FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of a Dynamically Extensible SpiNNaker Chip Computing Module

Rui Emanuel Gonçalves Calado Araújo

Master in Electrical and Computers Engineering

Supervisor: Jörg Conradt

Co-Supervisor: Diamantino Freitas

January 27, 2014

Resumo

O projeto SpiNNaker desenvolveu uma arquitetura que é capaz de criar um sistema com mais de um milhão de núcleos, com o objetivo de simular mais de um bilhão de neurónios em tempo real biológico. O núcleo deste sistema é o "chip" SpiNNaker, um multiprocessador System-on-Chip com um elevado nível de interligação entre as suas unidades de processamento. Apesar de ser uma plataforma de computação com muito potencial, até para aplicações genéricas, atualmente é apenas disponibilizada em configurações fixas e requer uma estação de trabalho, como uma máquina tipo "desktop" ou "laptop" conectada através de uma conexão Ethernet, para a sua inicialização e receber o programa e os dados a processar.

No sentido de tirar proveito das capacidades do "chip" SpiNNaker noutras áreas, como por exemplo, na área da robótica, nomeadamente no caso de robots voadores ou de tamanho pequeno, uma nova solução de hardware com software configurável tem de ser projetada de forma a poder selecionar granularmente a quantidade do poder de processamento. Estas novas capacidades permitem que a arquitetura SpiNNaker possa ser utilizada em mais aplicações para além daquelas para que foi originalmente projetada.

Esta dissertação apresenta um módulo de computação dinamicamente extensível baseado em "chips" SpiNNaker com a finalidade de ultrapassar as limitações supracitadas das máquinas SpiNNaker atualmente disponíveis. Esta solução consiste numa única placa com um microcontrolador, que emula um "chip" SpiNNaker com uma ligação Ethernet, acessível através de uma porta série e com um "chip" SpiNNaker. Além disso, um programa de computador multi-plataforma baseado em Java disfarça esta solução personalizada permitindo que este novo sistema seja retrocompatível com todas as ferramentas existentes para as máquinas convencionais.

Para desenvolver esta nova solução foi necessário um estudo profundo e detalhado da arquitetura e do funcionamento interno do "chip" SpiNNaker. Com o conhecimento adquirido a partir desta análise aprofundada, foi possível construir uma placa com um microcontrolador e um "chip" SpiNNaker conectados através dum protocolo interno da arquitetura SpiNNaker. Após a criação da placa, desenvolveu-se o software para controlar o microcontrolador e uma aplicação para manter a retrocompatibilidade com sistemas padrão. Para demonstrar as novas capacidades desta plataforma noutras aplicações para além de redes neuronais, desenvolveu-se uma simulação do movimento dum bando de pássaros.

A solução apresentada permite a utilização de dezenas de núcleos ARM eficientes num pacote de reduzida dimensão, sendo desta forma adequada para ser utilizada em pequenos robots e assim implementar algoritmos avançados que exijam poder computacional paralelo.

Abstract

The SpiNNaker project has created an architecture that is capable of scaling up to a system with more than a million embedded cores in order to simulate more than one billion spiking neurons in biological real time. The heart of this system is the SpiNNaker chip, a Multi-Processor System-on-Chip with a high level of interconnectivity between its processing units. Although it is a very powerful computing platform, even for non-neural application, it is currently only available in fixed configurations and it requires a workstation, usually a desktop or a laptop connected through an Ethernet connection, to be initialised and to receive the data to be processed.

Therefore if one wishes to take advantage of the capabilities of the SpiNNaker chip in other fields, as for example the robotics field specially in the case of small or flying robots, a new hardware solution with custom software must be built where the amount of processing power can be granularly selected. This new capability allows the SpiNNaker architecture to be used in more applications than it was originally designed for.

This thesis presents a Dynamically Extensible SpiNNaker Chip Computing Module to improve on the limitations of the currently available SpiNNaker machines. This approach is a single board with a microcontroller which emulates an Ethernet connected SpiNNaker chip accessible through a serial port and a single SpiNNaker chip placed together. Additionally it features a cross-platform Java based computer program that disguises this custom solution allowing this new system to be backwards compatible with all the existing tools for the standard machines.

In order to develop this new solution, a very deep and detailed study of the inner workings of the SpiNNaker chip was required. With the acquired knowledge from this thorough analysis, it was then possible to build a custom board with a microcontroller and the SpiNNaker connected through a internal protocol from the SpiNNaker architecture and develop the software to drive the microcontroller and an application to emulate a standard system for the SpiNNaker tools. A non-neural application, the simulation of the movement of a flock of birds, was developed to demonstrate the general purpose capabilities that this new platform has.

The presented solution allows the deployment of dozens of power efficient ARM cores available in a very small package suitable to be used in small robots which makes it possible to implement advanced algorithms that require truly parallel computational power.

Acknowledgments

I wish to thank, first and foremost, to Professor Jörg Conradt who continuously supported and encouraged me, besides the technical advice essential for the development of this work. I am also thankful to my co-supervisor Professor Diamantino Freitas for his availability, support and criticism. Besides my supervisors, I would like to thank Nicolai Waniek and Christian Denk for their precious suggestions and feedback.

This thesis would not have been possible without the precious help from Steve Temple, Luis Plana and Francesco Gallupi from the University of Manchester who provided essential guidance while studying the SpiNNaker system.

I thank my fellow student, Tobias Brennich, whose help in the assembly of the multiple testing and final boards proved to be invaluable.

I owe my deepest gratitude to my family, especially my parents, Rui and Manuela, and my siblings, Mónica, Beatriz and Francisco, for their unconditional love and support.

I reserve a special thank you, to my soul mate, Ângela Igreja for all your love, affection,encouragement, motivation, support and availability, along this path. I would also like to thank all of my friends, especially to Ricardo Castro, for all their encouragement and support .

Rui Araújo

"If you are thinking of the brain as a computer,
the neuron is the transistor."

Carl Schoonover

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

API	Application Programmable Interface
APLX	Application Load and Execute
APT	Advanced Processor Technologies
ASCII	American Standard Code for Information Interchange
BGA	Ball Grid Array
BSD	Berkeley Software Distribution
CCR	Cyclic redundancy check
CPU	Central Processing Unit
DDR	Double Data Rate
DMA	Direct Memory Access
DMIPS	Dhrystone Millions Instructions per Second
DTCM	Data Tightly-Coupled Memory
EAGLE	Easily Applicable Graphical Layout Editor
EDA	Electronic Design Automation
ELF	Executable and Linkable Format
EEPROM	Electrically Erasable Programmable Read-Only Memory
FIQ	Fast Interrupt Request
FLOPS	Floating-point Operations Per Second
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
FR	Fixed Route
GALS	Globally Asynchronous Locally Synchronous
GPIO	General Purpose Input Output
GPL	General Public License
IRQ	Interrupt Request
JAR	Java Archive
LED	Light-emitting diode
IPC	Interprocessor Communication
ITCM	Instruction Tightly-Coupled Memory
MAC	Media Access Control
MC	Multicast
MIPS	Millions Instructions per Second
MPSoC	Multi-Processor System-on-Chip
NN	Nearest-neighbour
NoC	Network on Chip
NRZ	Non-Return-To-Zero
OSI	Open Systems Interconnection
P2P	Point to Point

PCB	Printed Circuit Board
RAM	Random Access Memory
ROM	Read Only Memory
RTZ	Return To Zero
SARK	SpiNNaker Application Runtime Kernel
SATA	Serial Advanced Technology Attachment
SCP	SpiNNaker Command Protocol
SC&MP	SpiNNaker Control & Monitor Program
SDP	SpiNNaker Datagram Protocol
SDRAM	Synchronous dynamic random access memory
SoC	System on Chip
SpiNNaker	SPIKING Neural Network Architecture
SRAM	Static Random Access Memory
TCAM	Ternary Content-addressable Memory
UDP	User Datagram Protocol
VIC	Vector Interrupt Controller
VLSI	Very-large-scale integration

Chapter 1

Introduction

The inner workings of the biological brain is still one of the great challenges for computational neuroscience despite an increasing amount of experimental data and deeper scientific understanding of individual components such as neurons.

