MANCHESTER
1824

The University
of Manchester

SpiNNaker

Universal Spiking Neural Network Architecture

Software Specification and Design

version 0.0

25 October 2012

About this Document

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

All rights to the SpiNNaker design and its associated software are the property of the University of Manchester with the exception of those rights that accrue to the project partners in accordance with the contract terms.

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>

Change history

version	date	changes
0.0	03/05/2011	Initial draft
0.01	11/11/2012	Added description of the synaptic plasticity features

Contents

About this Document **2**
- Background . . . 2
- Intellectual Property rights . . . 2
- Disclaimer . . . 2
- Error notification and feedback . . . 2
- Change history . . . 2

1 Introduction **6**
- 1.1 Execution phases . . . 6

2 Boot and application loading sequence **8**
- 2.1 Node-Boot . . . 8
  - 2.1.1 Introduction to Node-Boot . . . 9
  - 2.1.2 First steps of Node-Boot . . . 9
  - 2.1.3 Watchdog caused Boot . . . 9
  - 2.1.4 Usage of the SerialROM . . . 9
  - 2.1.5 Tightly Coupled Memories . . . 11
  - 2.1.6 POST Self-Test for Processor Blocks . . . 11
  - 2.1.7 Monitor Processor Arbitration . . . 12
  - 2.1.8 Chip-Level POST . . . 14
  - 2.1.9 Router Initialisation . . . 14
  - 2.1.10 Ethernet Initialisation . . . 15
  - 2.1.11 Misc Chip Level Testing and Initialisation . . . 16
  - 2.1.12 Timer and VIC Initialisation . . . 16
  - 2.1.13 Final Initialisation Steps . . . 17
  - 2.1.14 The Main Loop . . . 17
- 2.2 System-Boot . . . 18
  - 2.2.1 State of SpiNNaker Node after Node-Boot . . . 18
  - 2.2.2 State of Each Processor Block Level . . . 18
  - 2.2.3 State at SpiNNaker Chip (Node) Level . . . 19
  - 2.2.4 Ethernet / Host Boot . . . 21
  - 2.2.5 Notes on the Host System . . . 23
  - 2.2.6 Inter-Chip Boot . . . 23
  - 2.2.7 System-Boot Loading - A state machine . . . 27
  - 2.2.8 Immutable Data Copier . . . 27
  - 2.2.9 Block size choices and Retransmissions . . . 28
  - 2.2.10 Pertinent notes on this process / creating a System-Boot image . . . 28
  - 2.2.11 Application Processor Loading . . . 28
  - 2.2.12 Homogeneous Image Loading To All Processors . . . 29
  - 2.2.13 IVB - ITCM Validation Block - a technique for graceful recovery of Watchdog resets 30

3 Run-time software **32**
- 3.1 Run-time software stack . . . 32
- 3.2 Inter-processor communication . . . 32
  - 3.2.1 Processor Virtualisation . . . 32
  - 3.2.2 Addressing SpiNNaker Nodes . . . 34
  - 3.2.3 SpiNNaker Datagram Protocol (SDP) . . . 34
  - 3.2.4 Software support for communication . . . 34
- 3.3 Runtime memory map . . . 35

4 Application programming interface (API) 36

4.1 Event-driven programming model . . . 36

4.1.1 Design considerations . . . 37

4.2 Programming interface . . . 37

4.2.1 Events . . . 37

4.2.2 Callback arguments . . . 38

4.2.3 Pre-defined constants . . . 38

4.2.4 Pre-defined types . . . 39

4.2.5 Kernel services . . . 40

4.2.6 Application Programme Structure . . . 50

5 Neural net simulation frameworks 52

5.1 Spiking Neural net simulation framework . . . 52

5.2 MLP simulation framework . . . 55

6 Neural net simulation development route 57

6.1 pyNN.spiNNaker . . . 57

6.2 PyNN API functions list . . . 61

6.3 Simulation setup and control . . . 61

6.4 Object-oriented interface for creating and recording networks . . . 61

6.5 PopulationView . . . 62

6.6 Assembly . . . 62

6.7 Object-oriented interface for connecting populations of neurons . . . 62

6.8 Procedural interface for creating, connecting and recording networks . . . 63

6.9 Neural Models . . . 63

6.10 Specification of synaptic plasticity . . . 64

6.11 Current Injection . . . 64

7 Damson development route 69

7.1 Damson program compilation . . . 69

7.2 Damson code components . . . 69

7.3 Mapping code to SpiNNaker processors . . . 69

7.4 Runtime system . . . 69

7.5 Damson development flow . . . 69

8 PACMAN: partition and configuration manager 71

8.1 Introduction . . . 71

8.2 Splitting . . . 73

8.2.1 Implementation . . . 76

8.3 Grouping . . . 76

8.3.1 Implementation . . . 77

8.4 Mapper . . . 77

8.4.1 Implementation . . . 79

8.5 Object File Generator . . . 79

8.6 Neural Data Structure generation . . . 81

8.7 Automatic Run Script generation . . . 82

9 Coding guidelines 84

9.1 All languages . . . 84

9.2 C . . . 84

9.3 ARM assembly . . . 85

9.4 Python . . . 85

10 Documentation guidelines **86**
10.1 C / C++ . . . 86
10.1.1 File header documentation template . . . 86
10.1.2 Function documentation template . . . 86
10.1.3 Structure documentation template . . . 87
10.2 Assembly language . . . 87
10.2.1 File header documentation template . . . 87
10.2.2 Function documentation template . . . 88
10.3 Robodoc configuration file . . . 88

1 Introduction

1.1 Execution phases

The task of running a model on SpiNNaker can be broken down into a series of execution phases, as illustrated in figure 1. The SpiNNaker hardware is linked to a host machine, which is usually a linux desktop or laptop. The physical link between the host machine and SpiNNaker is via:

- a 100Mbit/s Ethernet cable, which is used to load code and data into SpiNNaker and to receive outputs and monitor information from SpiNNaker, possibly plus
- a USB cable, which can be used to support a remote reset of the SpiNNaker system and (for small SpiNNaker systems) may be the source of power for SpiNNaker.

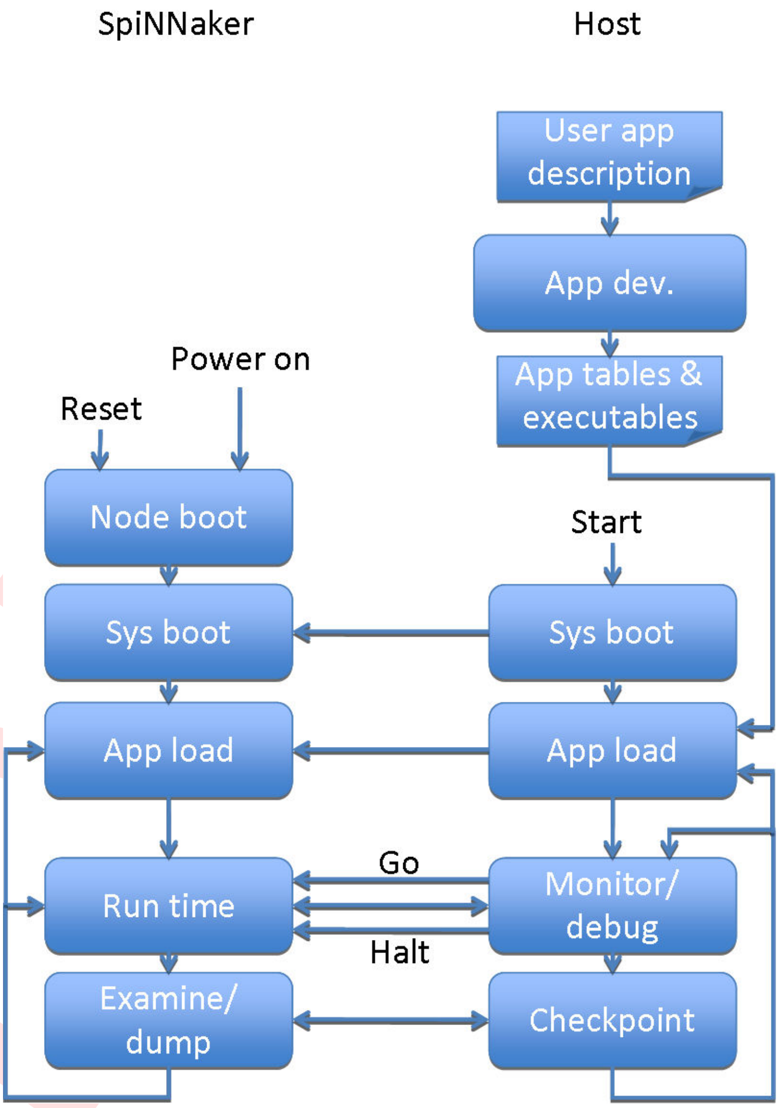

Figure 1: SpiNNaker Execution Phases.

The top-level view of the SpiNNaker execution phases and associated software is then as follows:

- Application development phase:
  - This is done in the host machine, and does not require a connected SpiNNaker machine.
  - High level application development is described in Section 7 (for Damson) and Section 6 for PyNN and Lens.
  - Low level application development is covered by the API for the C language (in Section 4), and the simulation frameworks for spiking neural networks and for multi-layer perceptrons (in Section 5).
- Boot and application loading phase:
  - This phase prepares SpiNNaker to run a model, and requires SpiNNaker to be connected to a host. Both machines are involved. This phase is described in more detail in Section 2.
  - At power on or reset each SpiNNaker node carries out an internal initialisation process.
  - Then the host loads system code into SpiNNaker.
  - Finally, application code and data are loaded into SpiNNaker.
- Run-time phase:
  - Again, SpiNNaker and the host machine must be connected and are both involved throughout this phase.
  - First, the host sends a ”Go” signal to SpiNNaker, and real-time execution commences.
  - During execution the host can convey external inputs and outputs to and from SpiNNaker, and it can monitor internal activity and provide run-time debug facilities.
  - When required, the host can issue a ”Halt” signal to SpiNNaker, causing the real-time execution to cease.
  - While SpiNNaker is halted the host can inspect internal SpiNNaker state and, potentially, check-point the simulation.

During the run-time phase the kernel provides support for application execution using an event-driven programming model (Section 4), support for communication (I/O) (using the SDP protocol, see Section 3) and support for external hardware devices.

2 Boot and application loading sequence

SpiNNaker boots in three phases (figure 2):

1. *Node-Boot* at power-on executes code retrieved from the read-only *BootROM* (Section 2.1). This code does the initial chip testing and initialisation, electing a Monitor Processor, leaving the node ready to receive the externally originated 2nd phase image...

2. The *System-Boot* image (see Section 2.2) is received by one or more Ethernet attached SpiNNaker nodes from the host, and self-propagates itself to its immediate neighbours, and so on, until all Monitor Processors in the system are running the same homogeneous image. This propagation process is known as *flood-fill*. Following a System-Boot the machine is then numbered, permitting the 3rd stage of boot...

3. *Application-Load* (Section 2.2.11) is where the Operational software for both Monitor and Application Processors is loaded to the appropriate cores, route tables populated, and application supporting data uploaded to occupy the shared SDRAM of each chip.

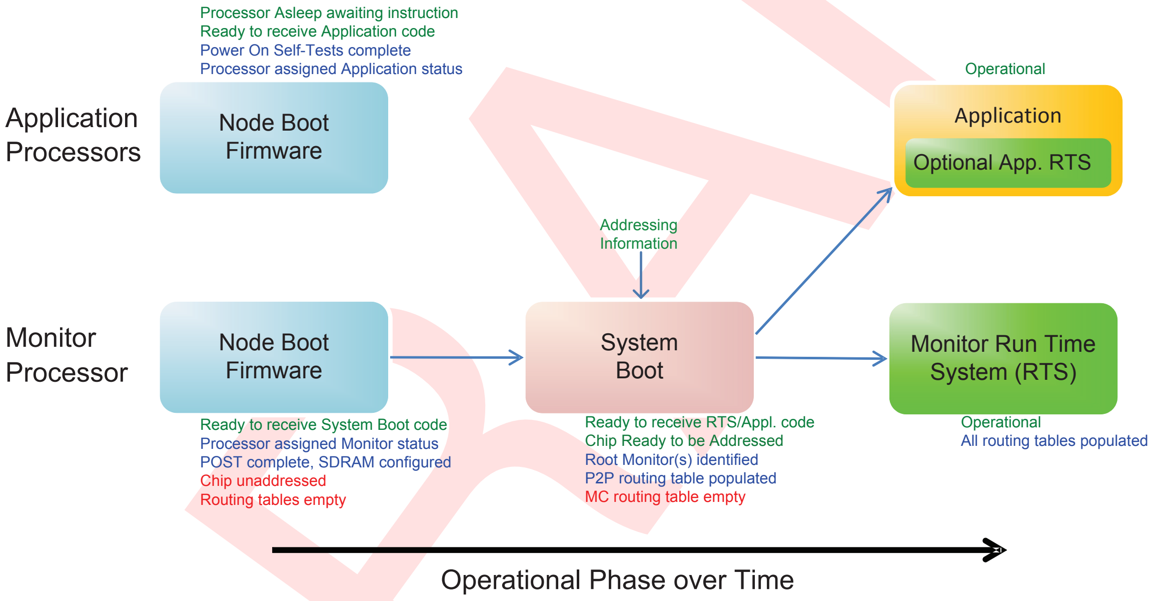
Processor Asleep awaiting instruction
Ready to receive Application code
Power On Self-Tests complete
Processor assigned Application status
Application Processors
Node Boot Firmware
Operational
Application
Optional App. RTS
Addressing Information
Monitor Processor
Node Boot Firmware
System Boot
Monitor Run Time System (RTS)
Ready to receive System Boot code
Processor assigned Monitor status
POST complete, SDRAM configured
Chip unaddressed
Routing tables empty
Ready to receive RTS/Appl. code
Chip Ready to be Addressed
Root Monitor(s) identified
P2P routing table populated
MC routing table empty
Operational
All routing tables populated
Operational Phase over Time

Figure 2: SpiNNaker boot and application loading sequence.

2.1 Node-Boot

Node-Boot is the first stage of the SpiNNaker boot-up process (Figure 2) and is initiated by the BootROM contents. This code takes the chip through the Power On Self Tests (POST) (see Table 1, hardware initialization and, assuming its functionality, continues on to the 'listening' state for the node. From this operating mode the node may transition to System-Boot, which is the next phase of processor operation. The boot process is designed to be as resilient as is practicable as it is possible that faults develop on a chip during its lifetime. Gracefully handling such situations means that it may not be an immediate requirement to change a circuit board (which may contain dozens of functional chips), or cause downtime for the machine as a whole.

The following sections outline the process of Node-Boot in greater detail in order that the reader can interact with the machine at this stage of boot.

2.1.1 Introduction to Node-Boot

In this section the boot sequence is chronologically detailed, including the initialised state in which the node hardware remains awaiting transition to System-Boot. If further specifics are required then the reader is directed to the gold BootROM source code which is available for inspection in the usual SVN SpiNNaker software repositories.

It is simplest to visualise the Node-Boot process by means of two flow-charts (Figures 3 and 4), which are augmented by description within this text. Please refer to the flow-charts as necessary while reviewing this section.

The SpiNNaker chip design ensures that each processor initialises in High Vectors, therefore at power-on operating instructions come from the memory map at 0xFFFF0000, which is aliased to the BootROM. All 18 cores in each SpiNNaker node populate their instruction pipelines from this location and the BootROM code from here branches unconditionally into the initialisation routines of the SpiNNaker chip.

2.1.2 First steps of Node-Boot

The first test in the boot sequence checks the hardware status of external GPIO pin 7 (attached to the chip package) to determine whether a manufacturing test should be run (for die testing), or whether the default BootROM should operate. Assuming the default latter behaviour, each core checks the chip-level 'Reset Code' located at Register 12 of the System Controller which influences the booting behaviour as described in this text and flow-charts. Usually the BootROM is encountered at power-on, but it may also be executed again if a reset is triggered remotely for the chip or core, or if a watchdog enforced reset has been initiated due to errant run-time behaviour.

2.1.3 Watchdog caused Boot

If the cause of boot is indeed errant behaviour triggering the watchdog, application authors may optionally provide a user-controlled software restart mechanism that avoids passing back into the BootROM. This optional facility is called the IVB (ITCM Validation Block) which detects whether the instruction memory is intact and has not been corrupted by the software malfunction, before branching to restart the core at a user-defined ITCM function. This optional facility is detailed for potential users in Section 2.2.13. If the facility is not used, or encounters a problem the usual reset procedures apply.

