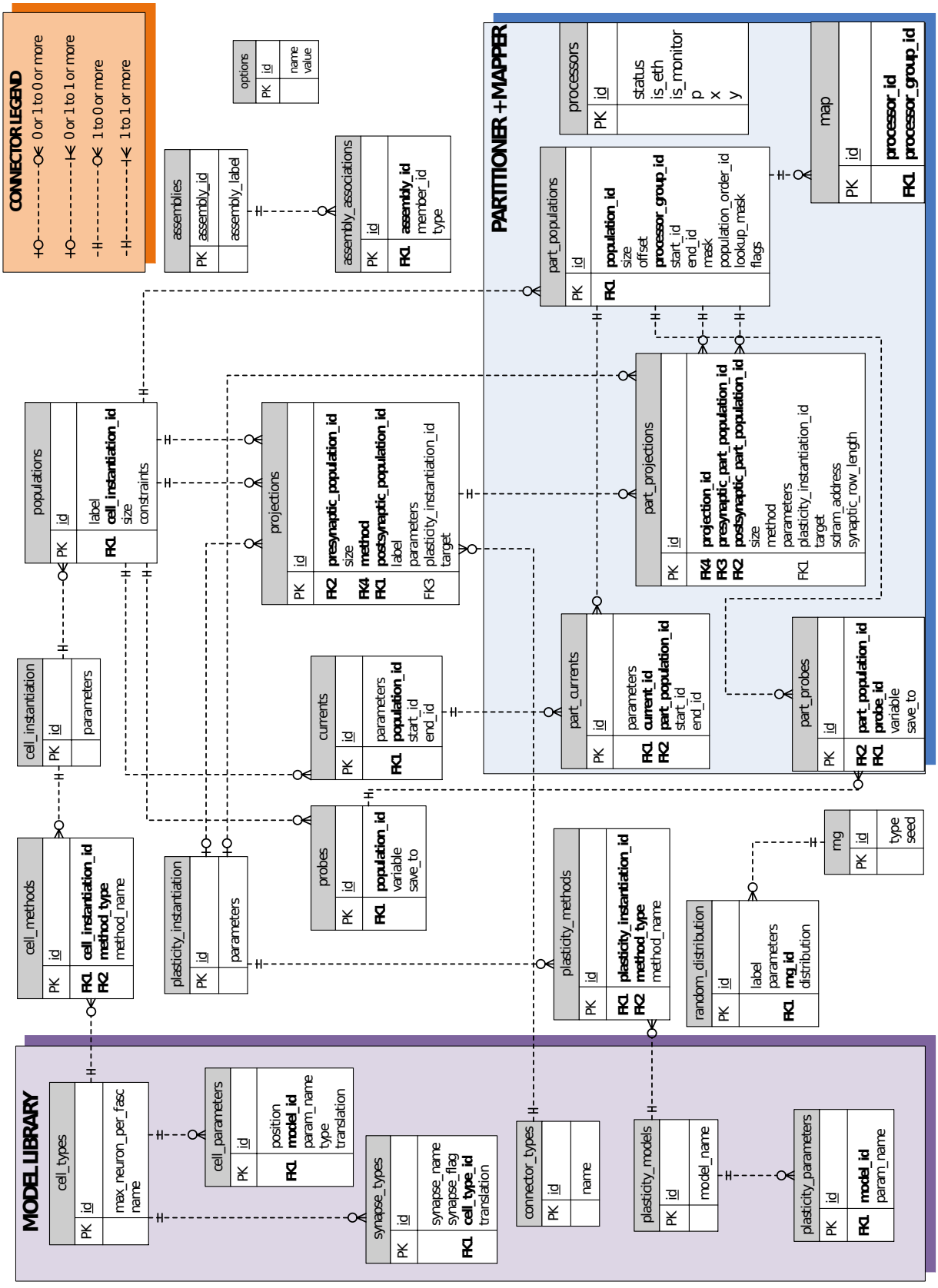


Figure 3.3.11: Database structure: model library and network specification

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 Offset
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.