There is a general consensus that the human brain has roughly about 85 to 100 billion neurons total [Ngu10] where each neuron can have up to 15000 connections to other neurons via synapses [Bro09]. Using techniques such as magnetic resonance imaging it is possible to observe large-scale brain activity. However, this knowledge is insufficient to truly understand how thoughts are constructed and how information is generally processed. It is believed that these functions probably lie in the intermediate levels of the brain [FB09]. In order to understand these middle layers, it is necessary to construct very large systems of spiking neurons with structures inspired by the latest findings from the neuroscience field.

General purpose digital architectures are not well suited to simulate these kinds of networks since these networks are characterized by massive processing parallelism and a high level of interconnectivity between the processing units. A possible approach is the usage of neuromorphic systems such as the BrainScales [PGJ+12] or the Neuro-grid [CSF+12] which *emulate* the neural network with a physical implementation of the individuals neurons. Another possible approach is a massively-parallel computer architecture with a high bandwidth inter-process communication like the SpiNNaker system [FB09].

1.1 Context

The SpiNNaker system is a massively-parallel computer architecture based on a Multi-Processor System-on-Chip (MPSoC) technology that can scale up to a million cores and is capable of simulating up to a billion spiking neurons in biological real time with realistic levels of interconnectivity between the neurons.

The SpiNNaker system was designed under the latest paradigm for high-performance computing, highly-parallel systems. However, it is motivated by the attempt to understand and study biological computing structures which achieve high level of parallelism with frugal amounts of

energy as opposed to traditional electronics designs which up until the last few years were mostly driven by the serial throughput. The biological approach to the design of this *many-cores* architecture also brings new concerns in terms of fault-tolerance computation and efficiency. The SpiNNaker chip, the basic building block of a SPiNNaker machine, relies on smaller processors than other machines but in greater number, it has 18 highly efficient embedded ARM processors that allows the SpiNNaker system to be competitive according to two metrics, MIPS/mm^2 and MIPS/W.

1.2 Motivation

The currently available machines with SpiNNaker chips are relatively large, the minimum size at this moment is 105×95mm, which limits their deployment on systems with limited size as for example, small mobile robots, specially flying ones due to very strict weight and space constraints. Additionally the SpiNNaker systems currently require a workstation, usually a desktop or a laptop, connected through an Ethernet connection to bootstrap the system every time it powers on and to feed the processing data into the system. This requirement seriously limits the independence and deployment capabilities of systems with embedded SpiNNaker chips. At present, it is necessary to add an wireless router in order to have a mobile system with a SpiNNaker machine [DLBG+13]. The drawbacks from this approach are fairly obvious, such as increased power consumption and space requirements since the typical wireless router consumes around 4 to 5 Watt and even though there are fairly small models available at the market it would still take up some space.

Furthermore the amount of extensibility provided by standard SpiNNaker machines is very limited since it only allows increases of computing power in fixed amounts. The current single board SpiNNaker machines are available in two versions, one with four chips and another with forty eight. These are wildly different amounts of processing capability which make it difficult to create intermediate solutions. It would be interesting to have the capability of selecting how many SpiNNaker chips one needs to deploy without having to design new hardware.

The current requirements of the SpiNNaker architecture are not suitable for a lot of applications where its processing power and capabilities would be helpful. It is then necessary to design a new solution that can overcome the limitations of the present options.

1.3 Goals

This thesis focused on the detailed study of the SpiNNaker architecture, with special care given to the analysis of the inner workings of the SpiNNaker chips during its bootstrapping procedures and its communications protocols with other chips in the system. The main objective from this study is the identification of the most relevant limitations and possible paths to overcome these.

The primary goal of this research is to bring this advanced and efficient computing platform to new ventures with increased flexibility. Having identified the main flaws and possible solutions to overcome them, the most significant requirements can be determined. These are as follows:

- **Small size** – in order for this system to be available for deployment in as many fields as possible, it must have reduced dimensions or allow for significant reductions in size with future iterations. This requirement makes it feasible to include this new computing platform in many more applications;
- **Extensible** – the basic solution should have only one SpiNNaker chip but it must allow the system to be extended in order to be a compelling option for applications that require greater amounts of computing power;
- **Low cost** – it should strive to use low cost components and only the ones required for the solution;
- **Backwards Compatibility** – the new solution must be compatible with tools and frameworks that are currently available for the SpiNNaker machines. The requirement increases the value of the new solution by allowing previous developments based on standard SpiNNaker machines to be used on the new system with minimal or no work required.

1.4 Methodology

The methodology devised for this research to reach the defined goals includes the following phases, as depicted in Figure 1.1.

1. **Information Gathering:** The initial phase of the research was dedicated to gather and synthesize information on the SpiNNaker system and specially on the detailed behaviour of the SpiNNaker chip and on its communication with other chips. The results of this research are presented in chapter 2.
2. **Development Approach:**

   The solution development was done in stages. The initial step was the design of a basic hardware version in order to allow the start the development of the software while still iterating on the final hardware design. The second stage was the continued improvement of the software up until the point it could boot SpiNNaker system by itself. The final step was the competition of the software to be feature complete in terms of the various capabilities that the communication subsystem of the SpiNNaker system has. This style of development led to successive improvements on the previous layers when new features or changes of functionality were required.
3. **Evaluation Approach**:

   In order to do a validation of the developed solution, besides the standard test-bench measurements, a case study of a computation-intensive process with special random characteristics was built. The simulation of the movement of a flock of birds [Rey87] was implemented on a desktop computer and on the new system to prove the gains that a SpiNNaker system

may bring. This example was tested on the developed solution as well as on a standard SpiNNaker machine to validate the backwards compatibility requirement.