2.1.4 Usage of the SerialROM

With non-watchdog reset codes, including power on or remote reset, then the usual Node-Boot routines are followed in order to initialise and check the hardware. One of the cores is selected as Boot Processor by setting external GPIO pins, and this Boot Processor checks whether the presence of an external SerialROM chip is indicated (again via a GPIO pin signal). The SerialROM chip is used primarily to provide and populate Ethernet attached chips with MAC and IP addressing information (Table 2), but may also optionally be used to exit this Node-Boot sequence early, to load and execute its own self-contained image.

At this stage all other non Boot Processors wait on Bit1 of the System Controller's R14 (Misc Control) register to be set by the Boot Processor (on completion of the SerialROM routine), before rejoining the main flow. Note the bits of R14 used by the Node-Boot Routine are not documented in the hardware data sheet as they are only transiently required.

All processors now set the core clocks to 160MHz, to accelerate the boot process, after running at an initial 10MHz (see Table 3). Note all functional processors do this simultaneously, without glitches being introduced.

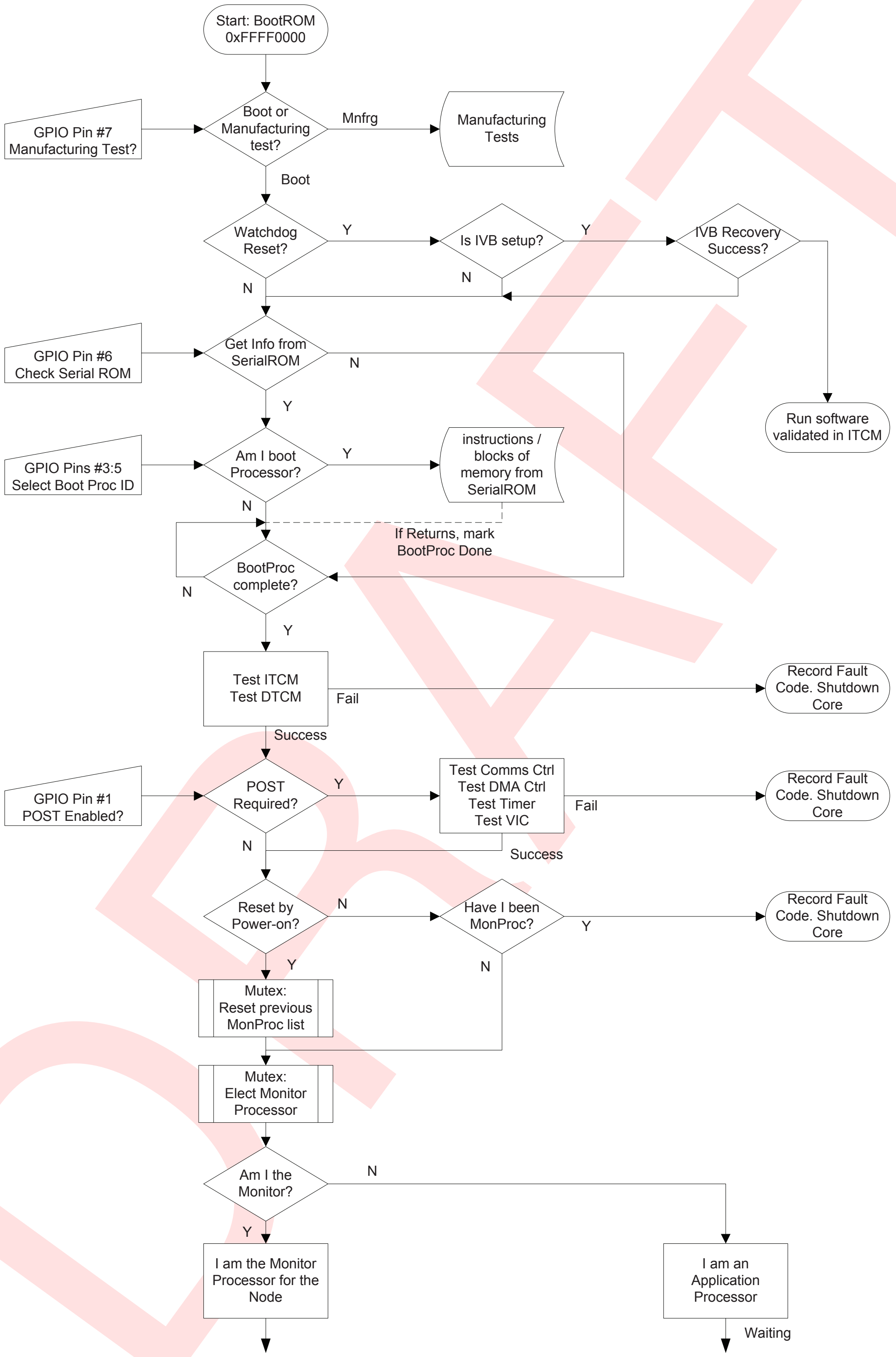
Start: BootROM 0xFFFF0000
GPIO Pin #7 Manufacturing Test?
Boot or Manufacturing test?
Mnfrg
Manufacturing Tests
Boot
Watchdog Reset?
Y
Is IVB setup?
Y
IVB Recovery Success?
N
N
GPIO Pin #6 Check Serial ROM
Get Info from SerialROM
N
Run software validated in ITCM
Y
GPIO Pins #3:5 Select Boot Proc ID
Am I boot Processor?
Y
instructions / blocks of memory from SerialROM
N
If Returns, mark BootProc Done
BootProc complete?
N
Y
Test ITCM Test DTCM
Fail
Record Fault Code. Shutdown Core
Success
GPIO Pin #1 POST Enabled?
POST Required?
Y
Test Comms Ctrl Test DMA Ctrl Test Timer Test VIC
Fail
Record Fault Code. Shutdown Core
N
Success
Reset by Power-on?
N
Have I been MonProc?
Y
Record Fault Code. Shutdown Core
Y
N
Mutex: Reset previous MonProc list
Mutex: Elect Monitor Processor
Am I the Monitor?
N
Y
I am the Monitor Processor for the Node
I am an Application Processor
Waiting

Figure 3: Flow of all cores during Node-Boot, from reset until Monitor Processor allocation. This flowchart continues in Figure 4.

Peripheral	Method	Executor	Failure response	Error Code
		At Power-on		
ITCM	RAM test	All	Shutdown Core	0x4
DTCM	RAM test	All	Shutdown Core	0x2
		After Scatter loading		
Comms controller	Register test	All	Shutdown Core	0x0
DMA controller	Register test	All	Shutdown Core	0x1
Timer	Register test	All	Shutdown Core	0x9
VIC	Register test	All	Shutdown Core	0xA
		After Monitor election		
Previous Monitor	Check Register	All	Shutdown Core	0xC
SystemRAM	RAM test	Monitor	Shutdown MP Core	0x7
Router	Register test	Monitor	Shutdown MP Core	0x6
Watchdog	Register test	Monitor	Shutdown MP Core	0xB
PL340	Register test	Monitor	Record, continue	(see SDRAM
SDRAM	RAM test	Monitor	Record, continue	(registers
		Exceptions		
Exception	Hi Vectors	All	Shutdown Core	0xD
Exception	Low Vectors	All	Shutdown Core	0xE

Table 1: Ordered list of power-on self-tests performed during Node-Boot. An error code is written to the processor's failure log in the indicated bit position if detected.

2.1.5 Tightly Coupled Memories

All cores initialise their TCM memories (both data and instruction), although this is 'belt-and-braces' as chip hardware already forces the TCM memories to be enabled at power-up. The per-processor Failure Logs are now cleared (Table 2) so that hardware faults discovered during the power-on self-tests can be recorded, before the POST of the hardware actually begins.

Each processor checks its ITCM and DTCM memories exhaustively with multiple test-patterns repeatedly across all words of these memories. As per Table 1, if a TCM error is detected then the core is disabled as TCM memories are considered critical to Node- and System-Boot stages. If there are no faults then stacks and heaps are set up for both SVC and IRQ processor modes (the only modes used at this boot stage) and the processor proceeds with Node-Boot.

Scatter Loading As previously indicated, Node-Boot code is stored in the BootROM - a persistent shared read-only memory. Executing code directly from the ROM places the system interconnect under heavy access contention and each instruction is subject to heavy fetch latency. To counter this, all 18 processors execute scatter-loading routines which act according to description files provided to the linker in the build of the Node-Boot code. For all 18 cores this involves copying instruction code and data from the BootROM into the core's appropriate ITCM and DTCM locations, and initialising that processor block's state.

2.1.6 POST Self-Test for Processor Blocks

It is possible for the user to specify that the POST routines are avoided, this can be indicated by setting GPIO pin 1. If this is the case then the comprehensive tests that exercise the peripherals are not performed, similarly if the reset code for the processor is not 'power-on', then these tests are not performed again.

8 Words	Ethernet Parameters (loaded from optional SerialROM)	0xF5007FE0
4 Words	Mailbox (message passing for image loading)	0xF5007FD0
2 Words	SDRAM Information (detected size, and any errors found)	0xF5007FC8
18 Words	Processor Failure Log (18 cores)	0xF5007F80
1 Word	Monitor History (cores elected MP since power-on)	0xF5007F7C
257 Words	Shared Assembly Block (1KB + 4Byte CRC for flood filling)	0xF5007B78
	↓ Not allocated ↓	

Table 2: SystemRAM memory allocations following Node-Boot from ROM. (1160 Bytes)

The POST tests begin by exercising the non-volatile registers for each processor's Communications, DMA, Timer, and Vector Interrupt Controllers in turn. If a failure is detected in any of these components the processor is not functional so far as the Node- and System-Boot states are concerned, so the reason code is written to the failure register for that processor (Table 2), and the processor has its interrupts disabled, and is put into a low-power sleep mode (effectively shutdown).

If no errors are found the Communications and DMA controllers are initialised and enabled following their successful testing.

2.1.7 Monitor Processor Arbitration

It is necessary to select a Monitor Processor which handles the Node level functions for the chip, including the progression from Node-Boot into subsequent stages of system operation. Before the Monitor Processor can be selected the reset cause code is read once again from R12 of the System Controller. If the reset cause is a power-up then the list of historic Monitor Processors (Figure 2) is reset, so that all functional processors have a chance to become elected. Clearly only 1 processor should reset this value, and this is arbitrated using R95 from the System Controller which is a test-and-set register providing mutual exclusion (mutex). The first processor accessing this register reads a 0 in its MS bit, subsequent processors read a 1 in this bit, and wait on the first processor to signal its completion by setting Bit2 of the System Controller's R14 (Misc Control) register. If the reset reason is *not* a power-up (a soft-reset), e.g. a software problem causing a watchdog reset, or reset intervention triggered due to an unexpected condition, this may be due to an error with the Monitor Processor. To prevent this failure endlessly recurring, if a processor has already been a Monitor it will shut itself down and become ineligible for election to become the new Monitor Processor. The node is now ready to elect a Monitor Processor which is achieved by mutex hardware in the System Controller R32-R63. The first processor reading back the register following a reset event will be installed as Monitor Processor, signified by R13 in the System Controller. Additionally it marks itself as having undertaken the Monitor role in the bit-wise 'Monitor History' SystemRAM location (Figure 2). Subsequent processors will become Application Processors, and wait for the Monitor Processor to complete node-level hardware initialisation (signalled by setting Bit3 in the Misc Control register).

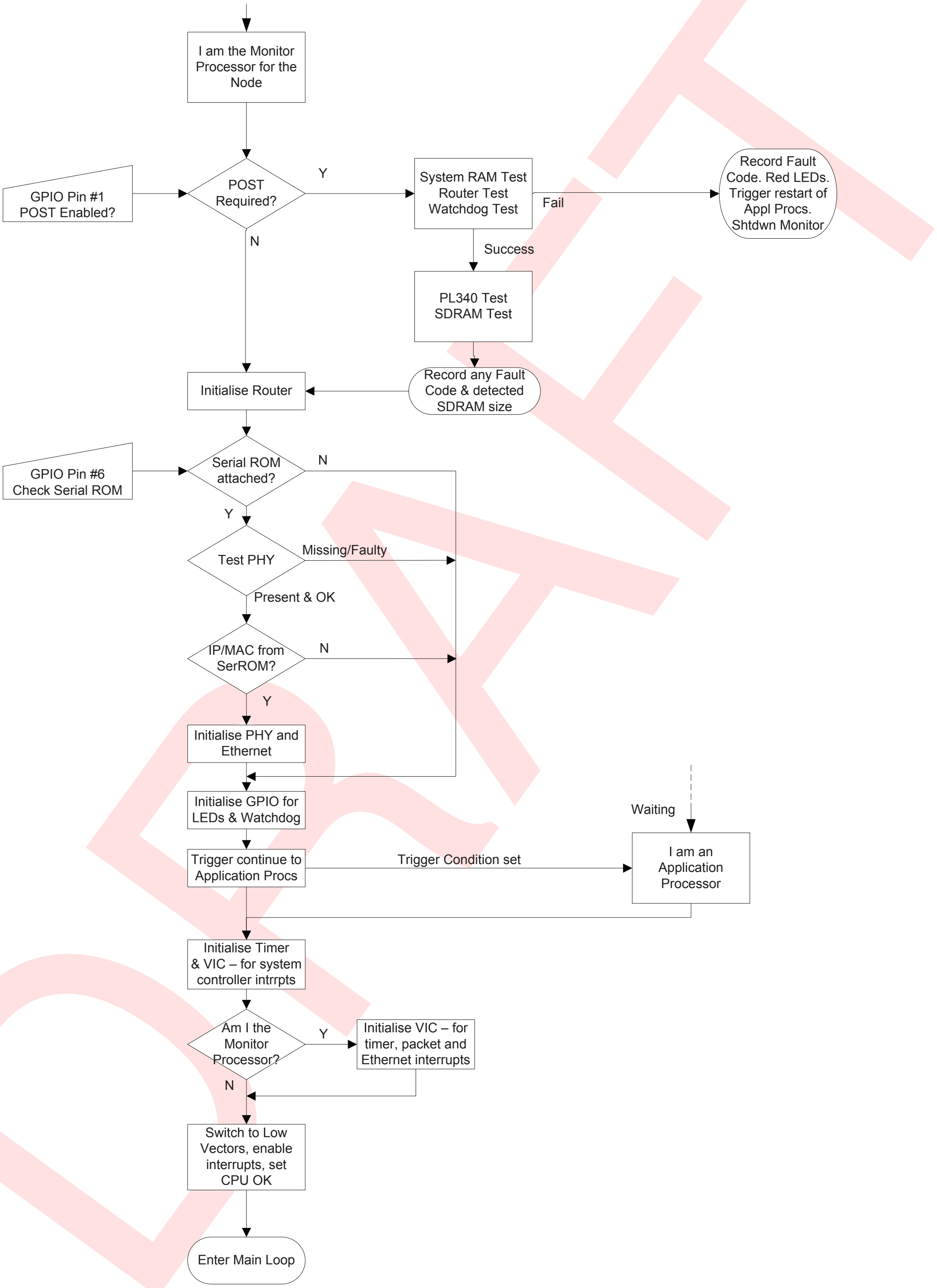
I am the Monitor Processor for the Node
GPIO Pin #1 POST Enabled?
POST Required?
Y
N
System RAM Test Router Test Watchdog Test
Fail
Record Fault Code. Red LEDs. Trigger restart of Appl Procs. Shtdwn Monitor
Success
PL340 Test SDRAM Test
Record any Fault Code & detected SDRAM size
Initialise Router
GPIO Pin #6 Check Serial ROM
Serial ROM attached?
N
Y
Test PHY
Missing/Faulty
Present & OK
IP/MAC from SerROM?
N
Y
Initialise PHY and Ethernet
Initialise GPIO for LEDs & Watchdog
Waiting
Trigger continue to Application Procs
Trigger Condition set
I am an Application Processor
Initialise Timer & VIC – for system controller intrrpts
Am I the Monitor Processor?
Y
Initialise VIC – for timer, packet and Ethernet interrupts
N
Switch to Low Vectors, enable interrupts, set CPU OK
Enter Main Loop

Figure 4: Node-Boot flow of processors immediately following Monitor Processor allocation, through until entering the listening state in the main loop. (Follows on from Figure 3).