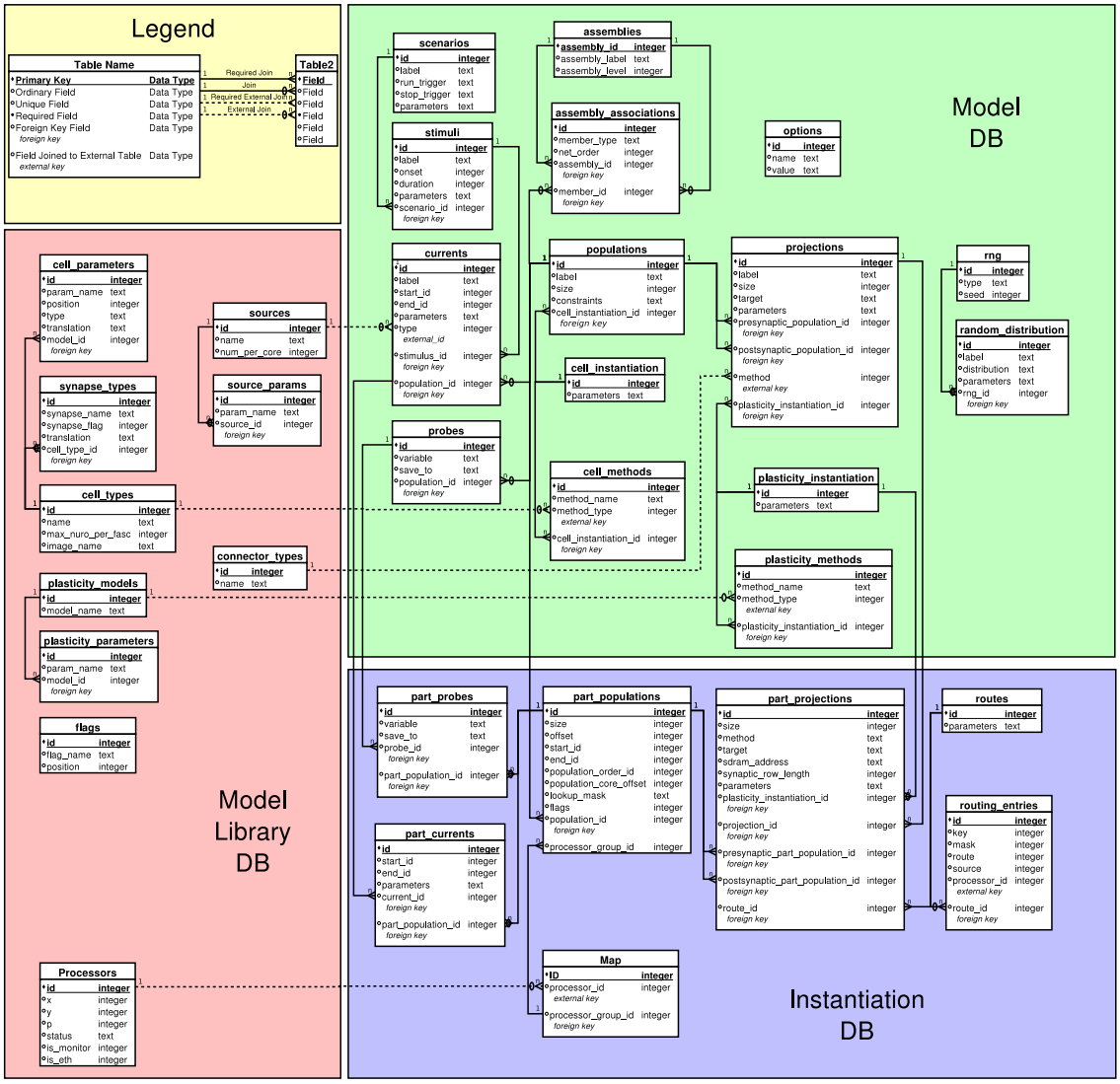


Figure 3.3.12: Updated PACMAN database schema for Lens support

3) pacman_objs: A new, object-oriented interface to PACMAN. This permits object-style interface to the database.