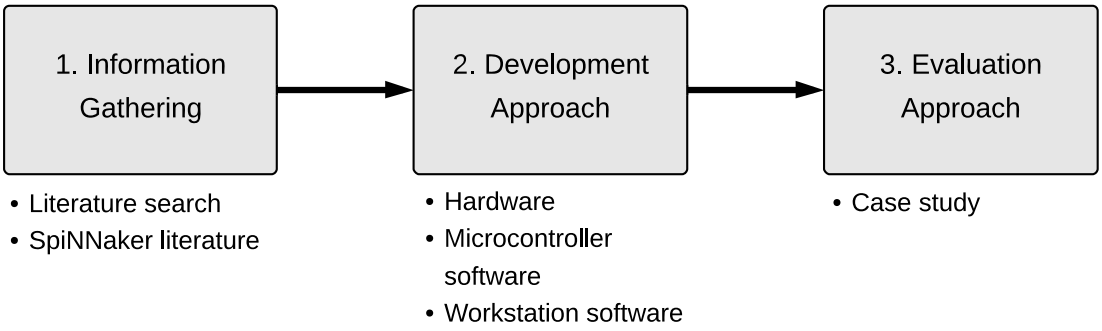


Figure 1.1: Work Methodology: phases and respective research methods

1.5 Main Results

The main contribution of the research is a new high performance computing module based on the SpiNNaker chip which is small and cost efficient to be used on mobile platforms such as small robots. The main features of the developed module are:

- **Small size** – Removing the Ethernet jack among other non-essential components allowed significant reductions in total size of the new board even though the current proof of concept is a only one sided. It features also a single SpiNNaker chip and a microcontroller that emulates an Ehternet connected SpiNNaker chip.
- **Extensible** – The developed board has a 34 pin connector that allows the system to be augmented with other SpiNNaker machines or a custom single SpiNNaker board.
- **Backwards Compatibility** – A computer application written in Java was developed that allowed the system to masquerade as a standard SpiNNaker machine, capturing the UDP datagram used by the standard tools.

Another contribution from the developed work during this thesis, is the detailed study presented in chapter 2 on the innards of the SpiNNaker architecture. It is a collection of the knowledge present in the several articles and documents available from the researchers who created this biological inspired computing platform.

1.6 Related Works

The SpiNNaker system, although biological inspired and designed to help the study of the behaviour of large neural network of spiking neurons, is still a fairly general purpose architecture

since it makes use of small ARM embedded cores which are general purposes processors. There is an alternative named neuromorphic system which are a class of devices that implements particular features of biological neural networks in their physical circuit layout [CSBR10]. One of these type of systems has been designed by the BrainScaleS project, a universal neuromorphic computing substrate [PGJ+12], whose central component is the neuromorphic microchip Spikey. This microchip contains analog very-large-scale integration (VLSI) circuits that model the electric behaviour of neurons and synapses. In this type of modelling, measurable quantities in the circuit have biological equivalents as for instance, the membrane potential V_m of a neuron is modelled by the voltage over a capacitor C_m. On Spikey, the standard leaky integrate-and-fire (LIF) neuron model with conductance-based synapses, depicted in equation 1.1, is implemented.

$$C_m \frac{dV_M}{dt} = -g_1(V_m - E_1) - \sum_i g_i(V_m - E_i) \tag{1.1}$$

This neuromorphic system includes a Field Programmable Gate Array (FPGA) to interface the system with a Host computer that is responsible for generating configuration data as well as input stimuli to the network. It was not possible to find the power consumption of this system but depending on the selected FPGA it should be lower than the comparable SpiNNaker system since the power consumption of neuromorphic chips is much lower than equivalent digital designs [ID00].

Another neuromorphic system, although this one is a multi-chip solution, is the *Neurogird* [CSF+12] which is capable of simulating a million neurons connected by billions of synapses in real-time. It has sixteen 12×14mm Neurocores where the layers of the neural networks are mapped and SRAMs and a FPGA to relay the packets with the spikes. The power consumption for the entire system is 3.1W which is a very low number for the number of neurons *emulated*. For comparison purposes, a comparable SpiNNaker system would consume 90W.

The biggest advantage from the neuromorphic systems is the low power consumption specially when compared to regular digital circuits. On the other hand, these systems are very tailored to the original models they were designed for and although they have some reconfigurability, it is still very limited when compared with a more general purpose approach like the SpiNNaker architecture.

There are other general purposes solutions which have designed as massively parallel high-performance computing, as for instance, the IBM BlueGene/Q Compute Chip [HOF+12] which is the basic unit of computing for the Blue Gene Project [GBC+05] which is an IBM project aimed at designing supercomputers that can reach operating speeds in the petaFLOPS (for FLoating-point Operations Per Second) range, with low power consumption. This is the third design generation after BlueGene/L and BlueGene/P. The compute chip design shares some designs decision with the SpiNNaker chip. It features 18 processing units, with one used as spare and another in charge of management tasks, although the cores used are a variant from the PowerPC A2 [IBM12] down-clocked to 1.6 GHzfrom the original 2.3 GHz design speed as opposed to the embedded ARM cores used in the SpiNNaker architecture. Nevertheless, the main objective of this chip is to have

maximum throughput, as opposed to the SpiNNaker design where the design guidelines lead to some performance sacrifice for increased efficiency. The rated power consumption for the BlueGene/Q Compute Chip is 55 W while delivering a peak performance of 204.8 gigaFLOPS. It was not possible to find a Dhrystone performance measurement for this chip in order to compare it with the SpiNNaker architecture which achieves 1.1 DMIPS/MHz.

1.7 Document Structure

This dissertation is organized in four more chapters, besides the current one.

Chapter 2 describes the SpiNNaker chip in detail, starting by the hardware perspective and later analysing the several layers of software that are needed on a standard SpiNNaker machine.

Chapter 3 presents the developed solution, the SpiNNaker Chip Computing Module while the chapter 4 describes the evaluation.

Lastly, chapter 5 reviews the dissertation as a whole, explains results and points questions for future research, that might improve the proposed solution. Appendix A displays the developed hardware, with the schematics and the PCB layouts for all the developed boards, and appendix B has a detailed diagram with the format of the packets used in a SpiNNaker specific protocol. Appendix C has some details on the code organization and on its availability and appendix D has the available API from the kernel that runs on the Application processors.

Chapter 2

The SpiNNaker System

The SpiNNaker chip is the basic building block of the SpiNNaker system. This system was designed with the aim of simulating up to a billion spiking neurons in (biological) real-time [NLMA+09]. This system is intended to serve as the *brain* of mobile robots to provide real-time stimulus-response behaviour [ES03] and to help improve the understanding of the brain architecture. The system was biologically inspired which allowed it to take advantage of several characteristics such as the lack of memory coherence and the slow pace of biological neurons when compared to artificial ones together with regular neuron losses. In fact, the average human adult loses about a neuron per second without any visible consequences [FB09] meaning that the SpiNNaker machine must be resilient to failures, since the sheer scale of the project will lead to frequent problems from which the system must recover and work around without the need for manual intervention.

This chapter presents the SpiNNaker in detail since it is a state-of-art biologically inspired system that it is not available for sale to the general public.

2.1 Architecture

The SpiNNaker machine has massively parallel architecture which can hold up to 65536 nodes, where each node is a SpiNNaker chip, a System on Chip (SoC) device with 18 low power ARM968 processors and a common 128 M byte SDRAM. One of the most important guidelines while the SoC was designed was low energy consumption since it was assumed that the cost of processors can be considered negligible when compared to the cost of the energy for the duration of the system lifetime. These guidelines explain the use of efficient embedded ARM9 cores and Mobile DDR (Double Data Rate) SDRAM where some performance was sacrificed in exchange of for lower power consumption. For inter-chip communications, self-timed channels were used which require much less power than synchronous links of the same capacity although they are more costly in wiring.