2.1.8 Chip-Level POST

This section onwards is illustrated by the 2nd flowchart, (figure 4). The Monitor Processor continues on to perform testing on the chip-level hardware of SystemRAM, the Router and the Watchdog controllers. (This testing is bypassed if GPIO pin 1 is set (POST = off)). Any failures of these chip-level components are marked in the failure log (Tables 1 and 2), and the Monitor Processor disables its interrupts and shuts itself down by entering a low-power mode 'sleep' state. This shutting down of the Monitor Processor effectively disables the node, rather than just a single processor, and as an additional signal to the operators the red LED lines (if provisioned via GPIO output pins 1 and 7) are lit on the SpiNNaker PCB. (The Application Processors are signalled to continue into sleep mode by setting Bit3 in the Misc Control System Controller Register). In subsequent stages of boot a neighbouring chip may read the cause-code, and it may be possible to 'nurse' the node back to some level of functionality. For example, even if all processing is disabled on a node it will be useful for route optimisation to populate the routing table of the chip and to enable the router so it does not have to be explicitly routed around. For simplicity the BootROM itself does not provide any recovery functionality in Node-Boot, this will be loaded in later software stages.

SDRAM Testing This POST operates slightly differently to the other tests, as any failure detected in the SDRAM or its supporting hardware does not result in the shutting down of the Monitor Processor, consequently the chip remains active. Failure information is logged in the 'SDRAM Information' registers in SystemRAM (Table 2), detailed formats of which are illustrated in Figure 5. The testing begins with the registers of the PL340 RAM controller, a failure of which is noted in the SDRAM Error log by setting Bit0 to 1. Assuming the PL340 is functional, it is initialised with parameters for 1Gbit of Micron SDRAM running at 130MHz (Table 3. A countdown loop subsequently waits for the DLL to lock, a time-out of which leaves the SDRAM status in an indeterminate state. This 'unknown' status is registered by marking the SDRAM Error log as all 1s: i.e. 0xFFFFFFFF, and further SDRAM testing is aborted. If the DLL has locked then a series of tests are performed to:

1. Determine the size of the SDRAM
2. Note any errors when reading/writing samples of this memory (doing a full memory test was deemed too time-consuming to perform at each boot and may optionally be performed by a later software stage).

The sample tests are performed at 2^n word positions, and the recorded size (in bytes) is stored in the SDRAM size register - see Figure 5, (a RAM detected as zero-sized will be recorded as this). If any faults are recorded at a single word position this is noted in the SDRAM Error log in bit positions 1-29 indicating an error validating word 2^{bitpos} of SDRAM.

i.e. If *bitpos* $17 = 1$, then an error has been detected in word offset $2^{17} = $ 0x20000. The Byte position is therefore 0x40000, and an SDRAM fault is deduced at being at 0x70080000.

A thorough RAM test (as per the TCM and SystemRAM) is performed on both the first and last 16 Bytes of memory. This is a sample of a full functionality test using a greater combination of data and addressing lines. If an error is found in the lowest addressed part of detected SDRAM, Bit30 of the SDRAM Error log is set, and an error in the highest part of detected SDRAM will be noted as an error at bit position 31.

2.1.9 Router Initialisation

As well as populating the router tables with blank entries, this initialisation section also has the job of updating the Router Control Register (R1) with the identity of the Monitor Processor in order that packets directed to the Monitor Processor are delivered correctly across the communications fabric.

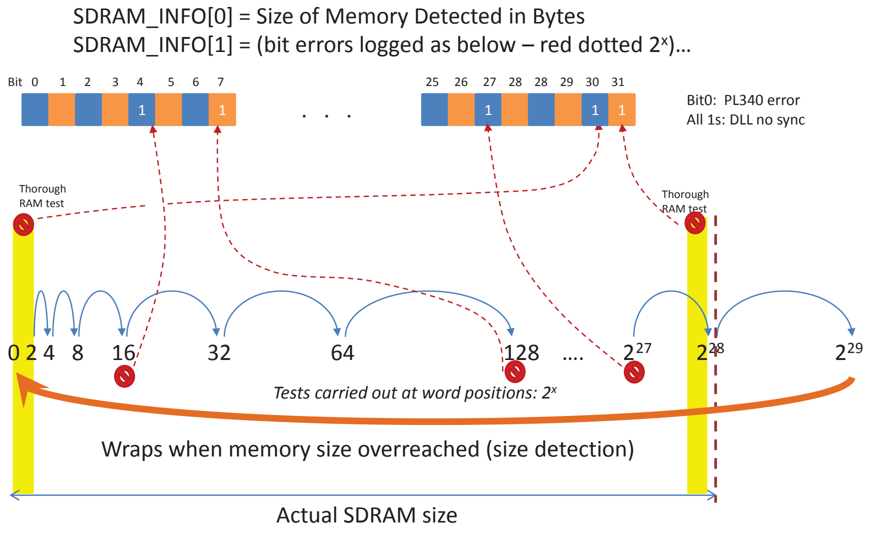
SDRAM_INFO[0] = Size of Memory Detected in Bytes
SDRAM_INFO[1] = (bit errors logged as below – red dotted 2ˣ)…
Bit 0 1 2 3 4 5 6 7
1
1
. . .
25 26 27 28 28 29 30 31
1
1
1
Bit0: PL340 error
All 1s: DLL no sync
Thorough RAM test
Thorough RAM test
0 2 4 8 16 32 64 128 …. 2²⁷ 2²⁸ 2²⁹
Tests carried out at word positions: 2ˣ
Wraps when memory size overreached (size detection)
Actual SDRAM size

Figure 5: SDRAM testing in the BootROM. Size is determined minimally by writing at increasing word powers of 2 until the wraparound is determined. More thorough tests are carried out on blocks at the top and bottom of detected size, and errors are reported in the SDRAM information fields.

multicast routing tables All 1024 × 3 fields are initialised to provide a known-state routing table, and also ensure that it does not attempt to do any other routing based on the undefined initial values in the TCAM. The outgoing state of this routine will fail-safe to only perform default routing, where received packets traverse the router and egress on the opposite external link. All 1024 key entries are set to 0xFFFFFFFF to ensure they do not match any bit, the mask is correspondingly set to 0x0 so that it provides a bit-miss across all entries. The target vector is also nullified with 0x0 to ensure packets cannot be routed anywhere, and again so that they can always be found in a sensible and useful known state at reset.

point to point (P2P) routing tables At the Node-Boot stage none of the homogeneous SpiNNaker chips are numbered nodes. Therefore all 2^{16} 3-bit fields are set to target vector 6, which means drop. Therefore any Point to Point packets arriving at the node will be dropped. This should be noted for fault-tolerance should a chip reset and enter Node-Boot.

fixed route The output vector is defined in R33 of the router and is zeroed at reset, therefore any packets arriving will be discarded at there is no onward path. No specific initialisation is therefore performed.

2.1.10 Ethernet Initialisation

All SpiNNaker chips have Ethernet controller hardware on-board, but only when this is connected to an external PHY chip is connectivity possible. As the chip receives its MAC and IP addresses from a block of data on an optional SerialROM chip (Section 2.1.4), the initialisation is only possible if this chip is present. Firstly the PHY hardware is tested by writing to, and reading from the PHY control register on the PHY chip. If the same data is sent and returned then there is a PHY, and Bit31 of the System Controller CPU OK register (R4) is set to indicate the presence of a PHY, or remains reset where there is no functional PHY. If the PHY is functional, and that there is valid IP and MAC information in the Ethernet block (Table 2), the PHY chip is set to 100Mbps Full-duplex to auto-negotiate,

PLL	Frequency
PLL1	160MHz
PLL2	260MHz

Component	Clock source
Odd Processors	PLL1
Even Processors	PLL1
SDRAM	PLL2
Router	PLL1 ÷ 2
System AHB	PLL1 ÷ 2

Table 3: Initial configuration of the phased-locked loops and clock source multiplexers.

and the Ethernet Controller is initialised with the source MAC address from the SerialROM (this is used for unicast frame matching). Any outstanding interrupts are cleared and the controller set to respond to properly addressed Unicast, Multicast and Broadcast frames.

2.1.11 Misc Chip Level Testing and Initialisation

GPIO for LEDs The 4 GPIO pins are notionally attached to LEDs on the PCB, these are set to output mode. By default up to 4 LEDs are used. GPIO 0 and 6 with green LEDs and GPIO 1 and 7 with red.

Watchdog The Monitor Processor sets up the chip watchdog time, which is a function of the AHB clock (Table 3). The watchdog clock is initialised to 100 million and thus, as the AHB runs at 80MHz, it takes 1.25 seconds for a watchdog timer to exhaust. The Node-Boot image is setup simply, not taking advantage of the warning signalling/interrupt from the Watchdog timer. Should the 2nd time-out of 1.25 seconds expire, i.e. after a total of 2.5 seconds, the chip will reset with a watchdog cause-code. In normal Node-Boot operation this will not happen as the watchdog is explicitly refreshed every 5ms (see Section 2.1.14).

Signalling Chip Level work complete The Monitor Processor sets Bit4 in the 'Misc Control' System Controller Register (R14) as it has now completed its Chip-level testing and initialisation, signalling that all Application Processors can also progress to the next stage.

2.1.12 Timer and VIC Initialisation

Timers All processors set up their timer1 to give a tick approximately every 1ms. This value is based on the 160MHz processor clock settings (Table 3), and thus as there are 160,000,000 cycles per second and 1000 milliseconds per second, the timer is set up to 160,000 cycles. All processors including Monitor and Application Processors are set up with this timer, but as is seen below with the VIC setup, the Application Processors ignore their locally generated timer interrupts.

VIC The VIC is set up to respond to different interrupts depending on the processor role as can be seen in Table 4. Although different priorities are listed, the interrupt handling in the BootROM is not pre-emptive, meaning that handling of the current event must end before another event can be dealt with - even if it has a higher priority. The Application Processors rely solely on System Controller interrupts informing them that they have a message to be processed (in the mailbox, see Table 2), whereas the Monitor Processors receive interrupts

Source	Priority	Cause	Handler	ISR
System controller	0	Monitor s/w	Appl Proc	sys_ctrl_isr() See §2.2.11
Timer	1	Counter → 0	Monitor	timer_isr() See §2.1.12
Comms. controller	2	Packet receipt	Monitor	cc_rx_isr() See §2.2.6
Ethernet interface	3	Frame receipt	Monitor	eth_rx_isr() See §2.2.4

Table 4: Vectored interrupt controller configuration.

when packets are received from the inter-chip links, by Ethernet frames arriving (if a PHY is provisioned), and the timer interrupts as mentioned above. When a processor is not dealing with the events as listed in Table 4, it enters a wait-for-interrupt (sleep) state where it uses a small fraction of its usual operating power.

2.1.13 Final Initialisation Steps

All processors, subsequent to initialising their VIC to respond to relevant interrupts, switch to low vectors ensuring that interrupt events use the vectors table at the bottom of ITCM (address 0x0). This takes care of distributing IRQ events to the VIC handler, and other exceptions which shutdown the processor after setting the error bit code in their failure log. Interrupts are then enabled for the processor as per the VIC setup (Table 4). The last instruction before entering the main operating loop sets the relevant processor bit in the CPU OK register, as all self-tests have passed and the core is operating correctly insofar as the tests have ascertained.

2.1.14 The Main Loop

Upon completion of self-testing and initialisation, all processors enter the main loop of the program, described in the pseudo-code listing below:

```
while(true)
{
    wait_for_interrupt()   // put processor into 'sleep' mode

    if(monitor && (time % wdog_update == 0))
    {
        update_watchdog()
    }
}
```

The main-loop is the kernel of the event-driven operation of Node-Boot: processors are placed into a low-power wait-for-interrupt state and only wake up in response to the interrupts listed in Table 4. On returning from an interrupt, processors are immediately put back into wait-for-interrupt state, with the exception of the Monitor Processor which periodically refreshes the watchdog counter. All processors are now in 'listening' mode in Node-Boot, which ends when a complete System-Boot image has been received by the chip over the Ethernet or inter-chip links, and that image begins its execution.

2.2 System-Boot

In order for the SpiNNaker machine to progress beyond Node-Boot, (software retrieved from the immutable BootROM), new operating code must be pushed to it. A decision was made to keep the Node-Boot image in ROM simple, and in operation for it to remain in a passive state awaiting its instructions. All nodes at power on / reset have elected a Monitor Processor which is listening on the node's six external inter-chip links (and on the Ethernet connection too - where provisioned). The code transmitted is known as the 'System-Boot' image, and as it is software originated from outside the machine, it can contain as much or as little functionality as is required. Its functions include preparing the system to load application data and code, performing further checks / diagnostics, and to facilitate numbering of the physically identical / homogeneous chips.

In summary the process begins with the responsible Host System seeding the transmission of the System-Boot image, to one or more of the Ethernet linked chips. The code is assembled and checked by the Monitor Processor on each chip receiving this transmission, and the validated code is executed by this core. The first two tasks of the System-Boot image are to send an acknowledgement to the sender, so that they can cease transmission of the image to the node, and then to self-propagate the image to the node's immediate neighbours over the chip-to-chip links (a process known as flood-fill). As Node-Boot code consists entirely of passive data reception to reduce complexity, the System-Boot image is repeatedly transmitted to their targets until either an acknowledgement is received (from the System-Boot code successfully running on the target), or if the maximum repeat count threshold is met.

The following sections contain the required detail to create a System-Boot software image, and to transmit it to a SpiNNaker instance in order that the machine may successfully exit Node-Boot.

2.2.1 State of SpiNNaker Node after Node-Boot

When creating a System-Boot image the state of the system running Node-Boot should be taken into account. This state is a combination of power-on / reset hardware initialisations, as defined within the SpiNNaker data sheet, and the operations and initialisations performed by the Node-Boot software itself. This section details the latter, in order that a System-Boot image may be successfully created and operated.

2.2.2 State of Each Processor Block Level

For each processing core beginning execution of System-Boot image, all interrupts (IRQ/-FIQ) are turned off and the processor is in SVC mode with low vectors set. The following details the status of each processor block (18 per node):

Vector Interrupt Controller - VIC For both the Monitor and Application Processors Vector 0 is used for the System Controller interrupt (Source 18). This is used in Node-Boot to indicate a message has been passed from Monitor to Application Processors in order to populate memory blocks and execute as per Section 2.2.11. No other vectors are set for Application Processors which are entirely submissive to Monitor control via this message passing technique. For the Monitor Processors, vectors 1-3 are also used. Vector 1 is used by the timer1 interrupt (source 4), vector 2 is used by the Communications Controller packet received interrupt (source 6) and vector 3 is used when an Ethernet is attached to the chip to indicate an Ethernet frame has arrived and requires processing (interrupt source 20).

All other vector controls are initialised to be disabled and use interrupt source 31 (formerly null interrupt, but actually GPIO 7), the vectors themselves are initialised to address

0x0 (reset in low vector operation). In reality due to all interrupts being disabled at entry to System-Boot, it is safe to overwrite all these vectors as required as they are consistent with the Node-Boot code only. However the user must take care to acknowledge raw interrupt status here, and in the peripheral, as necessary for their needs.

All entries are set up for IRQ operation and not FIQ.

Timer Controllers Timer2 is not used. Timer1 is enabled for periodic operation, generating an interrupt every 160,000 cycles. This periodicity is a function of the clock rate applied to the processors of 160MHz in Node-Boot (see Table 3), giving a 1 millisecond timer interrupt. Note, as in Section 2.1.12, only the Monitor Processor is set up to actively respond to interrupts from the timer1 interrupt, this millisecond timer is also set up on the Application Processors, but the interrupt is not handled and effectively ignored.

On Monitor Processors the timer interrupt is used by the Node-Boot code to:

1. Flash the Green LED (GPIO line 0) every 2^{10} ticks (approximately every second)
2. Send an Ethernet ‘Hello’ message to enable auto-discovery every 2^{12} ticks (approximately every 4 seconds)
3. Reset the Watchdog timer periodically

DMA Controller All DMA controllers are initialised with the CRC-32 polynomial.

Comms Controller The route field of the ‘Source Address and Route’ register (R6) is set to 0b111. This ‘fake’ route is used to indicate packets originating from this processor are internal packets. This field (for diagnostic purposes) may be set to ‘fake’ its packets as originating from a specified external link when entering the routing engine.