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)

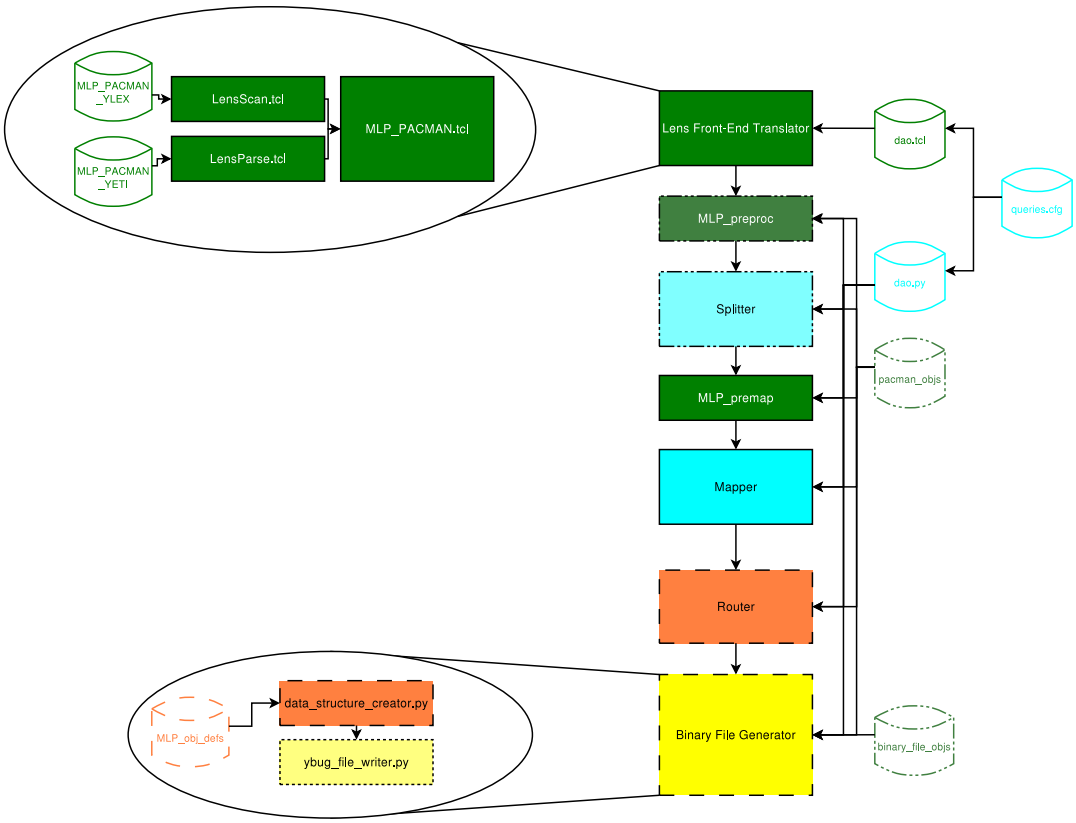


Figure 3.3.13: PACMAN Architecture including MLP extensions

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 associated 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:

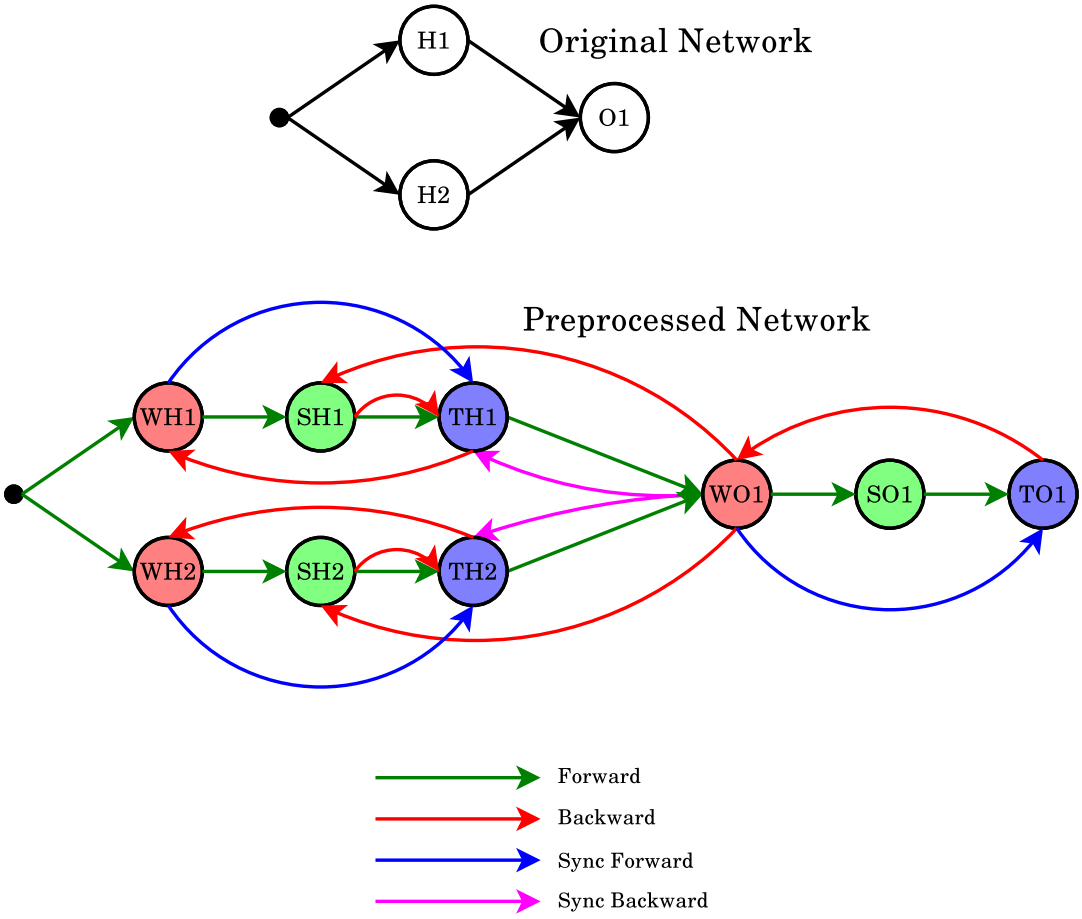


Figure 3.3.14: Mapping of the MLP preprocessor

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. Each benchmark has a set of quality measures. It is left to the user to decide which 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 criterion for high performance computers. This well known 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:

- Energy usage for a fundamental operation;
- Computational resource usage (neurons, synapses, transistors);
- Silicon area or volume;
- Execution time for a specific task
- Number of events/spikes processed per second
- Time to configure/upload a network
- Precision of the solution compared to a numerical solution obtained on traditional computing hardware
- Trial-to-trial reproducibility of the result
- Robustness against hardware mismatch

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 simulations, 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 seen 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 straightforward. 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 transmission 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.