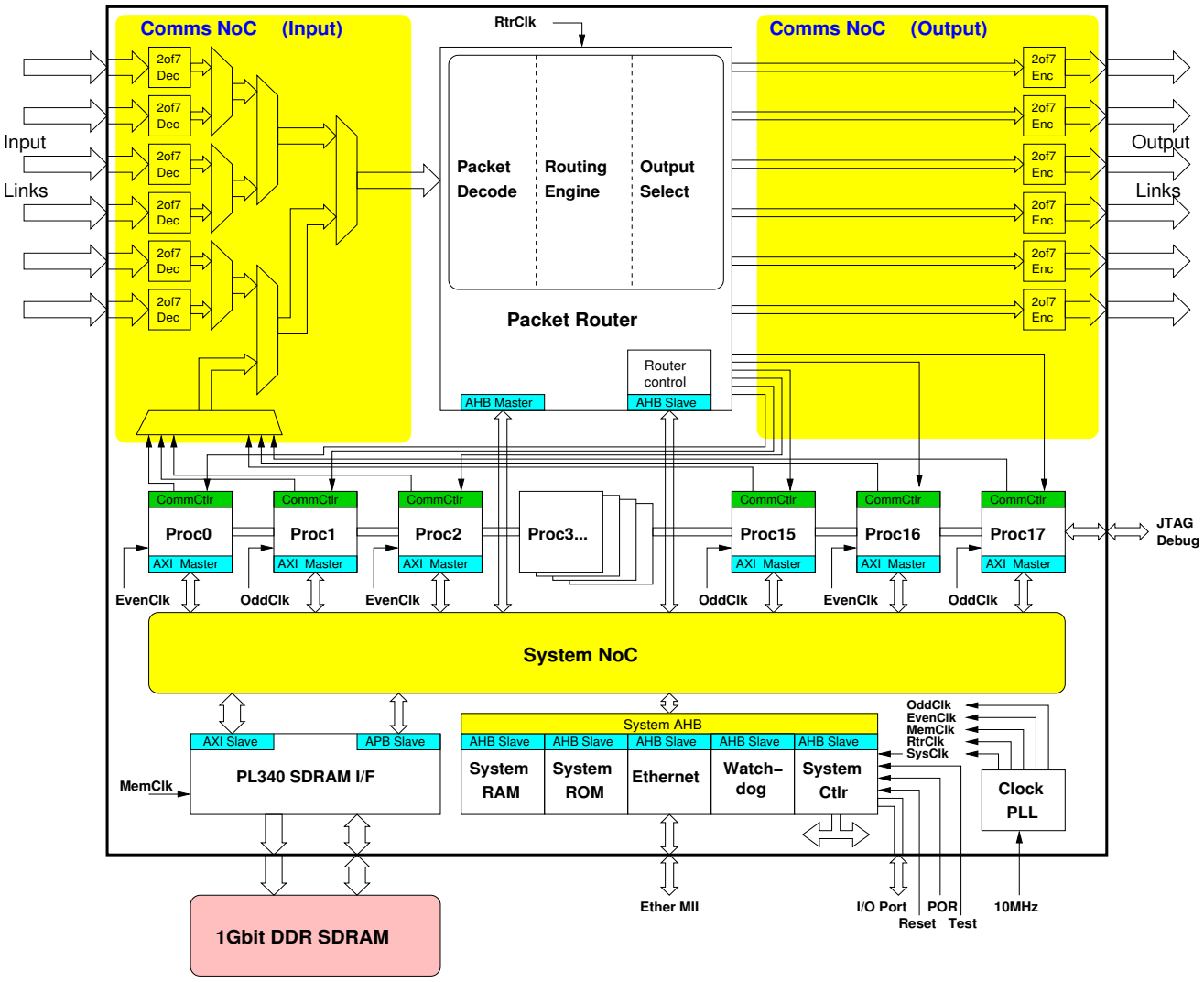


Figure 2.1: Block diagram of the SpiNNaker chip [Gro11b].

The chip itself is a Globally Asynchronous Locally Synchronous (GALS) system with the mentioned 18 low power ARM968 processors nodes connected through an energy-efficient packet-switched asynchronous communications infrastructure. Initial SoC designs used an interconnect paradigm based on a shared bus design [FB05]. However this paradigm is not conveniently up-scalable when the complexity of the system increases. In order to solve this problem, SoC designers use a complex hierarchy of buses which allows concurrent communication with the different components partitioned in separate buses. These buses are connected through complex protocols and multiple bridges between them. This increased complexity makes it harder to meet the timing requirements. A solution to this problem is the use of packet-switched networks [DT01] which offer greater flexibility in the topology of the SoC, reduced latency and increased bandwidth through the use of additional area. Networks on Chip (NoC) decouple the timing domains of each block which simplifies the timing closure process. The SpiNNaker chip uses CHAIN, a solution developed at the University of Manchester, which uses self-timed circuits with delay insensitive data

encoding combined with a return-to-zero signalling protocol to implement the packet switching network [BF02]. Self-timed circuits (asynchronous) are an alternative circuit design which uses acknowledgement to explicitly indicate and validate the data as opposed to synchronous circuits where there is a global clock to indicate the moments of data stability. There are a number of different designs in how to signal data validity, CHAIN uses a delay-insensitive style [Ver88] where the data validity is transmitted implicitly in the data encoding which removes the need for much of the timing analysis since this design operates correctly regardless of the delays involved in the interconnected wires.

Figure 2.1 illustrates the main functional components. The shaded areas indicate the asynchronous interconnect areas. It is clear that each chip has two NoC. The Communications NoC is responsible for transmitting packets between on-chip and off-chip processors while the System NoC replaces the traditional system bus by providing access to an off-chip DDR SDRAM, which is usually available in the same package mounted on top of the SpiNNaker die and stitch-bonded to it, and to other system control peripherals like the Router's configuration registers, the System Controller and the Watchdog Timer.

One of the most important components is the ARM928 core which is the main processing resource of the SpiNNaker system and whose block diagram can be seen in Figure 2.2. The ARM698E-S is the smallest, lowest power consuming ARM9 family processor [ARM13]. Each core in the SpiNNaker chip was configured to have 32 Kbyte of instruction tightly-coupled memory (ITCM) and 64 Kbyte of data tightly-coupled memory (DTCM) available. It also has a timer which is used by the kernel to provide time sensitive services, a DMA controller to be used for transfers with the attached SDRAM and to communicate with the system bus, and a Communications Controller. The Communications Controller is the peripheral that allow each core to access the packet switching network present in the SpiNNaker system.