2.2.3 State at SpiNNaker Chip (Node) Level

The following initialisations are performed on a node level - i.e. are shared by all hardware and all processors on that SpiNNaker chip. This configuration is performed by the elected Monitor Processor which has overall chip-level responsibility for the node:

Router The times for Wait1 and Wait2 stages are initialised to 108 (0x2F) cycles. All routing entries in the TCAM Multicast table are initialised to all 1s for the key, with a zero mask to ensure no match, and the key is initialised to 0 - no egress interface for the router (nowhere). The Point-to-Point routing table is also initialised, with all entries set to drop (0b110). The entry for Fixed Route is initialised to ‘nowhere’ as part of the hardware initialisations.

SDRAM Memory Controller (PL340) The PL340 is initialised for Micron DDR memory, and provided with a clock of 130MHz (see Table 3), which is deemed a ‘safe’ rate to operate at. The Power-On Self-Tests (Figure 5) initialise the SDRAM size and error registers in Table 2 per Figure 5. Note these registers are only populated if the POST is performed (GPIO pin 1 Section 2.1.6 above) - if the POST is not performed then this memory will not be initialised. To a high-degree of certainty the values in the registers may be validated by noting there being only 0 or 1 bit set in the size register (a valid power of 2) - or by reading GPIO pin1 in input mode.

System Controller LEDs are expected to be attached on some or all of the GPIO pins 0,1,6 and 7, therefore the direction of GPIO is set to output on these lines, (LEDs here are typically configured active low).

R95, a mutual-exclusion register was used to select a single processor to clear the Monitor History field in SystemRAM if the reset cause was a power-on.

Three undocumented bits within the Misc Control Register R14 are also utilised by Node-Boot. Bit1 is used when there is a SerialROM, it is set high after control returns from the SerialROM read operation. Therefore if valid data has been read from the optional SerialROM, and control is handed back to the BootROM process this will be high. Bit2 is used as a synchronization bit around the clearing of the Monitor History field after a power on reset as above. Therefore if this bit is set high, the cause of the last reset may be indirectly inferred as a power-on, and a Monitor was elected. Bit3 is used as a signal that the Monitor Processor has completed the node-level system initialisations and that all other processors may proceed. This bit can be used to determine that the Application Processors have been signalled to go into low power sleep mode.

Within the system controller the tested state of the 18 processors is recorded in the CPU OK register (readible in R4 and R5). Bit*n* represents the physical core number and if set 1, then the processor block has passed all the tests performed upon it. If it has failed any tests this is not set and a more detailed failure code is recorded in the processors failure log. (Tables 1 and 2). Bit31 of this register is used to record whether or not a PHY is detected attached to this node, with a 1 indicating that there is.

In R13 Monitor ID the physical processor elected as Monitor is recorded, this is used by the router in order to determine which processor to direct traffic which must be handled by the Monitor Processor.

Ethernet (and PHY) If an Ethernet connection is to be provided to a SpiNNaker chip this is done so via a PHY (Physical Layer) chip. The PHY is serially attached (via GPIO pins) to the Ethernet controller within the SpiNNaker node. As detailed, if a PHY is detected (Bit31 of the CPU OK register is set), and the PHY is set to 100Mbps Full-Duplex mode and then set to negotiate this. The model of PHY used in this instance is irrelevant as the PHY registers used to specify this are standardised. In addition to a PHY chip a SerialROM should also be provisioned, (via other GPIO pins), and a block of data within this ROM contains the unique network parameters to be used for this SpiNNaker node (see Table 5). This data is copied to the Top of SystemRAM as the Ethernet data block and contains both Data and Network layer information (MAC and IP addressing). If this data is successfully read (the first word populated is non-zero), then the local MAC address of the Ethernet controller is set, hardware byte re-ordering is used, and frames received with errors are dropped. The controller is enabled for transmission and and receipt of broadcast, multicast and unicast Ethernet framed traffic. NB. The user may wish to *disable* byte re-ordering if this is the preference for the software they are writing (also known as host ordering).

Watchdog The watchdog is ON. Therefore the System-Boot code must deal with the watchdog timers, or the chip will be reset, triggered by the watchdog timer expiring. The watchdog has its reset output enabled, as well as its counter. The register is loaded with the value 160 Million (0x5F5E100), due to the System Bus at Node-Boot being 80MHz (Table 3), this provides for a period of 1.25 seconds until it counts down to 0 and is exhausted. On the first expiration, an interrupt is raised, and on the second the reset is activated. Therefore without suitable resetting of the watchdog, around 2.5 seconds after a System-Boot the watchdog reset will be invoked. In Section 2.2.13 a bespoke mechanism for recovery from an inadvertent watchdog failure is detailed.

Register	Word	Sub-Reg.	Comment
Network Block	0	Bit0	1 = Ethernet detected, 0 = none
		Bit15	SerialROM has populated this data
		Bytes 2-3	MAC 1-2
	1		MAC 3-6
	2		IP Address 1-4
	3		Default Gateway 1-4
	4		Subnet Mask 1-4
	5	Bytes 0-1	Suggested UDP Port Number
	5	Bytes 2-3	Not Used
	6-7		Not Used
Message Passing	0	Bytes 0-1	OpCode: 0x5EC[op]. op=1: copy (DMA), op=2: copy (regwise) & branch (execute)
		Bytes 2-3	Length of Data (Bytes)
	1		Source Address (for op=1 must be shared address as DMA used, op=2: any)
	2		Destination Address (op=2: must be local TCM address as DMA used, op=2: any)
	3		Execute Address
SDRAM parameters	0		SDRAM size (Bytes)
	1		SDRAM faults detected (see Figure 5)
Processor Failure Log	0-17		No Fault = 0, Fault != 1(see Table 1)
Monitor History	0	Bits 0-17	Bit=1 if core has been Monitor
Block Buffer	0-256		For Flood Filling Block Assembly + CRC

Table 5: Detail of Fields located in the top of SystemRAM. Data is stored in Host Byte order.

Memory Map It should be noted that the top section of SystemRAM is used by NodeBoot to store system variables and diagnostic information. This can be seen in outline form in Table 2. A more detail appraisal of the contents of these fields is found in Table 5.

Addresses beneath the Monitor History field may be used for other purposes after System-Boot, otherwise it would be prudent to maintain the remainder of these fields in the top of SystemRAM, particularly the ’Monitor History’ field as it is used by the IVB mechanism in the event of a watchdog caused reset (Section 2.2.13). The remainder of SystemRAM not detailed by Tables 2 and 5 is available for use at all times on the node.

2.2.4 Ethernet / Host Boot

The host machine pushes out a System-Boot image using raw UDP SpiNNaker-type packets to the root chip(s). The Monitor Processors on these route chip(s) are listening for this transmission on active Ethernet interfaces. In order to facilitate automated discovery of SpiNNaker nodes, each chip having an active PHY transmits out a broadcast ‘Hello’ message on the local Ethernet every 4 seconds triggered by the timer interrupt as above. These broadcast messages are transmitted to IP destination 255.255.255.255 with target MAC FFFF.FFFF.FFFF (as the Host is unknown the the chip at this stage). These broadcast destinations ensure that the ‘Hello’ messages are not routed beyond the local Ethernet, and are only transmitted once every 4 seconds to ensure they are not burdensome to the local LAN. A SpiNNaker chip will respond to ARP requests made to its IP address, and will also respond to unfragmented ICMP echo requests (pings) of up to 1472 Bytes.

An example of a SpiNNaker ‘Hello’ message transmission, can be seen in Figure 6. This

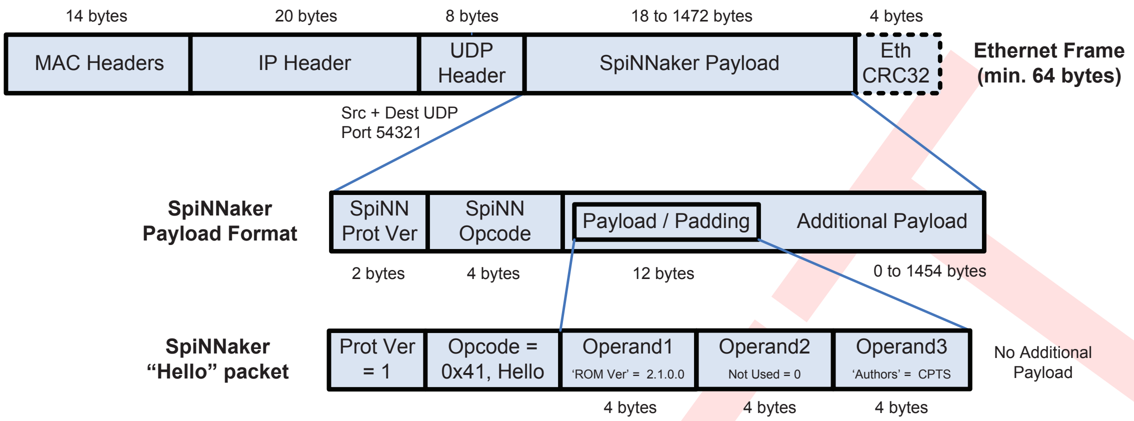

Figure 6: The basic SpiNNaker Ethernet framing/packet format (used by Node-Boot). An example of the 'Hello' packet is also illustrated, broadcast by each Ethernet attached node every 4 seconds.

same simple packet scheme is used by the Host to transmit the System-Boot image across to the machine (Figure 7) using 3 different packet formats:

1. Flood-Fill Start
2. Flood-Fill Block
3. Flood-Fill Control

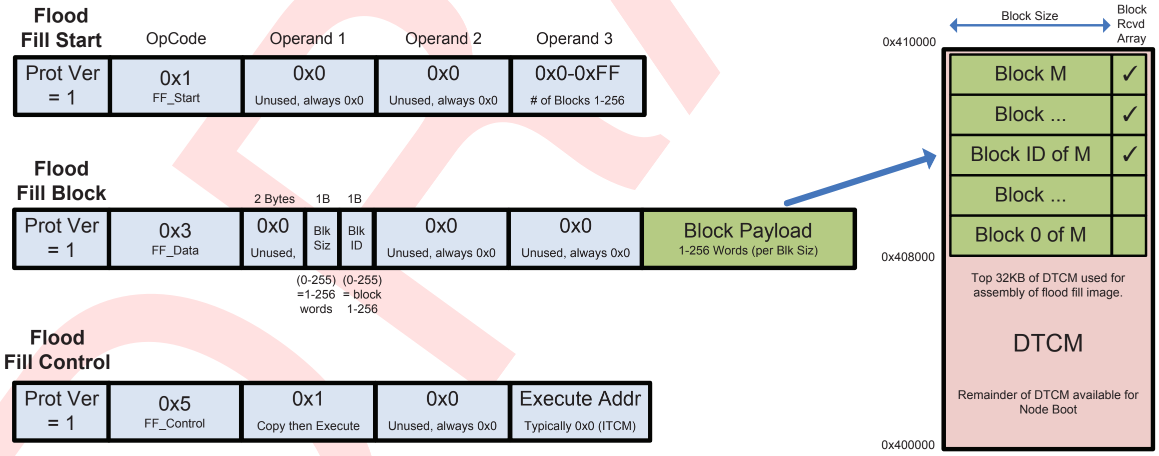

Figure 7: The three packet formats used by the Host pushing a System-Boot image to a SpiNNaker node, and the assembly of the data blocks into the image in DTCM.

Flood-Fill Start When a SpiNNaker chip receives the Start message it readies itself for receipt of an image of up to 32KB split up into the number of blocks indicated in the start message + 1 (a range of 1-256). The image is assembled in the top half of the Monitor Processor's DTCM (Figure 7). A receive array is initialised empty with an entry for each Block ID expected.

Flood-Fill Block Data is now transmitted block by block by the host to the SpiNNaker system. Block IDs are numbered beginning 0, and the block size in words is indicated +1 (a range of 1-256 words). Note blocks MUST all be equal in size - it is not permitted for the block size to change between blocks as this is used as an offset in assembling the image. If the final block is not full, then the data should be padded by the transmitting host, and the same block size used. Clearly, the amount of data ($blocks \times blocksize$) should not exceed 32KB, and it makes most sense to use a block size divisible by the maximum image size. In testing up to 32 blocks of 256 words were used as they have the lowest overhead, and this is recommended for production use. If a block is successfully received then the data is copied to the appropriate position in the assembled image, and the receive array is updated to indicate that the block with this ID is in place.

Flood-Fill End Following transmission of the Start and all Block packets, the host transmits the Control packet. When a SpiNNaker chip receives this it firstly validates that all the blocks in the receive array are populated. If they are not (for example a frame has been lost), then it continues to listen for the missing Flood-Fill Blocks. Typically however, all blocks are in place at the first attempt, and the system copies the assembled image over to ITCM 0x0 - and branches to the indicated start address for execution. Typically the start address will be also be at 0x0.

2.2.5 Notes on the Host System

The Host system should send SpiNNaker packets formatted as per Figure 7 to the SpiNNaker machine. The ordering of bytes to be used is network order (ie. the first byte onto the wire is the most significant). A sensible inter-packet delay should be employed to maximise the first transmission success - it has been found that transmitting at a rate of ~1ms per block is dealt with well by the chip. As the SpiNNaker packets are sent as datagrams, the receipt of all blocks is not guaranteed, in-fact the host system may be remote - somewhere on the Internet. As the SpiNNaker chips are quiescent, passively listening to image data sent to them - they are unable to request specific blocks to be re-transmitted which are missing. Therefore in these circumstances the host is relied upon to retransmit the missing data, and as this host does not know which blocks are missing, it must retransmit full set. It is therefore suggested that the System-Boot image includes a 'cease transmission' message to the host, in order that the host knows it may stop transmitting to that chip, or a limitation made on the number of image retransmissions made by the host.

Other IP packet formats may be supported by the System-Boot code in order to provide additional functionality, for example communication bridged between the IP and SpiNNaker processor domains, and indeed down to the thread level. The simple SpiNNaker IP format used to get to System-Boot is designed as simply as possible to deliver System-Boot code reliably. More detailed packet format specifications, such as SpiNNaker Datagram Protocol (SDP) are beyond the scope of this document.

2.2.6 Inter-Chip Boot

Sending the image from the Host to Root Ethernet attached chips is the first stage of the flood fill. When running the System-Boot image the top 32KB of DTCM is still home to the image, and the System-Boot image should flood-fill itself out to its immediate neighbours. This is done on a link-by-link basis using Nearest Neighbour (NN) packets (other packet types are not available at this stage as their routing tables are not populated). The NN packet type provisions a 8-bit control field, a 32-bit key and an optional 32-bit payload. The payload is nowhere near as large as available in the SpiNNaker UDP/IP datagram, therefore an additional level of hierarchy is added to the flood-fill transmission as for each block multiple packet transmissions are required. Additionally it is not possible to make use of the inherent CRC-32 check within the Ethernet hardware, so a combination of validation methods to verify data transmissions is used. There is a locally calculated checksum at the packet level, and at the block level the SpiNNaker programmable CRC hardware is taken advantage of to generate and validate checksums at each end of the link.

It is recommended that each flood fill packet is sequentially sent around all the six external links. The nearest neighbour packet facility does offer the opportunity to send out each packet on a combination of links simultaneously, however if *any* of the output links becomes blocked due a fault or congestion, then *all* transmissions to all interfaces become stalled. Therefore by transmitting individually round each link in turn such a fault is constrained to a single transmission path. Clearly as the data propagates around the machine, due to the inherent multi-path network connectivity each node will receive the flood-fill data multiple times. Therefore while the node is being flood filled over its inter-chip links, duplicate packets are discarded. Subsequent to a successful image load, a similar strategy can be taken to the Ethernet flood-fill by either generating an acknowledgement packet ceasing transmission on a link, or by ignoring the flood-fill packets until the repeating transmissions dampen down. Transmissions are not system-wide broadcasts, they occur in waves originating from each node that reaches System-Boot, so there will not be a broadcast storm in flood-fill, just wave-front(s) of transmissions (an example of which is shown in Figure 8).