4.6 Implementation

4.6.1 Defining models and tasks

As stated above, all benchmark code should be under version control. Each repository may contain one or more models, and for each model one or more tasks should be defined. The top-level of the repository should contain a JSON-format configuration file named "benchmarks.json", with the following general structure:

```
[
  {
    "model": "Description of model A",
    "tasks": [
      "task_1_for_model_A.py {system}",
      "task_2_for_model_A.py arg1 {system}",
    ]
  },
  {
    "model": "Description of model B",
    "tasks": [
      "task_1_for_model_B.py {system}",
      "task_2_for_model_B.py arg1 {system}",
      "task_3_for_model_B.py --option1={system} arg1 arg2 arg3",
      "task_3_for_model_B.py --option1={system} arg4 arg5 arg6",
    ]
  }
]
```

i.e., each task should be expressed as a command-line invocation of a Python script. The Python script should in general use the PyNN API, in which case the placeholder "system" must be provided, and will be replaced by the name of the PyNN backend used when running the benchmark, e.g. "nest", "spiNNaker", or "hardware.hbp_pm". If the benchmark is known to run only on a subset of the available backends, this can be indicated by listing the suitable backends within the placeholder, e.g. "system=spiNNaker,nest". For low-level benchmarks for a single neuromorphic system, the Python script should use the low-level APIs of that platform, and in this case the "system" placeholder should be absent.

A specific example, for the repository `https://github.com/CNRS-UNIC/hardware-benchmarks`, is:

```
[
  {
    "model": "A population of IF neurons, each of which is injected with a different current
    "tasks": [
      "run_IF_curve.py {system}"
    ]
  },
  {
    "model": "A population of random spike sources, each with different firing rates",
    "tasks": [
      "run_spike_train_statistics.py {system}"
    ]
  }
]
```

4.6.2 Returning numerical measures

Each task should run a simulation of a neuronal network model, record data from the neurons, perform analysis of the data, and calculate numerical measures of the system performance. The numerical measures should be reported in a JSON-format file, consisting of a top-level record with required fields "timestamp" and "results". The field "configuration", containing a copy of the parameterization of the model and simulator/hardware system", is optional. The field "results" contains a list of records with the following fields:

type What is being measured. For example "quality", "performance", "energy consumption".

name A unique name for the measurement. It is suggested that this name takes the form of a URI containing the URL of the version control repository followed by an identifier for the task and an identifier for the measurement.

value A floating point number.

units (optional) if the measurement is a physical quantity, the units of the quantity using SI nomenclature.

measure the type of the measurement, for example "norm", "p-value", "time".

(A controlled vocabulary will be developed for the fields "type" and "measure").
Here is an example:

```
{
    "timestamp": "2015-06-05T11:13:59.535885",
    "results": [
```

```
        {
            "type": "quality",
            "name": "https://github.com/CNRS-UNIC/hardware-benchmarks.git/I_f_curve#norm_di
            "value": 0.0073371188622418891,
            "measure": "norm"
        },
        {
            "type": "performance",
            "name": "https://github.com/CNRS-UNIC/hardware-benchmarks.git/I_f_curve#setup_t
            "value": 0.026206016540527344,
            "units": "s",
            "measure": "time"
        },
        {
            "type": "performance",
            "name": "https://github.com/CNRS-UNIC/hardware-benchmarks.git/I_f_curve#run_time
            "value": 1.419724941253662,
            "units": "s",
            "measure": "time"
        },
        {
            "type": "performance",
            "name": "https://github.com/CNRS-UNIC/hardware-benchmarks.git/I_f_curve#closing_
            "value": 0.03272294998168945,
            "units": "s",
            "measure": "time"
        }
    ],
}
```


The task may also optionally produce figures and other output data files.

4.6.3 Registering benchmarks

To add a new benchmark model or task within an existing repository, just modify the "benchmarks.json" configuration file. To add a new repository, e-mail andrew.davison@unic.cnrs-gif.fr. In future, a web form for registering new repositories will be introduced.

4.6.4 Running benchmarks

A continuous integration system will be put in place, which will run the entire suite of benchmarks on each neuromorphic system every time the system configuration (software or hardware) is changed, and which will run the benchmarks from a given repository on both neuromorphic systems (where appropriate) each time a new commit is made to the repository. To indicate

that a given commit should not trigger a run (for example because only documentation has been changed), include the text "[skip ci]" or "[ci skip]" within the commit message.