All 18 processors nodes are identical which is the cause of problems during the start-up procedures. One of the processors is chosen as the Monitor Processor through a process which will be described later and it will be responsible for booting the chip, the communication with the Host PC and performing the necessary system management tasks. The other processors are named Application Processors and they will be responsible for modelling a group of neurons with associated inputs and outputs, named neuron fascicle, in neural application since the processor are general purpose in essence. Some Application processors may not be used to serve as spares for fault-tolerance purposes. The Monitor processor runs a different kernel, named SpiNNaker Control & Monitor Program (SC&MP), from the Application cores which run the SpiNNaker Application Runtime Kernel (SARK). The Monitor processor is also given access to SystemRAM which is an extra block of 32 Kbyte of on-chip RAM accessible through the System NoC. This memory is not exclusive to the Monitor processor although the latter is its main user to improve its data resources as it will be responsible for the management of the chip and run complex algorithms.

The Router is the hearth of the Communications NoC, taking up 10% of the chip's area [NLMA+09], as it is responsible for the routing of packets between on-chip processors and with other SpiNNaker chips. The system supports four basic type of packets which are distinguished

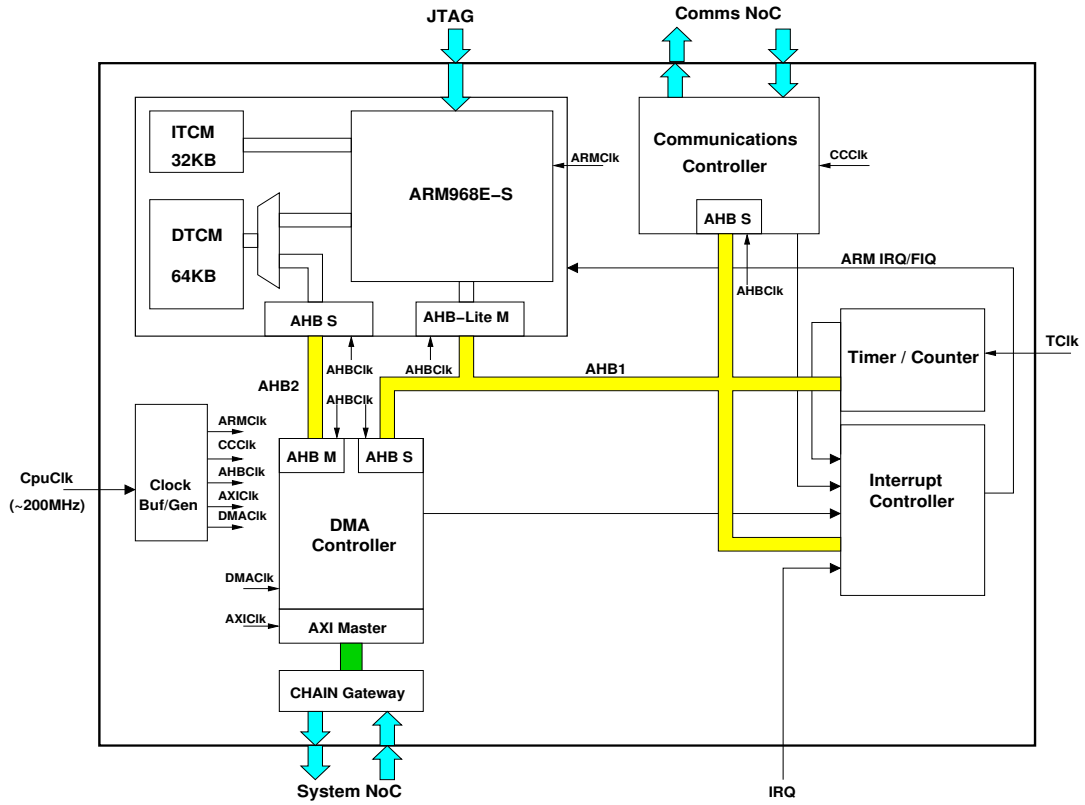


Figure 2.2: Block diagram of the ARM928 core [Gro11b].

through the two most significant bits in the header, the different packet formats are showed in the diagram 2.4. All packets have a 32 bit optional payload whose presence is indicated through bit 1 in the header control byte. The least significant bit indicates if the entire packet has odd parity. For almost all packet types, there is a two bit time stamp present in the control byte. This time stamp is set to the current time phase of the system. There is a global time phase that cycles through 00 -> 01 -> 10 -> 11 -> 00, synchronization should be accurate to within one time phase, which is programmable and dynamically variable. If the Router finds a packet to be two time phases old, easily checked through an XOR operation, then it will drop it to the Monitor Processor. The Router is the one responsible for inserting this time stamp on local packets during normal operation though this behaviour can be overridden through configuration on the Communications Controller.

Emergency Routing

Each chip has six bidirectional links to communicate with other chips. These are numbered from 0 to 5. The recommended connection configuration is a triangular mesh where each chip is connected to six different neighbours. This allows for easy emergency routing in the event of a failing or congested link, the traffic that would be using the congested link is redirected using two adjacent link that form a triangle with the failing link, shown in figure 2.3. Since there are circular

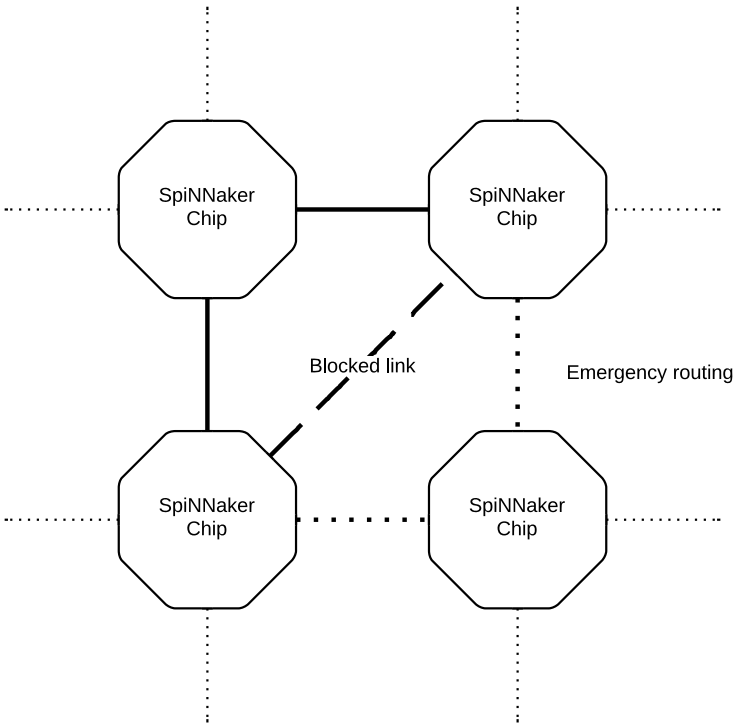


Figure 2.3: Emergency routing.

dependencies between links there are potential deadlock scenarios for which there is a policy in place to prevent them which is "no Router can ever be prevented from issuing its output". In order to enforce this policy, the Router has several mechanisms available, the output has sufficient buffering and capacity detection so that the Router can know whether or not the output can accept another packet. If an output is blocked for any reason then emergency routing is used if possible to avoid overloading the blocked output. In cases where the emergency routing fails the packet is "dropped" to a Router error register and the Monitor Processor notified of this failure. The Monitor Process will then track the problem using a diagnostic counter:

- if the problem was temporary, it will only note it but do nothing further;
- if emergency routing keeps being used for the same route, it will negotiate a new one and divert some traffic to the new link;
- if the problem is permanent, it will establish new routes for all the traffic using this link.