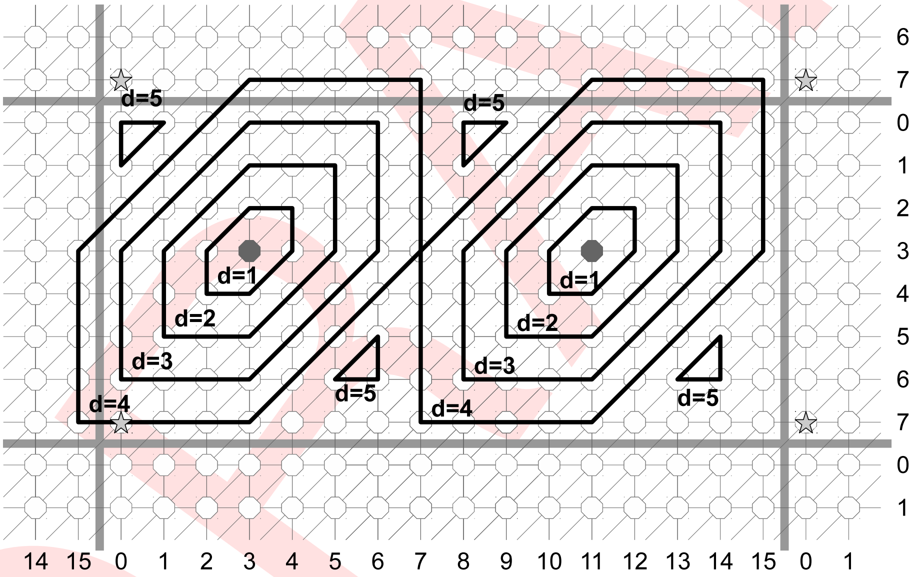

Figure 8: 'Waves' of System-Boot flood-filling a tessellated SpiNNaker torus over time. The two dark nodes represent two Root Ethernet attached chips seeded by the host, the star represents the origin of coordinates node. Distance from the Root node is noted by *d*.

Due to the small packet payload, the original model outlined in Figure 7 for Ethernet is extended to a 5 packet-type model (shown in Figure 9):

1. Flood-Fill Start
2. Flood-Fill Block Start
3. Flood-Fill Block Data
4. Flood-Fill Block End
5. Flood-Fill Control

It should also be noted from Figure 9 that each packet is transmitted with a 4-bit, ones complement checksum calculated across the key and payload, any packet where the checksum received does not match a locally generated version is discarded. This check is in addition to the intrinsic parity check on packets passing through the router.

Flood-Fill Start In common with the Ethernet attached nodes, when a SpiNNaker chip receives the Start message it readies itself for receipt of an image of up to 32KB split up into the number of blocks indicated in the start message + 1 (a range of 1-256). The image is assembled in the top half of the Monitor Processor's DTCM block by block. A receive array is initialised empty with an entry for each Block ID expected.

Flood-Fill Block Start The Block start message indicates the block ID, and the size of the block. If this block has not already been completed, a received word array is initialised with an empty entry for each word in that block. Subsequently these data words are listened for. In order to later take advantage of the CRC hardware the words are collated into the block of data in SystemRAM (see Table 2 and Figure 9).

Flood-Fill Block Data The block data message indicates both the block ID and the Word ID within that block. Words from blocks other than the one being listened for are discarded - only 1 block is populated at a time. The word itself is carried in the payload, and if it has not already been received is copied into the space used to collate the block. In addition the received word array is updated to indicate that this word of data was successfully received.

Flood-Fill Block End Firstly a check of the received word array is carried out to ensure that all the expected words are populated. If the list is not complete then the software waits for a retransmission of the missing words, however typically there is a complete set and the data block can now be validated. For the indicated block ID a CRC-32 is incorporated as the payload of the block end message. At the transmit end this can be generated in software, or via the hardware supported DMA process similarly to the receive checks. At the receive end the CRC check is generated by performing a DMA from the block assembled in SystemRAM into the appropriate position in the DTCM image, with the CRC option enabled. The received CRC-32 is then compared with the one calculated locally as part of the DMA transfer, and if they match, the entry in the received block table is populated (Figure 9). If it fails, then the whole block is discarded as it is impossible to calculate which word is in error (the 4-bit 1s complement packet checksum is not particularly strong). The system then listens for the next Block Start or Control Message.

Once again it is mandatory that the 'Block Size' field is maintained consistently across ALL blocks including the final one (padded if not fully populated). Typically this is achieved by always retransmitting the maximal image size: e.g. 32 blocks of 1KB (256 words), or 256 blocks of 128Byte blocks. The transmission and assembly times are not significant, and are in the order of ∼30ms per link, (or 1ms per 1KB block).

Flood-Fill Control Finally the control message is transmitted, which contains as payload the address to which to branch to begin executing the System-Boot image. The receiver firstly validates that all the blocks in the receive array are populated, if not it continues to listen for the missing blocks. Typically however, everything is in place, and the system copies the assembled image from the top of DTCM over to ITCM 0x0 - and branches to the indicated start address for execution. Note: The start branch address must be hard-coded into the System-Boot image at compile time in order for the transmitter to know how to set it. In 90+% of cases this is expected to be the standard base address in ITCM of 0x0.

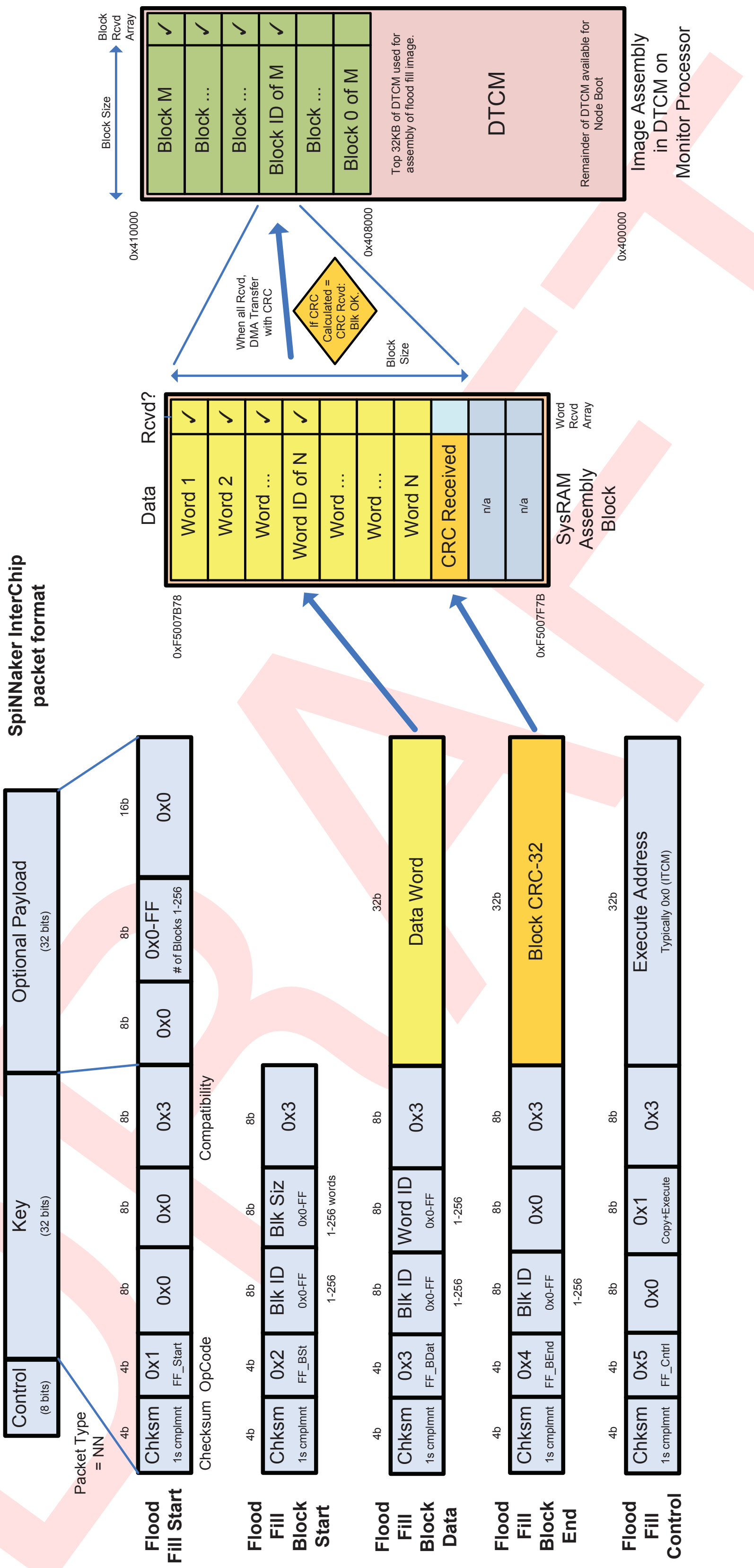

Figure 9: The five packet formats used by the chip-chip flood filling of a System-Boot image, detailing the staged validation and assembly of the data into the DTCM image.

2.2.7 System-Boot Loading - A state machine

The state of the SpiNNaker node during the loading of a System-Boot image may be considered as a state machine as detailed in Figure 10. Note that if a start message is received either on Ethernet or an inter-chip link the state-machine remains in this mode, there is no 'interworking' of the two methods.

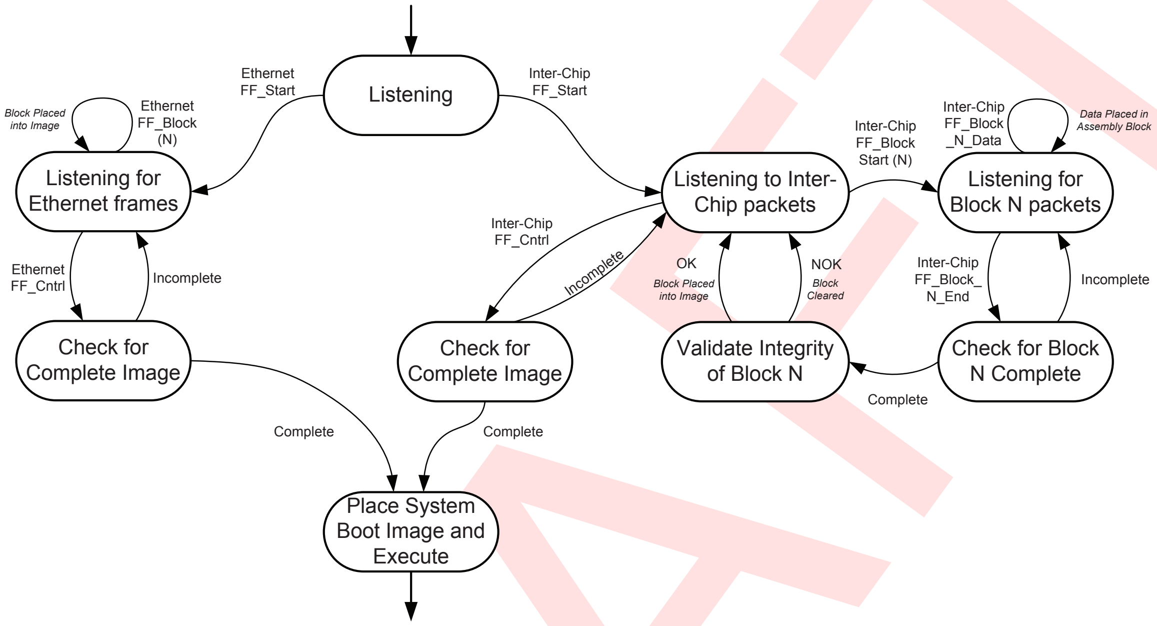
Listening
Ethernet FF_Start
Inter-Chip FF_Start
Block Placed into Image
Ethernet FF_Block (N)
Listening for Ethernet frames
Ethernet FF_Cntrl
Incomplete
Check for Complete Image
Complete
Inter-Chip FF_Cntrl
Incomplete
Check for Complete Image
Complete
Listening to Inter-Chip packets
Inter-Chip FF_Block Start (N)
Inter-Chip FF_Block _N_Data
Data Placed in Assembly Block
Listening for Block N packets
OK
Block Placed into Image
NOK
Block Cleared
Inter-Chip FF_Block_ N_End
Incomplete
Validate Integrity of Block N
Complete
Check for Block N Complete
Place System Boot Image and Execute

Figure 10: The state diagram transitioning from Node-Boot to System-Boot via the Flood Fill from the Host or a neighbouring chip.

2.2.8 Immutable Data Copier

Node-Boot itself operates in ITCM, therefore copying of the System-Boot code into ITCM after assembling the image in DTCM is handled by a routine pre-loaded into the immutable BootROM. This small assembly routine can copy data around and branch arbitrarily for execution. Using the code in the BootROM ensures the program counter remains outside the TCMs and enables the full set of memories (TCMs, SystemRAM and SDRAM) to be available as destinations. e.g. to copy the assembled DTCM image into the base of ITCM (0x0) the following would be performed:

```
LDR R0,0x4008000  %(src addr),
LDR R1,0x8000     %(len 32KB),
LDR R2,0x0        %(dest addr),
LDR R3,0x0        %(execute addr),
B 0xF60004AC      %(start the copy and execute ROM routine).
```

These are effectively the steps taken when reception of the flood-filled image is complete in order to move and execute the System-Boot image. It may also prove to be useful for authors of System-Boot (and later) code for other purposes, hence its documentation here.

2.2.9 Block size choices and Retransmissions

All packets on both Ethernet and inter-chip links are transmitted as datagrams, with no guarantee of delivery, so mechanisms are built in to compensate for any loss/errors. Purely in terms of data transmitted, Ethernet is usually twice as efficient as the inter-chip link method:

- A 32KB Ethernet flood fill has 1 start message, 32 blocks and 1 end message: 34 packets, 34944 bytes, 94% efficient.
- Worst case Ethernet 32KB: 1 start, 256 blocks, 1 end: 258 packets, 49280 bytes, 66% efficient.
- For the inter-chip transmissions there is 1 start, 32 block starts, 32 × 256 block data, 32 block ends, 1 control: 8258 packets, 74322 bytes, 44% efficient,
- Worst inter-chip case: 1 start, 256 block starts, 256 × 32 block data, 256 block ends, 1 control: 8706 packets or 78354 bytes with 42% efficiency.

There is far less to be gained by having large block sizes in the internal system compared with the Ethernet as can be seen, in a high noise environment however the shorter block sizes would result in fewer corruptions of individual blocks for the same number of bit errors, and as the transmissions are repeated sequentially this may be a valid approach to take.

2.2.10 Pertinent notes on this process / creating a System-Boot image

An image of up to 32KB may be used by System-Boot, the image is assembled as in Figures 7 and 9, within the upper half of DTCM memory before being transferred to the base of ITCM at memory address 0x0 and executed. Directly it is only possible to load code into the ITCM of the Monitor Processor on each node, although a user-defined execute address may be applied. If it is desired for an image to be loaded elsewhere, this may be done indirectly by applying a small (5-10) word piece of code prefixed to the image, and to make use of the copier code in BootROM (Section 2.2.8) to shift the image around to the desired location.

Specifically, this strategy could be used, prefixing a piece of machine code that populates of R0 to R3, and then branches to the immutable copier code. By branching directly to this routine the actual image code anywhere can be copied anywhere in memory. The code would need to be built to scatter into the chosen location if this required.

e.g. Compile an image to position and execute at 0xF5000000. Create the prefix code with the 5 assembly (meta)instructions below and attach this to the beginning of the binary image code. Flood fill the compiled image to the chip and execute at 0x0 (the prefix code).

```
LDR R0,0x14         %(src addr), e.g. if the prefix code is 5 words of ARM
LDR R1,0x4000       %(len), for example 16KB, (<=(32KB-prefix length))
LDR R2,0xF5000000 %(dest addr)
LDR R3,0xF5000000 %(execute addr).
B 0xF60004AC        %(start the copy and execute ROM routine).
```

Executing this would now leave the processor running the required code from the correct location of 0xF5000000. Of course if required propagate the prefixed code out further by flood-filling to the system.

2.2.11 Application Processor Loading

Once a System-Boot image has been loaded onto the Monitor Processors of all chips in the SpiNNaker system, the system may be numbered. This enables the capability of application code being downloaded to designated Application Processors. The residual code loaded in Node-Boot to the Application Processors awaits instructions to be delivered to it via a well-known mailbox that it processes on receipt of a system-controller interrupt. These instructions can include copying of program and application data across the system bus from a shared system memory into private TCM memory, and execution as required. Further details about this message format and mailbox location can be found in Table 5.