After running each task, the continous integration system will harvest the JSON-formatted measurement report, and update a database of benchmark measurements. This benchmark database will be visualized in an "App" within the HBP Neuromorphic Platform Collaboratory.

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 United Microelectronics Corporation (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 41). 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] Intel NUC. `http://www.intel.com/content/www/us/en/nuc/overview.html`, 2014.

[2] Raspberry Pi. `http://www.raspberrypi.org`, 2014.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] Sebastian Millner. Development of a Multi-Compartment Neuron Model Emulation. PhD thesis, University of Heidelberg, 2012.

[9] The Human Brain Project. A Report to the European Commission, 2012.

[10] United Microelectronics Corporation (UMC). `http://www.umc.com`.

Glossary

10GbE

10-Gigabit Ethernet. 51

40GbE

40-Gigabit Ethernet. 51

ADC

Analog-to-Digital Converter. 53

AnaB

Analog Breakout PCB. 52

AnaFP

Analog Frontend PCB. 53, 285

AnaRM

Analog Readout Module. 50, 51, 53, 285

API

Application Programming Interface. 55–57

ASIC

Application Specific Integrated Circuit. 49, 52

AuxPwr

Auxiliary Power Supply PCB. 52, 284

Calibration

Calibration. 56

CMOS

Complementary Metal-Oxide-Semiconductor. 49

Compute Cluster

A collection of computers interconnected by a dedicated network. 51, 53

Compute Node

A single compute node as part of a Cluster. 52, 57

Cure

Monitoring and Control PCB for Reticles. 52, 284

DRAM

Dynamic Random Access Memory. 53

ESS

Executable System Specification. 55, 58, 59

FCP

FPGA Communication PCB. 50, 52, 284

Flyspi

Flyspi FPGA PCB. 53, 284

FPGA

Field-Programmable Gate Array. 52, 53

FsBo

Flyspi Breakout PCB. 53

GbE

Gigabit Ethernet. 51, 52

HALbe

Hardware Abstraction Layer Backend. 57

HDD

Hard disk drive. 52

HICANN

High-Input Count Analog Neuronal Network Chip. 49, 52

HICANN Wafer

A 20 cm silicon wafer with 384 HICANN ASICs interconnected by wafer-scale postprocessing. 50, 52

I/O

Input/Output. 51

I2C

Inter-Integrated Circuit Link. 52

MainPCB

Wafer Module Main PCB. 52, 283, 284

Mapping

Mapping. 56

MCU

Microcontroller Unit. 52

NM-PM

Neuromorphic Physical Model. 4, 49, 51, 55–58

NM-PM1

Neuromorphic Physical Model version 1. 4, 13, 50, 51, 283, 285, 296

NUC

Next Unit of Computing. 53

PCB

Printed Circuit Board. 53, 283, 284

PCIe

Peripheral Component Interconnect Express. 52

Power-FET

Power Field-Effect Transistor. 52

PowerIt

PowerIt Main Power Supply PCB. 52, 284

PyNN

PyNN. 55–57, 59

Python

Python Programming Language. 55

QSFP

Quad SFP. 51

RDMA

Remote Direct Memory Access. 50

SFP+

Small Form-Factor Pluggable. 51

SSD

Solid-state Disk. 52

StHAL

Stateful Hardware Abstraction Layer. 57

ToR

Top-of-Rack. 50–52

UMC

NM-PM1 semiconductor manufacturer: United Microelectronics Corporation UMC [10]. 283

USB 2.0

Universal Serial Bus version 2.0. 53

Wafer

silicon wafer used as the basis of micro-chip production. 283

Wafer Module

Assembly of an HICANN wafer, a Main PCB, 48 FPGA communication PCBs and power supply PCBs. 49–53, 283, 285

WIO

Wafer I/O PCB. 52

WIOH

Horizontal Wafer I/O PCB. 52

WIOV

Vertical Wafer I/O PCB. 52