The time taken by the Router to try emergency routing is controllable through its control register, there are two *wait* values, *wait1* is the number of clock cycles that Router waits before trying emergency routing and *wait2* which is the number of cycles that the Router tries to do route the packet through another link before dropping it to the error register and continuing to the next one.

There are 2 bits in the control byte which are used by Multicast and Fixed-route packets to control the emergency routing process. The meaning of each value is as follows:

- 00 - normal packet;
- 01 - the packet has been redirected by the previous Router through an emergency route along with a normal copy of the packet, the receiving Router should treat this as a combined normal plus emergency packet, meaning it will be routed in two different ways;
- 10 - the packet has been redirected by the previous Router through an emergency route which would not be used for a normal packet;
- 11 - this emergency packet is reverting to its normal route.

Multicast Packets (MC)

Multicast packets carry neural event information to be used during model simulations. Each packet contains an identifier which is used as a routing key as well as a neuron identifier in neural applications [PBF+08]. The multicast router behaves like a look-up table with two components, a parallel ternary content-addressable memory (TCAM) and a conventional RAM look-up table. A content-addressable memory is a special kind of memory whose input in a read operation is the data and the output is the address where the data is located as opposed to a read from RAM where the input is a memory location and the output the data stored at that address, the ternary variation include support for a *do not care* bit [PS06]. The router's TCAM has 1024 entries, which must be initialized after reset, each with its own mask and match value. The routing process uses the routing key present in the packet as input for the TCAM, then the TCAM result is used to retrieve from the look-up table the output vector whose value determines where this packet should be sent. In case of multiple matches from the TCAM look-up, the one with the lowest value will be used. The table 2.1 shows how each bit of the output vector, when set to 1, affects the propagation of the multicast packet. When the routing key has no matches then the default routing is employed. The default routing simply outputs the packet on the opposite link of the input link through where it was received. For local packets this routing is not available meaning that if the packet fails to match an entry it will be dropped.

When an output link is blocked, the Router will try to do emergency routing through a link with the next lower number. If the original port becomes unblocked before sending the packet through the emergency port, then the router will retry through the original one. If the Router receives a packet with the emergency packet bits set as diverted, then it will attempt to output it as a *reverting* packet to the output link with the next lower number than the input link number, where it was received. If it is also a normal packet then it will also perform the conventional routing. A received reverting packet is routed normally if it is recognised by the router, otherwise it is *default* routed to the link numbered two greater than the input link.

Point to Point Packets (P2P)

Point-to-point packets carry system management and control information. They are also used to implement a higher level packet transmission which it will be described later. These packets

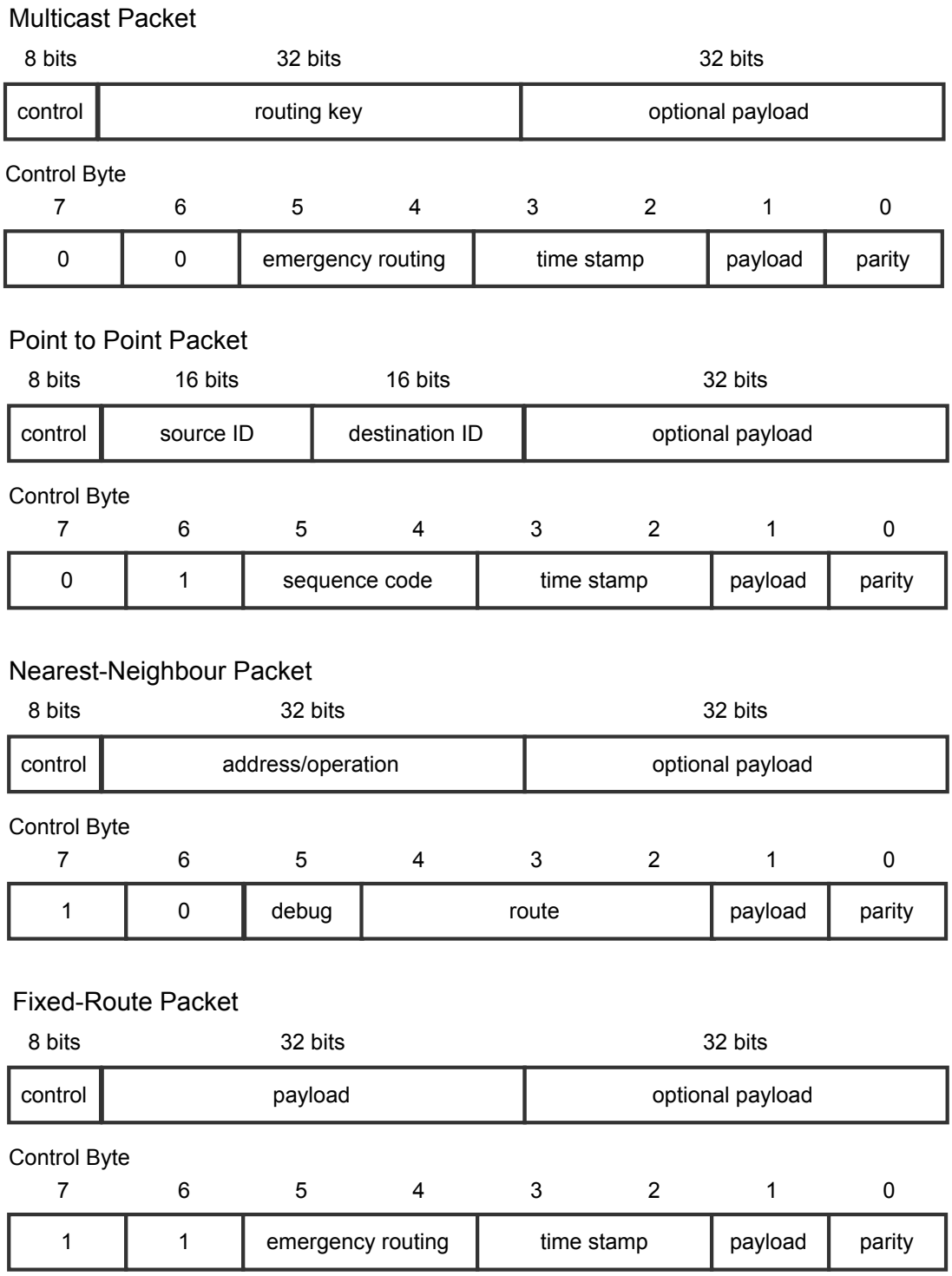


Figure 2.4: The 4 different SpiNNaker Packets [Gro11b].

Multicast Output Vector Entry	Output port	Direction
bit[0]	Tx0	East
bit[1]	Tx1	North-East
bit[2]	Tx2	North
bit[3]	Tx3	West
bit[4]	Tx4	South-West
bit[5]	Tx5	South
bit[6]	Processor 0	Local
bit[7]	Processor 1	Local
bit[8]	Processor 2	Local
...	...	...
bit[23]	Processor 17	Local

Table 2.1: Multicast Output Vector Assignment [Gro11b].

include a source and destination ID, each one with 16 bits. The destination ID is used to determine which output should the packet be routed to. For each ID, there is a three bit entry which is decoded to determine whether the packet should be delivered to the Monitor Processor, sent through one of the output links or dropped. The table 2.2 show the routing behaviour for each possible entry value. These values are packet in a 8K x 24 bit SRAM lookup table sequentially. The use of a Static Random Access Memory helps reducing the access time and power consumption while using the lookup table [YIK87]. The *sequence code* field present in the control byte is under software control and can be used for any purpose.