Due to the System-Boot image loaded on the Monitor Processor affording more functionality that Node-Boot, an Application Processor image may be uploaded to, and placed in a shared memory location on the node (in any suitable location). The Application Processor is then sent a system controller interrupt, which instructs it to look into the mailbox in SystemRAM (Table 5), interpret the message and copy the image between the required memory locations and begin execution as required.

2.2.12 Homogeneous Image Loading To All Processors

It may be desirable for the same image to be loaded to both Monitor and Application Processors on all nodes. It is possible to do this in a number of ways, but the most efficient involves only a single transmission of the data (at Node-Boot) across the network. Here is one approach to meeting this requirement:

In the System-Boot image compiled, have a conditional execution section at the beginning for Monitor Processors only:

1. As the full constructed DTCM image is still intact on the Monitor Processor, (DMA) copy this out to a 'known place' in Shared memory. e.g. 0x700000000 (SDRAM). This place must be safe from being overwritten at this stage.

2. Work out which Application Processors need the code... e.g.

   (a) Read the CPU OK register. Logically OR with 0x3FFFF. This gives us the list of active CPUs.

   (b) Clear the bit at our own processor ID from this list (as this processor is Monitor and already have the image). (e.g. minus 2^{proc_id}, or minus $2 << proc\_id$)

3. In the Message Passing block (Table 5), set the following:

   (a) 1st word (ie, 0xF5007FD0) set Opcode: 0x5EC20000 + image length in bytes.

   (b) 2nd word Source: 0x70000000 (or wherever the image is stored in shared memory).

   (c) 3rd word Destination : 0x0 (if the image runs from here).

   (d) 4th word ExecuteAddr: 0x0 (same)

4. The Application Processors may now be notified there is a message for them. This is done by setting the appropriate System Controller interrupt bit for them. (Option: all at once, or one at a time). When this interrupt clears it is known that the homogeneous code is executing on that Application Processor.

Following this - the Monitor performs the flood-fill out on the inter-chip links as per usual. This will ensure that all processors on a node get to execute the same image.

Of course it may be useful to use a combination of techniques, including using the prefixed code approach outlined in Section 2.2.10 above. It all depends on the desired result.

IVB Magic Number: 0xC0FFEE18		0x7FFC
Grand CRC		0x7FF8
CRC 31	↑	0x7FF4
CRC 30	↑	0x7FF0
CRC 29	↑	0x7FEC
...	↑	...
CRC 0	↑	0x7F78
Recovery execute address	↑	0x7F74
IVd area size (bytes)	↑	0x7F70
IVB area start address	↑	0x7F6C

Table 6: ITCM validation block structure.

2.2.13 IVB - ITCM Validation Block - a technique for graceful recovery of Watchdog resets

Should a Monitor Processor within the SpiNNaker machine experience a software fault, it is useful to enable the watchdog timer so that the chip experiencing malfunctioning run-away code on a processor is signalled to reset. As the Watchdog is a per-chip level peripheral, it is usually less useful to use this technique on an Application Processor. Should the user wish to have watchdog-type functionality this can be enabled in software (e.g. resetting shared register/memory location periodically) and having the Monitor processor validate the Application Processors. Should a Watchdog reset occur then usually the chip would return to the Node-Boot code, reset its routing tables, and end up quiescent awaiting new code to be sent to it.

This behaviour is not optimal during a simulation, as the surrounding chips will be running their application code and may not have code to restore the reset node, and the routing paths may be interrupted. The chip that has been reset will require recovery through all phases of boot, from Node-Boot to System-Boot and then to Application-Load before it may once again play an active role. However, it should be noted that should the fault have actually been a transient fault / glitch, then much / all of the operating environment may still be in place.

To facilitate recovery from such a situation, an optional ITCM Validation Block (IVB) may be installed at the top of ITCM. The IVB is a series of checksums and magic-numbers that can be used to circumvent going directly to the Node-Boot phase on a reset. This facility permits a recovery routine to be executed in the node whenever a watchdog reset is triggered if the IVB block is valid. If the IVB facility is initiated, but there has been some corruption of the instruction data that the IVB block protects, then the node will enter the Node-Boot routine as default.

Table 6 describes the structure of the optional IVB. This should be populated in this exact location by the image (which is is being setup for recovery) and consists of several fields. Firstly a ‘Magic Number’ is checked - only if this is in place then further checks are performed. For each 1KB of ITCM a 4-byte CRC is stored which is calculated using the programmable CRC on-board the SpiNNaker chip. Therefore for the check to succeed then the same CRC polynomial must be in place. Furthermore there is a CRC of CRCs (a ‘Grand CRC’) - this is to ensure that the IVB itself has not been corrupted. The block also contains start and length fields of a contiguous ITCM block, and a recovery address which should be branched to if all the IVB CRC checks succeed.

IVB Block Creation Creation of the IVB block is an iterative process performed optionally by the ultimate code operating on each processor core (i.e. not intervening images such as Node-Boot or System-Boot). It involves the creation of the IVB block as seen in Table

6. The Start address is usually 0x0 (the base of ITCM), the size typically the full ITCM memory minus the IVB block size. i.e. 32,620 Bytes. (There are limited circumstances where the IVB may be smaller, or point to different addressing taking into account the individual required circumstances). The start address will be a 'Recovery' routine within the code which will recover the image to a 'known good' state.

Firstly populate the start, length, and recovery execution address to the block. Then populate the CRC fields, and finally the Magic Number. An example pseudo-routine to populate the CRC 0-31, and Grand_CRC fields is as follows:

```
for (i=0;i<((ivb_size)/1024);i++)  { // loop for # of <=1KB blocks
  IVB_CRC[i] = DMA_CRC(ivb_start+(i*1024), shared_assembly_block,
            i==((ivb_size-1)/1024) ? ((ivb_size-1)%1024)+1 : 1024);
    // DMA <=1KB block from ITCM source to SystemRAM to generate CRC32.
}
GRAND_CRC = DMA_CRC(IVB_BLOCK, shared_assembly_block, 35*4);
  // Grand CRC calculated over CRCs and block description words
```

The DMA_CRC pseudo-function takes a source address, target address and length field, and using DMA with the CRC turned on generates a CRC for the data to be populated in the CRC locations of the IVB block.

Process of IVB recovery In the event of a Watchdog caused reset code, a recovery routine built into the Node-Boot image will validate the protected ITCM memory has not been trampled by malfunctioning code. A number of simple steps are performed, which mirror the steps in the IVB creation section above, just not necessarily in the same order:

1. As per the initial steps in Node-Boot (Figure 3), if the Reset Reason is a Watchdog then the IVB recovery option is initiated.
2. A simple check is made of the IVB Magic Number field for the expected value. If this is valid then this indicates:
   (a) The programmer has provisioned an IVB block to be evaluated.
   (b) An attempt should be made to recover the code.
3. The Grand CRC is validated to ensure the integrity of the block CRCs and the image parameters. This is performed by using DMA to copy the 35 words of data below the Grand CRC into the Assembly Block in SystemRAM (Table 2).
4. If the Grand CRC is good, then using the size data, block-by-block check the IVB block CRCs, by using DMA with CRC to copy the relevant ITCM data block into the Assembly Block, and checking calculated versus IVB Block CRCs.
5. Assuming all Blocks have validated, then the stored IVB Magic Number and Grand CRC are deliberately invalidated to ensure that it is not possible to indefinitely loop here - i.e. if an endlessly repeating code fault, rather than a transient fault.
6. The 'Recovery' routine is started by branching to the Recovery Execute Address.

If at any point any of the tests and checks fail, then the usual Node-Boot process is initiated as per Figure 3. Advanced techniques of validation may also be used by inclusion in the recovery code, for example to validate DTCM and other memories, but as this is achieved from instructions in the protected ITCM area, if required this is an exercise for the user.

3 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.

3.1 Run-time software stack

Figure 11 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 4

3.2 Inter-processor communication

3.2.1 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.

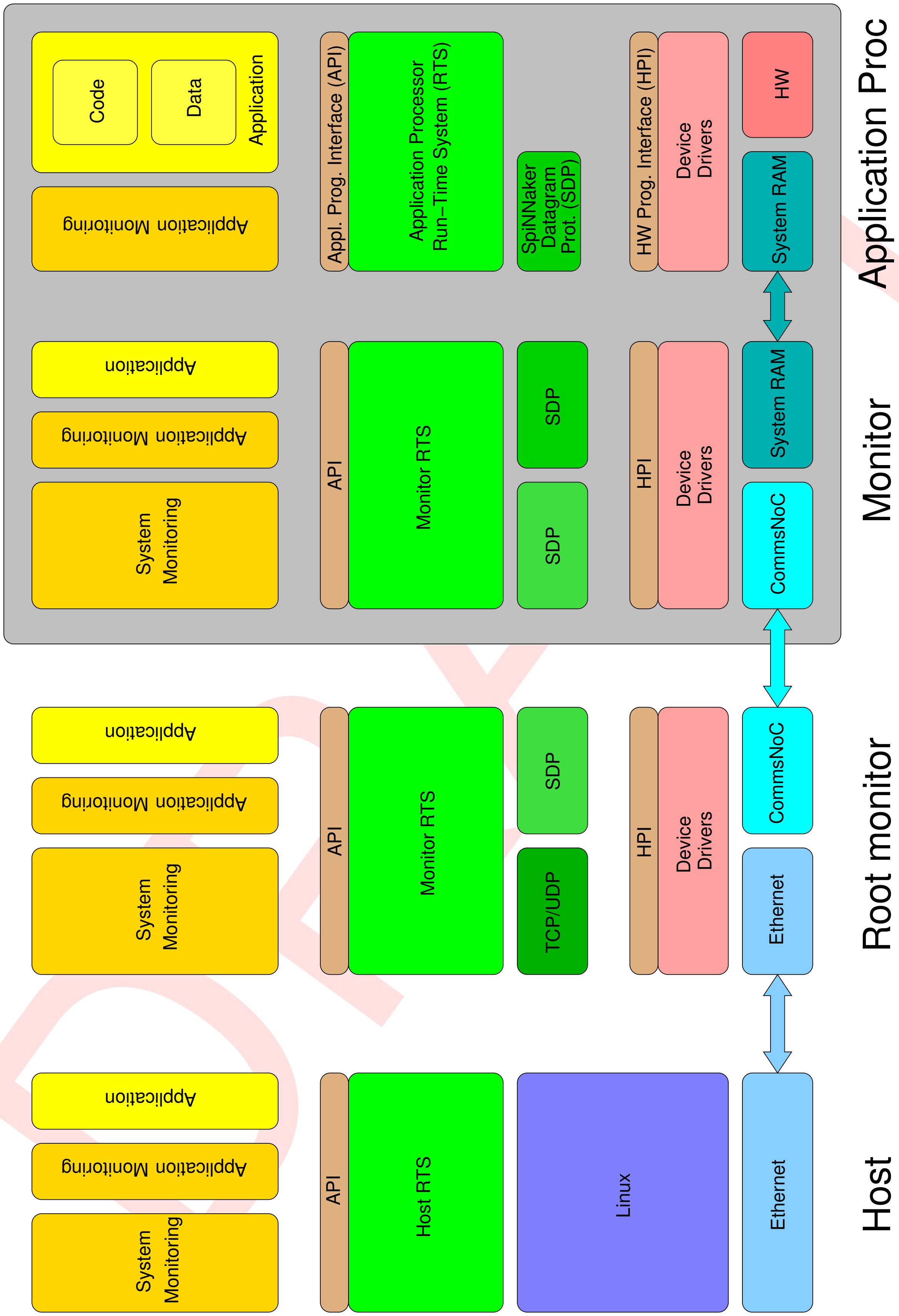

Figure 11: SpiNNaker run-time software stack.

3.2.2 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.

3.2.3 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).

3.2.4 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 Runtime memory map

Figure 12 shows the Application Processors run-time memory map.

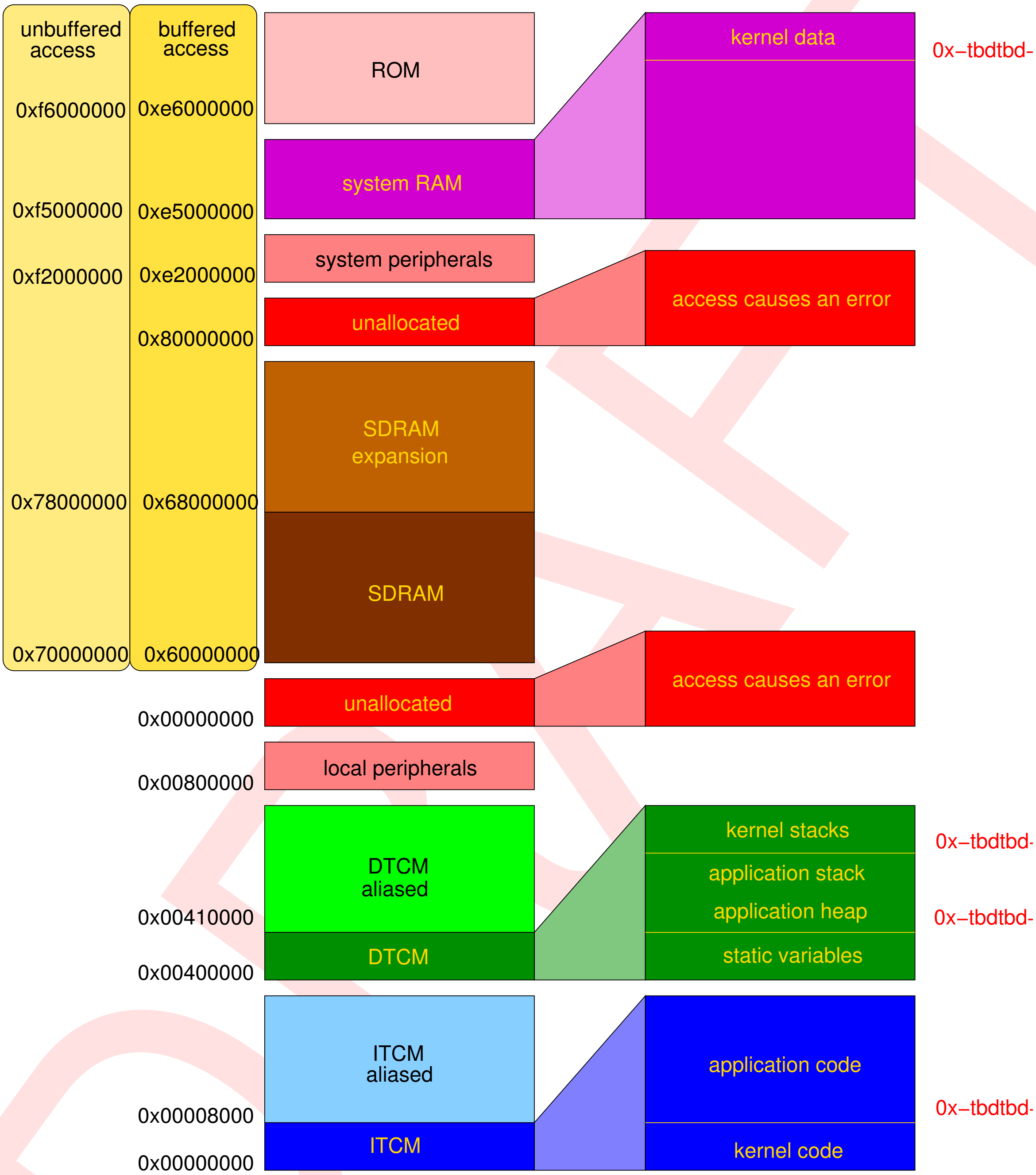

Figure 12: SpiNNaker run-time memory map.

4 Application programming interface (API)

4.1 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.

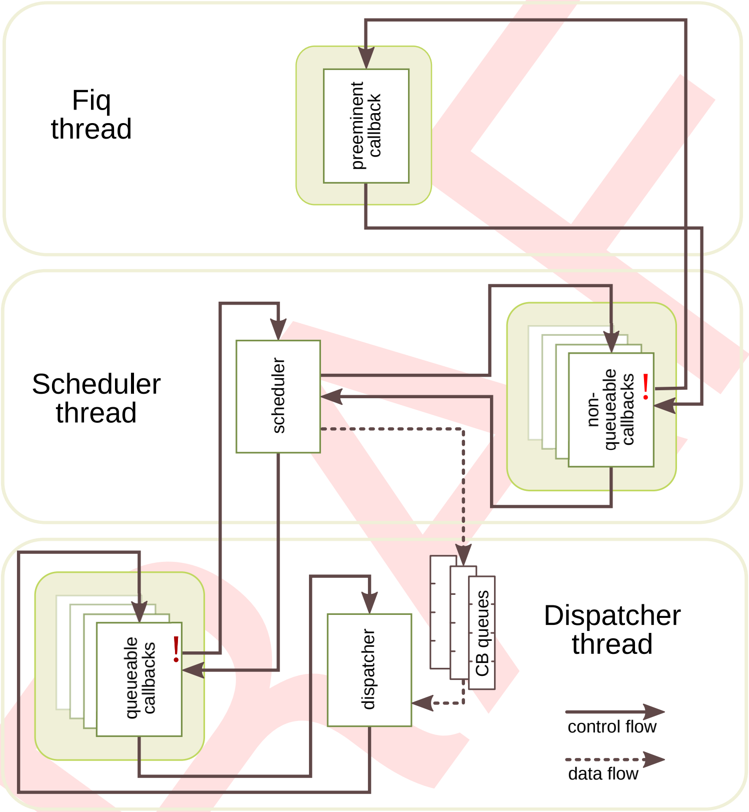

Figure 13: SpiNNaker event-driven programming framework.

Fig. 13 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.

4.1.1 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.

4.2 Programming interface

The following sections introduce the events and functions supported by the API.

4.2.1 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.

4.2.2 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

4.2.3 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

4.2.4 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;
```

4.2.5 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.	

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
};
```

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

4.2.6 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()
}
```

5 Neural net simulation frameworks

5.1 Spiking Neural net simulation framework

SpiNNaker applications are *event-driven* (figure 14) 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.

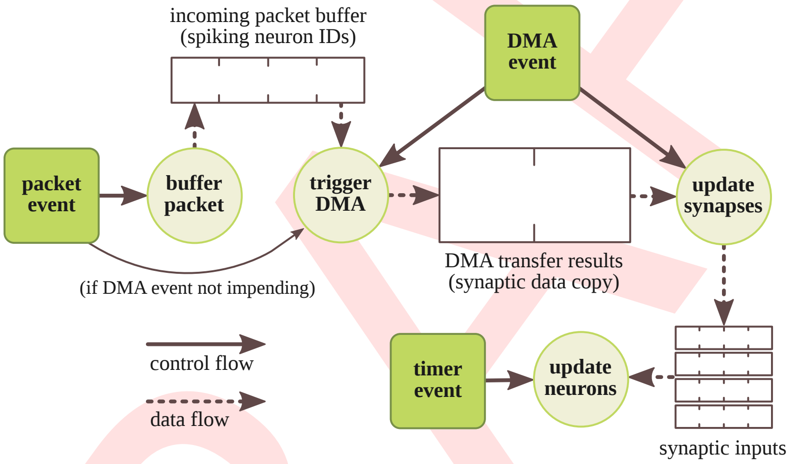

Figure 14: Events and corresponding tasks in a typical neural simulation.

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 15) 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 14, for example, a timer event schedules a neuron update task that is dispatched upon returning from the event.

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.

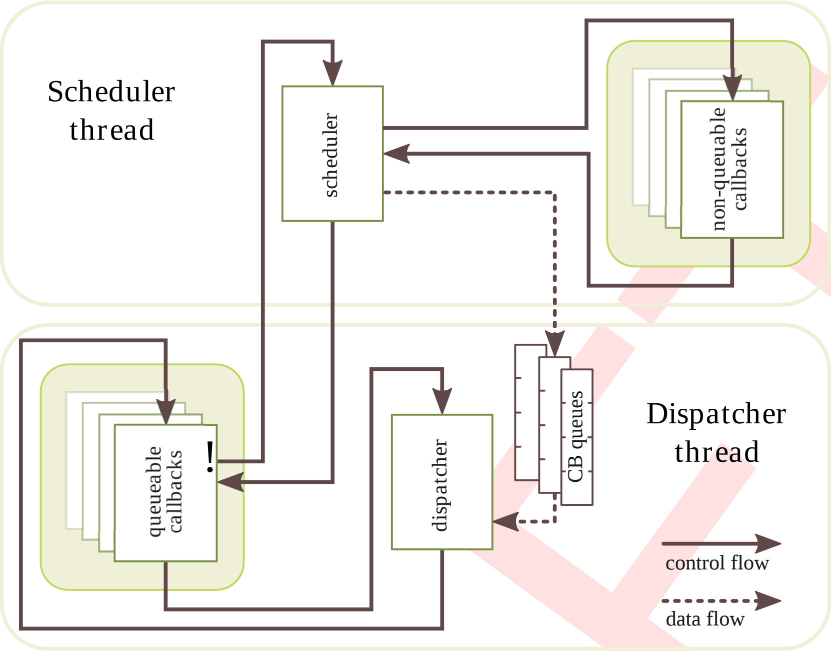

Figure 15: Control and data flow between the scheduler and dispatcher threads.