Nearest-neighbour Packets (NN)

Nearest-neighbour packets are used during boot-time to do flood-fill of the boot image (to be described later), broadcast of P2P addresses and for chip debugging. The routing process delivers *normal* NN packets to the Monitor processor when receiving through one of the six input links and it send through the appropriate output link the internally generated packets. This routing is essential to support the flood-fill load process.

When the *debug* bit in the header control byte is set, the NN packet is interpreted as a peek/poke

P2P Table Entry	Output Port	Direction
000	Tx0	East
001	Tx1	North-East
010	Tx2	North
011	Tx3	West
100	Tx4	South-West
101	Tx5	South
110	none (drop packet)	none
111	Monitor Processor	Local

Table 2.2: P2P Table Entry behavior [Gro11b].

packet which can be used by neighbouring chips to access System NoC resources without processor intervention which means it can be used to investigate a non-functional chip, to re-assign the Monitor Processor or to generally debug and test easily a SpiNNaker chip. The nature of the operation depends on the presence of a payload, meaning that a *write* operation will include a payload while the *read* operation will not. The address/operation field contains the address, within the System NoC address space, where the operation will be performed, either the payload will be written to this location or the current contents will be read into the response packet. There is always a response which is a *normal* NN packet with the same address field for identification purposes with the least significant bit set to indicate a response. In case of bus error while accessing then bit 1 will also be set. The response will also include a payload when replying to a peek packet.

Fixed-route Packets (FR)

Fixed-route packets usually convey application debug data back to the host computer which facilitates monitoring and debugging. Its routing procedure is identical to the multicast packet although since they do not include routing key, the value of a specific Router's register, in this case register 33, is used as its routing key. As a consequence, all fixed route packets are routed using the same output vector. When an output is blocked, the router will use the same emergency routing procedures that it uses for the multicast packet emergency routing.

2.1.1 Inter-chip communication

The on-chip interconnect employs a 3-of-6 return-to-zero (RTZ) self-timed codes and the switching fabric based on CHAIN [FB09]. Each inter-chip link is based on the 2-of-7 non-return-to-zero (NRZ) self-timed coding, where each unidirectional link consists of 7 data wires and an acknowledge signal. The data symbols are sent using transition signalling from the Sender to the Receiver where a transition on 2 of the data wires translate to a 4 bit symbol. The sender waits for a transition on the acknowledge signal to proceed to the next symbol. The logic levels 0 and 1 are represented by voltages of 0 and 1.8V respectively. After the chip reset, the output data wires are brought to logic 0 while the acknowledge goes to logic 1.

There were mainly two motivations to have two different protocols involved in the communications:

- Performance;
- Power consumption.

A RTZ protocol will always have two transitions on each line during a symbol transmission as opposed to a NRZ protocol which only needs one, this translate on easily doubling the throughput. As far as power consumption is concerned, minimizing transitions is essential to maximize efficiency. In this particular case, the 3-of-6 RTZ employs 8 transitions to send 4 bits of data, the 2-of-7 NRZ uses only 3 transitions to send the same data. in the off-chip domain these improvements are essential as chip-to-chip delays dominate performance and wire transitions dominate

Value	L[6]	L[5]	L[4]	L[3]	L[2]	L[1]	L[0]
0	0	0	1	0	0	0	1
1	0	0	1	0	0	1	0
2	0	0	1	0	1	0	0
3	0	0	1	1	0	0	0
4	0	1	0	0	0	0	1
5	0	1	0	0	0	1	0
6	0	1	0	0	1	0	0
7	0	1	0	1	0	0	0
8	1	0	0	0	0	0	1
9	1	0	0	0	1	1	0
10	1	0	0	1	0	0	0
11	1	0	1	0	0	0	0
12	0	0	0	0	0	1	1
13	0	0	0	0	1	1	0
14	0	0	0	1	1	0	0
15	0	0	0	1	0	0	1
EOP	1	1	0	0	0	0	0

Table 2.3: 2-of-7 Symbol coding [Tem12].

power consumption, The 2-of-7 protocol offers the double of performance with less than half energy consumption. These advantages do not apply in the on-chip domain as the simpler logic required to implement the 3-of-6 protocol dominates the decision on both power and performance.

One common issue that self-timed codes suffer from is the lack of resistance to transient or permanent faults. The SpiNNaker inter-chip links have been designed to improve on this weakness by reducing the risk of deadlock as much as possible since preventing data corruption is not possible. Additionally, the SpiNNaker chip has the ability to reset sub-circuits in order to simplify recoveries from deadlocked situations.

2-of-7 Symbol coding

There are 17 different symbols where one is *EoP* which means End of Packet and the others are values between 0 and 15. The table 2.3 shows the encoded symbol for each set of transitions, on this table the value *1* on the data wire should be read as transitions while *0* as no transition.

2.1.2 SDP packets

The figure 2.5 illustrates the different layers of the SpiNNaker communications model, the Open Systems Interconnection (OSI) model was added to show the relative function of each layer as it is a common abstraction. At this point, the bottom two layer have already been discussed. This section will analyse the third layer which builds upon the lower layers.

The basis of all communication from the host system with the SpiNNaker system is done using SpiNNaker Datagram Protocol (SDP) packets. The protocol is similar to User Datagram Protocol (UDP) since it is sent in datagrams and it has very few capabilities embedded in the protocol itself

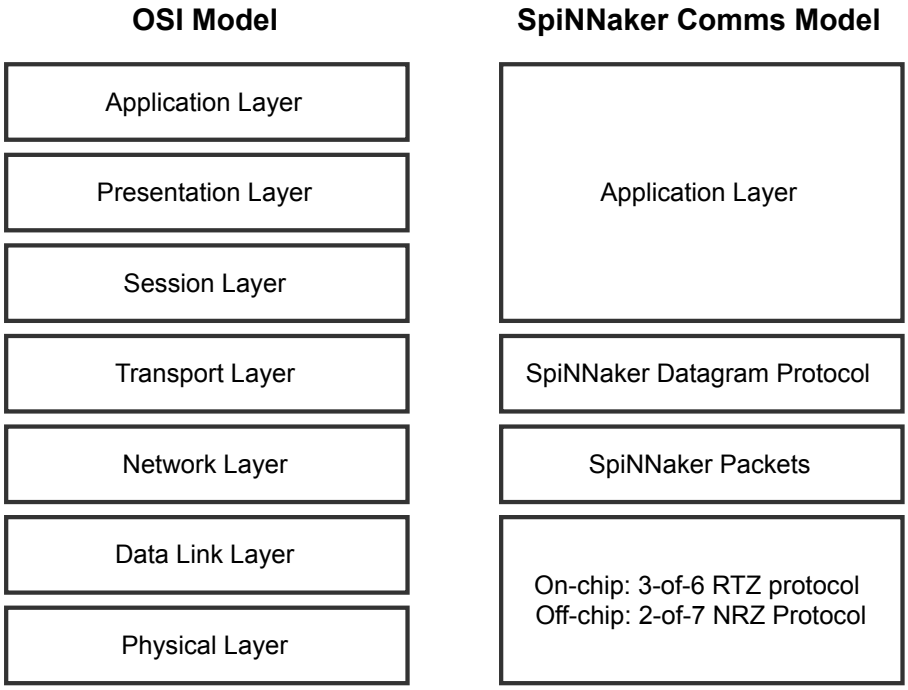


Figure 2.5: SpiNNaker Communications model.

besides the transmission of a limited amount of data. The SDP datagram may contain up to 64 kilobytes though the current implementation in the SpiNNaker kernel limits this value to 272 bytes in order to minimize the size of the buffers required.