```
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);
    }
  }
}
```

5.2 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:

$$\Delta w_{ij} = \eta \delta_j O_i \tag{1}$$

$$\delta_j = \begin{cases} (C_j - O_j)\frac{dT_j}{dS_j} & \text{if j is an output layer} \\ \frac{dT_j}{dS_j}\sum_k \delta_j w_{jk} & \text{if j is not an output layer} \end{cases} \tag{2}$$

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. 16 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:

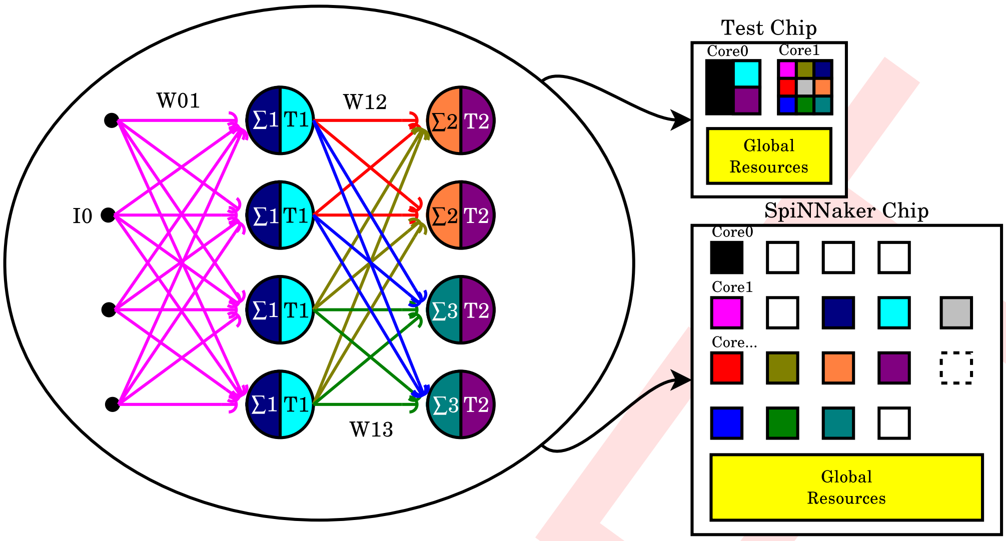

Figure 16: MLP network mapping.

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.

6 Neural net simulation development route

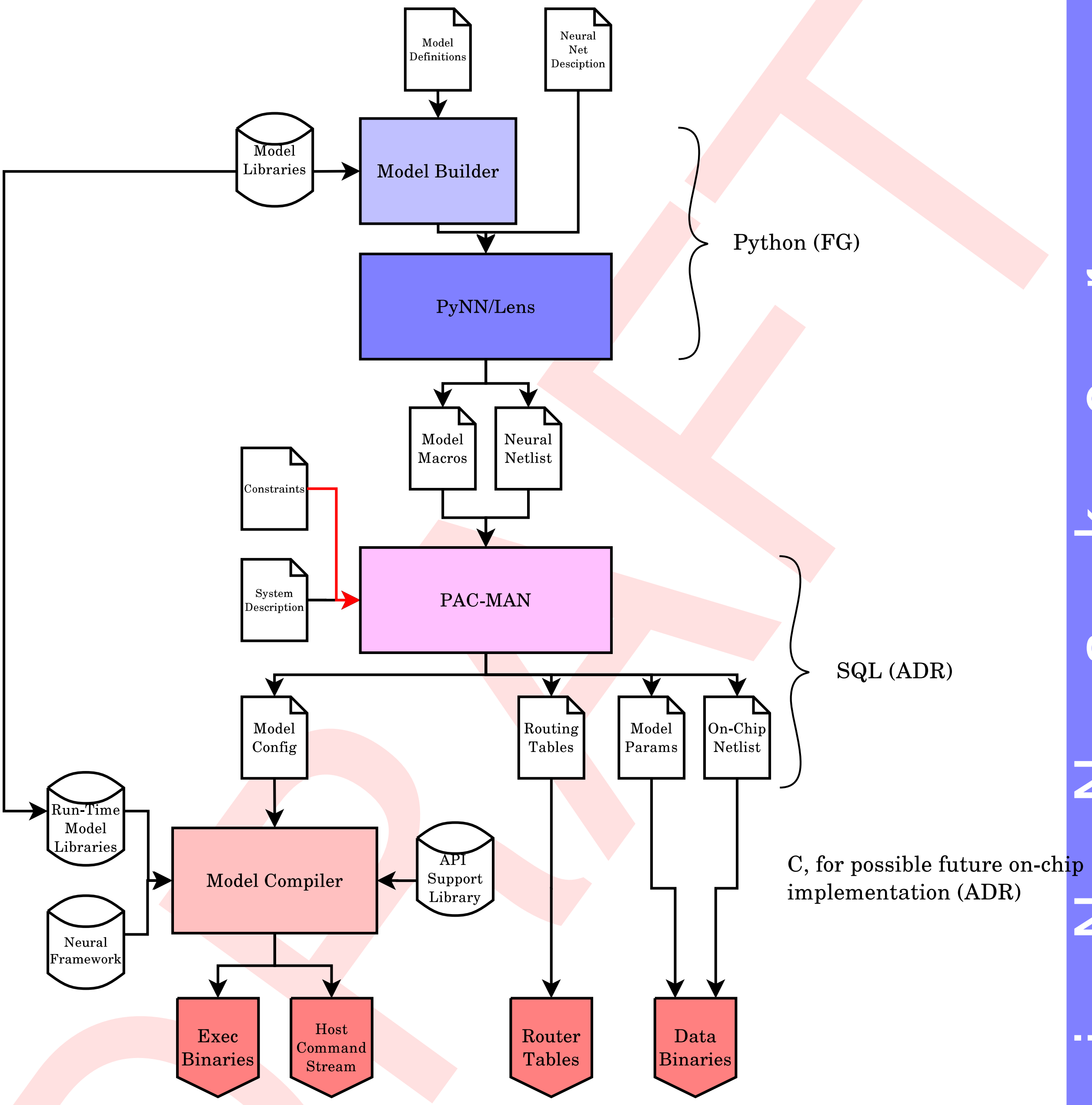

Figure 17: SpiNNaker neural net simulation development route.

6.1 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.

6.2 PyNN API functions list

Contents PyNN API version 0.7

6.3 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

6.4 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)

6.5 PopulationView

To be implemented, TBD how treat overlapping PopulationView

6.6 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

6.7 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)

6.8 Procedural interface for creating, connecting and recording networks

Not implemented as this is the low level Api.

6.9 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”.

6.10 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='stdpmecha
  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

6.11 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]'

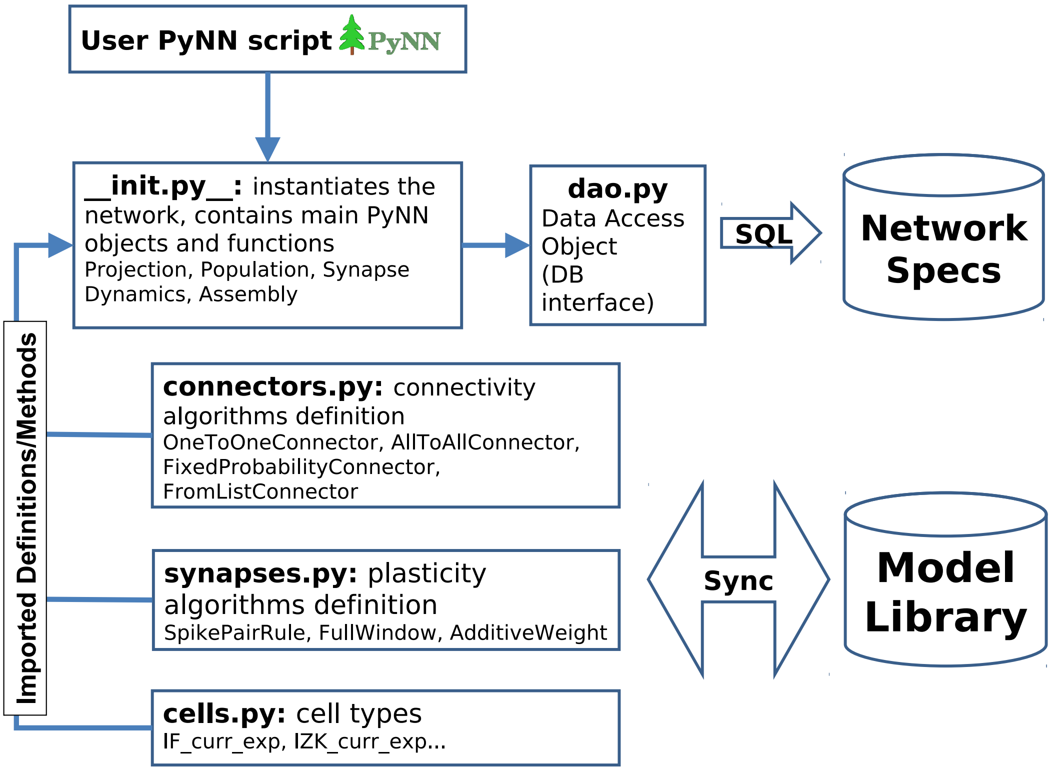
User PyNN script PyNN
__init.py__: instantiates the network, contains main PyNN objects and functions
Projection, Population, Synapse Dynamics, Assembly
dao.py
Data Access Object (DB interface)
SQL
Network Specs
Imported Definitions/Methods
connectors.py: connectivity algorithms definition
OneToOneConnector, AllToAllConnector, FixedProbabilityConnector, FromListConnector
synapses.py: plasticity algorithms definition
SpikePairRule, FullWindow, AdditiveWeight
Sync
Model Library
cells.py: cell types
IF_curr_exp, IZK_curr_exp...

Figure 18: PyNN/SpiNNaker interface structure.

7 Damson development route

7.1 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.

7.2 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.

7.3 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!

7.4 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.

7.5 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.

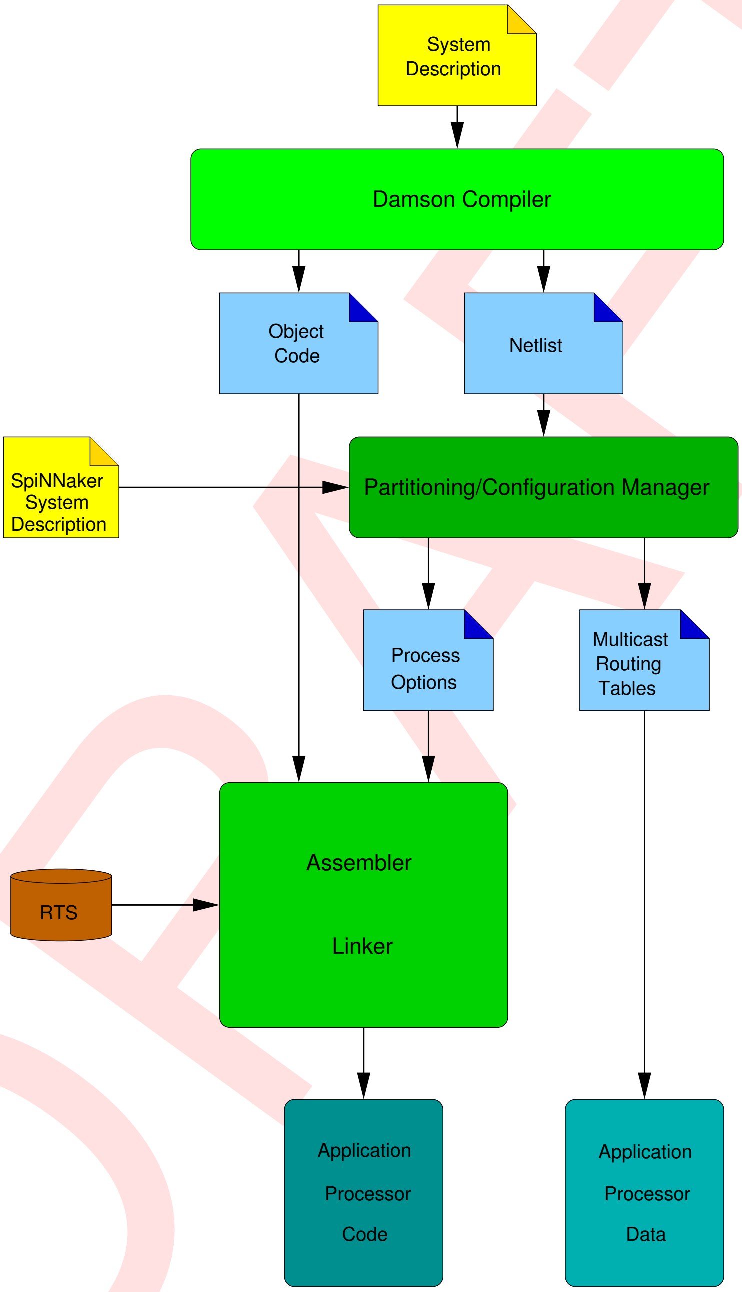
System Description
Damson Compiler
Object Code
Netlist
SpiNNaker System Description
Partitioning/Configuration Manager
Process Options
Multicast Routing Tables
RTS
Assembler
Linker
Application Processor Code
Application Processor Data

Figure 19: Damson development route.

8 PACMAN: partition and configuration manager

8.1 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.

PACMAN is based on a Database that holds three representations of the neural network (fig.21):

- **Model Level**: the network as specified in the high-level language (PyNN, Damson, LENS etc.)
- **PACMAN Level**: the network is partitioned in *proto* Populations that can be fit into a single computing core. Projections, probes and inputs are split accordingly. Proto Populations that can be fit into a core a grouped together
- **System Level**: a map linking groups of proto 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.20) 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 *ProtoPopulations* that will fit in a core (like a neural "place" operation)
- **Grouping**, responsible of collating ProtoPopulations 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. 22 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:

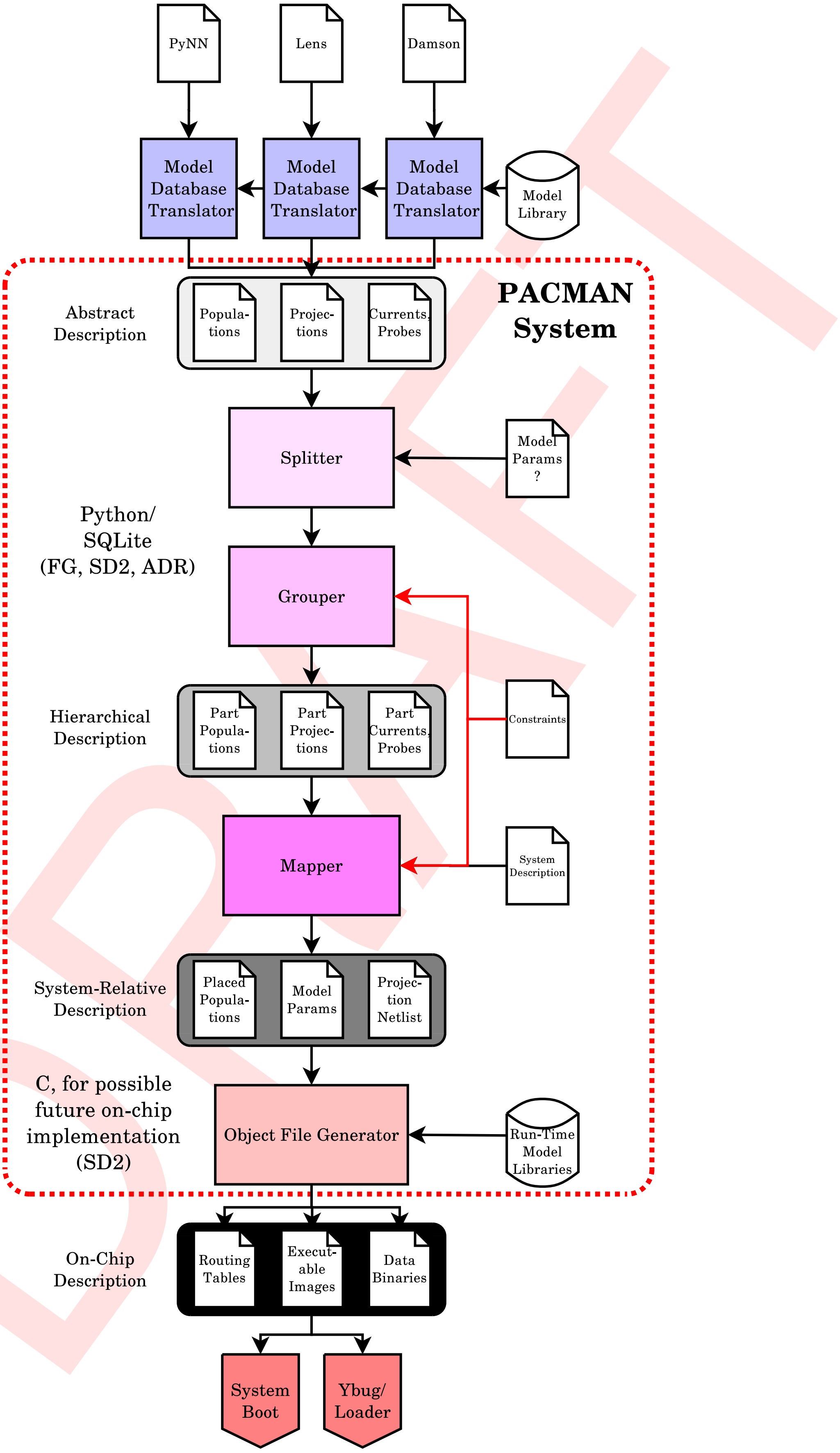

Figure 20: PACMAN internal structure

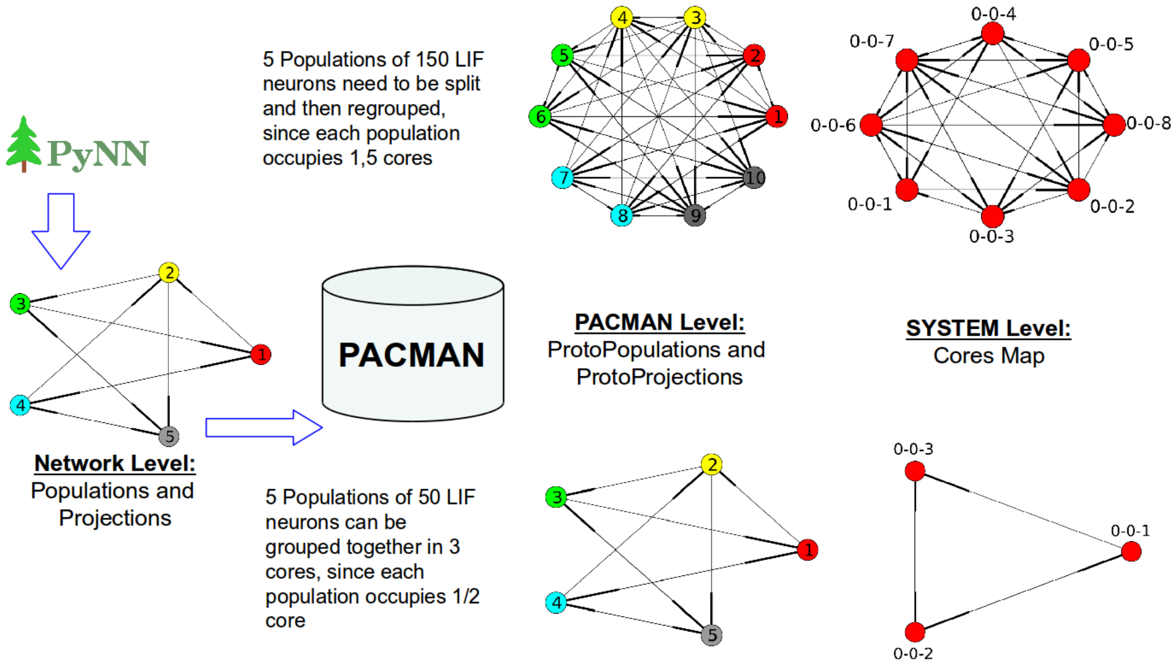
PyNN
5 Populations of 150 LIF neurons need to be split and then regrouped, since each population occupies 1,5 cores
Network Level:
Populations and Projections
PACMAN
PACMAN Level:
ProtoPopulations and ProtoProjections
SYSTEM Level:
Cores Map
5 Populations of 50 LIF neurons can be grouped together in 3 cores, since each population occupies 1/2 core
0-0-1
0-0-2
0-0-3
0-0-4
0-0-5
0-0-6
0-0-7
0-0-8

Figure 21: 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

- (top) if populations are too big to fit in a single core (150 neurons per population, top portion of the figure) they are splitted in 2 proto-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.

8.2 Splitting

During the Partitioning phase Populations that span more than one core will be divided into *ProtoPopulations* 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

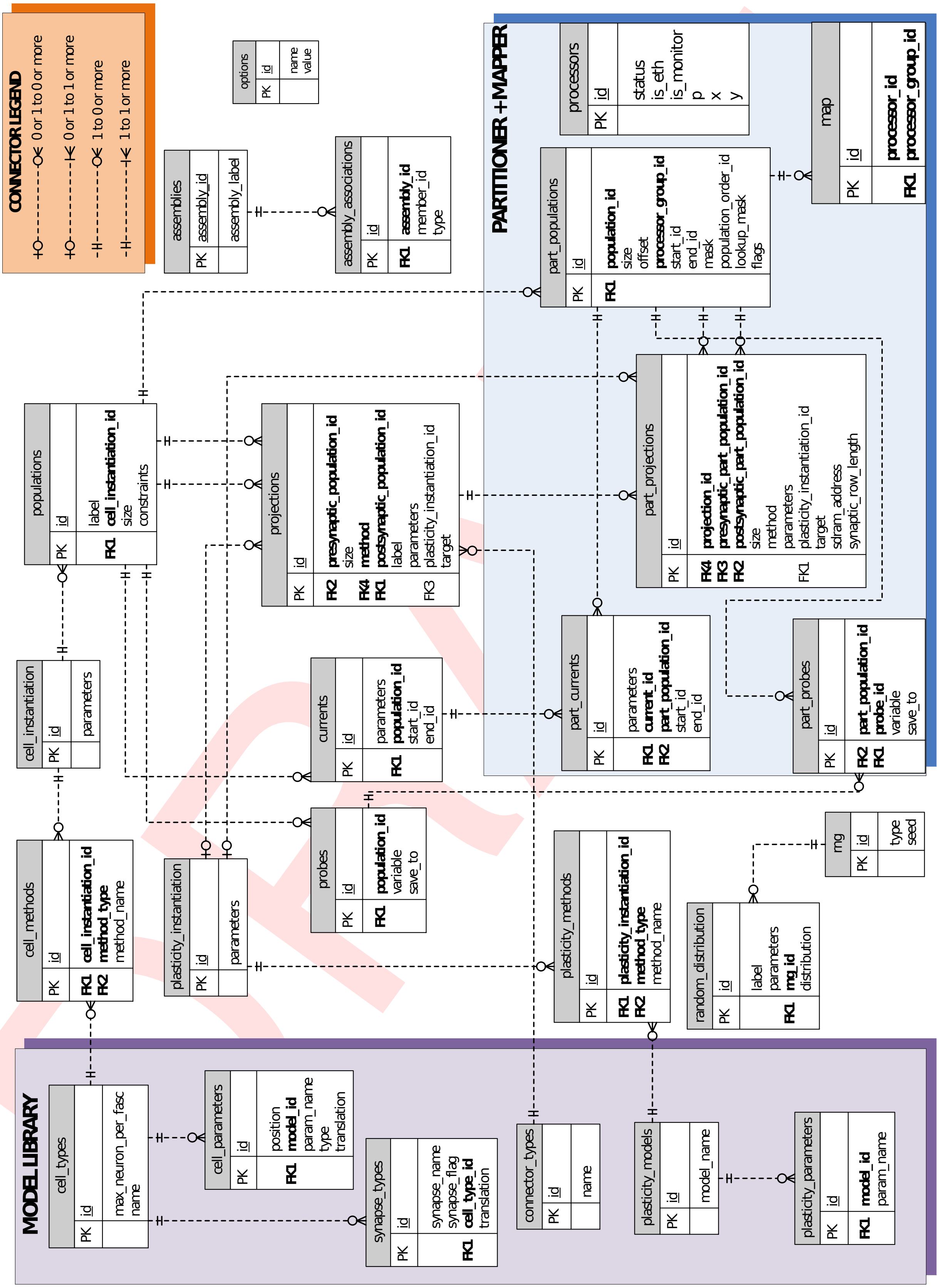

Figure 22: Database structure: model library and network specification

After having split Populations into ProtoPopulations Projections need to be exploded into ProtoProjections as well. A ProtoProjection is a Projection between 2 ProtoPopulations.

The output of the Partitioner will be stored in the proto_populations and proto_projections tables which have the following structure:

proto_populations		
Field	type	description
id (PRIMARY KEY)	INTEGER	ID defining a ProtoPopulation
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 ProtoPopulation
start_id	INTEGER	the core-relative starting id for the ProtoPopulation in the core
end_id	INTEGER	the core-relative ending id for the ProtoPopulation in the core
mask	INTEGER	mask defining Population and Neuron ID in the routing key
population_order_id	INTEGER	order of the ProtoPopulation 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. ProtoPopulations need to be ordered decreasingly according to their size

proto_populations		
Field	type	description
id (PRIMARY KEY)	INTEGER	ID defining a ProtoProjection
presynaptic_population_id (FOREIGN KEY: proto_populations.id)	INTEGER	ID defining the presynaptic ProtoPopulation in the ProtoProjection
postsynaptic_population_id (FOREIGN KEY: proto_populations.id)	INTEGER	ID defining the postsynaptic ProtoPopulation in the ProtoProjection
method (FOREIGN KEY: connector_types.id)	INTEGER	ID referring to the connector type between the 2 ProtoPopulations
size	INTEGER	number of single connections (neuron to neuron) in the ProtoProjection
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 ProtoProjection
label	TEXT	human readable label for ProtoProjection

notes: The ProtoProjection table has the same structure of the Projection table, but presynaptic_population_id and postsynaptic_population_id refer to ProtoPopulations rather than Populations. source is done for compatibility with PyNN

8.2.1 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 proto_populations. recalculates offsets for ids (FromListConnector)
- split_probes: splits Probes accordingly to proto_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)
```

8.3 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 ProtoPopulations, number of neurons in each ProtoPopulations, 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 proto_populations table, grouping different populations in the same group.

8.3.1 Implementation

The Grouper joins homogeneous proto_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)

8.4 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 *proto_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: proto_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

8.4.1 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.

8.5 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

8.6 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.

8.7 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.

9 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.

9.1 All languages

Comments: A Robodoc comment (see section 10) 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...?*

9.2 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;
        ...
    }

    ...
}
```

9.3 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
```

9.4 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/

10 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.

10.1 C / C++

The following templates are for C/C++ source code and header files

10.1.1 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: file:///mnt/peveril/home/amu4/spinnaker/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 $

10.1.2 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.

10.1.3 Structure documentation template

```
/****s* filename/structureName
*
* SUMMARY
*  abstract
*
* FIELDS
*  variable: description
*
* SOURCE
*/
```

```
STRUCTURE CODE
```

```
/*
*******/
```

“STRUCTURE CODE” indicates where the function should be written.

10.2 Assembly language

The following templates are for assembly language source code files

10.2.1 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: file:///mnt/peveril/home/amu4/spinnaker/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:

10.2.2 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.

10.3 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 - Spiking Neural Network Simulator 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
```