A SDP packet contains a header with information needed for its routing and a payload field with the data that is being sent. Since SDP provides point to point communication, an addressing scheme is needed to identify the endpoints. As it can be seen in Figure 2.6, for each endpoint there are 3 fields with each taking 1 byte. Two fields, "Addr X" and "Addr Y" , are used to identify the chip's position inside a SpiNNaker system shaped into a 2D grid. The third byte is divided in two fields where the lower 5 bits are the Virtual CPU number which is unique within a SpiNNaker chip and the 3 high bits are a port number which attaches the packet to a particular process on that CPU. The convention which is in place states that port 0 is reserved for communication with the kernel running on the core and the rest of the ports are available for applications. There are two more fields in the header that must be mentioned, the Flags field which should be set to 0x87 or 0x07 if a reply is expected or not, and Tag field which holds the IPTag.

An IPTag is a small number that is used by the SpiNNaker node connected to the Ethernet port to know where to send each output SDP packet bound to the Ethernet port. This SpiNNaker node maintains a mapping between the IPTag and the IP address/port pair. Each IPTag can be permanent or transient, permanent tags are set through commands which will later be described, transient tags are created when a new SDP packet arrives for which a reply is expected. The newly created tag is then written to the packet header before delivering it to its destination. As soon as

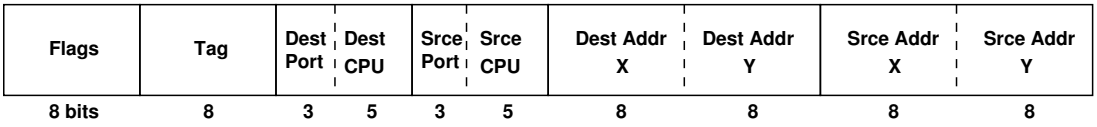


Figure 2.6: SDP packet header [Tem11c].

reply reaches the Ethernet-attached node, the tag is used to retrieve the IP address needed for its routing and the IPTag table entry is deleted at this point. A transient tag can have a associated timeout that in case the reply fails to arrive the table entry can be reclaimed. The current IPTag table implementation in the SpiNNaker kernel has 16 entries where the first 4 are reserved for permanent tags.

Regarding the Virtual CPU number, each core on a SpiNNaker chip has a physical CPU number which is hardwired from 0 to 17. During the boot up process, a processor is selected as Monitor processor and is assigned the Virtual CPU number 0 and the rest of Application processors from the number 1 upwards from which non-working CPUs are excluded.

SDP packets can be conveyed in a number of manners, for example, when sent from the host system to a SpiNNaker system they are embedded within a UDP packet as seen in Figure 2.7. The 2 byte pad present at the beginning of the UDP data aligns the data to a 4 byte boundary which makes the processing in a SpiNNaker rather easier. The first of the these two bytes is an argument which is used as the IPTag timeout. The valid values for this timeout go from 0 to 16 whose meaning can be checked in table 2.4.

SDP over P2P

While the SDP packets are usually introduced into a SpiNNaker system through the Ethernet port, they are transmitted between SpiNNaker chips using P2P packets. This protocol is undocumented and it was reverse engineered from the kernel source-code. The Figure 2.8 shows a typical protocol run with no errors or retries. The initial step is a request from the sender to open a *channel* for reception. The receptor will reply with an open acknowledgement packet with the *channel* id or with an error code in case either the SDP packet is too large (over 280 bytes) or there are no free *channels*. Having received a normal acknowledgement packet, the sender will send sets of 16 data packets, each with 24 bits of data. At end of this set, the receptor will send a data

Value	Meaning
0	No timeout (infinite)
1	10 ms
2	20 ms
...	...
N	$10 * 2^{N-1}$ ms
...	...
16	327680 ms

Table 2.4: IPTag timeout values.

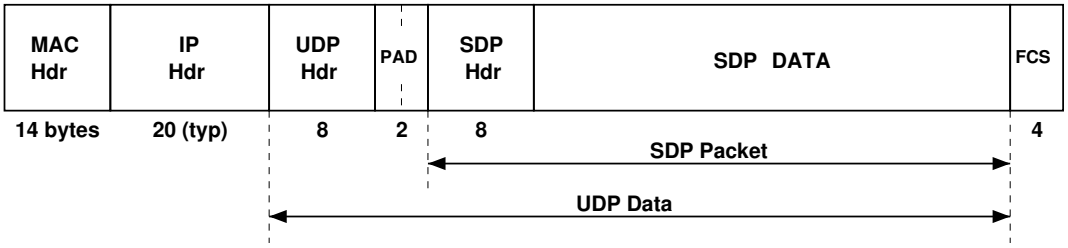


Figure 2.7: SDP packet embedded within a UDP packet [Tem11c].

acknowledgement packet which allows the sender to send the next set of data packets. This cycle continues until the receptor has all the data packets it needs. At this point, the receptor will send a close request packet to the sender to which it should reply with a close acknowledgement packet. Then both ends close the connection. The detailed packet format for each step and their content is presented in the appendix B.

Since this connection can be made between any two SpiNNaker chips in the system, there is fairly big timeout for each step of the protocol. Additionally, in case of errors or timeout there are also retries mechanisms for most steps of the protocol. The specific values being used in the current implementation are presented in the table 2.5 and 2.6.

SpiNNaker Command Protocol (SCP)

One of the uses for SDP packets is to convey commands and responses around a SpiNNaker system. In this case, the SDP data field is structured into 6 fields which are shown in the Figure 2.9. The *cmd_rc* field indicates either the command that is being specified or the return code of a command execution when the packet is a response. The *seq* field may be used to create a retry mechanism on top of SDP. The fields *arg1*, *arg2* and *arg3* may contain 32-bit arguments or return values while the data *data* field may contain any kind of data up to 256 bytes.

SDP packets with SCP commands should be sent to port 0 so that the kernel running at the core can process them. There are five commands which are currently implemented by both kernels. Their names and actions can be seen in Table 2.7, the reader interested in higher levels of detail should refer to [Tem11d].

Protocol Step	Timeout value (ms)
Waiting for Open ACK packet	250
Waiting for Data ACK packet	3000
Waiting for a Data packet	500
Waiting for a Close ACK packet	250

Table 2.5: Timeout values for SDP over P2P.

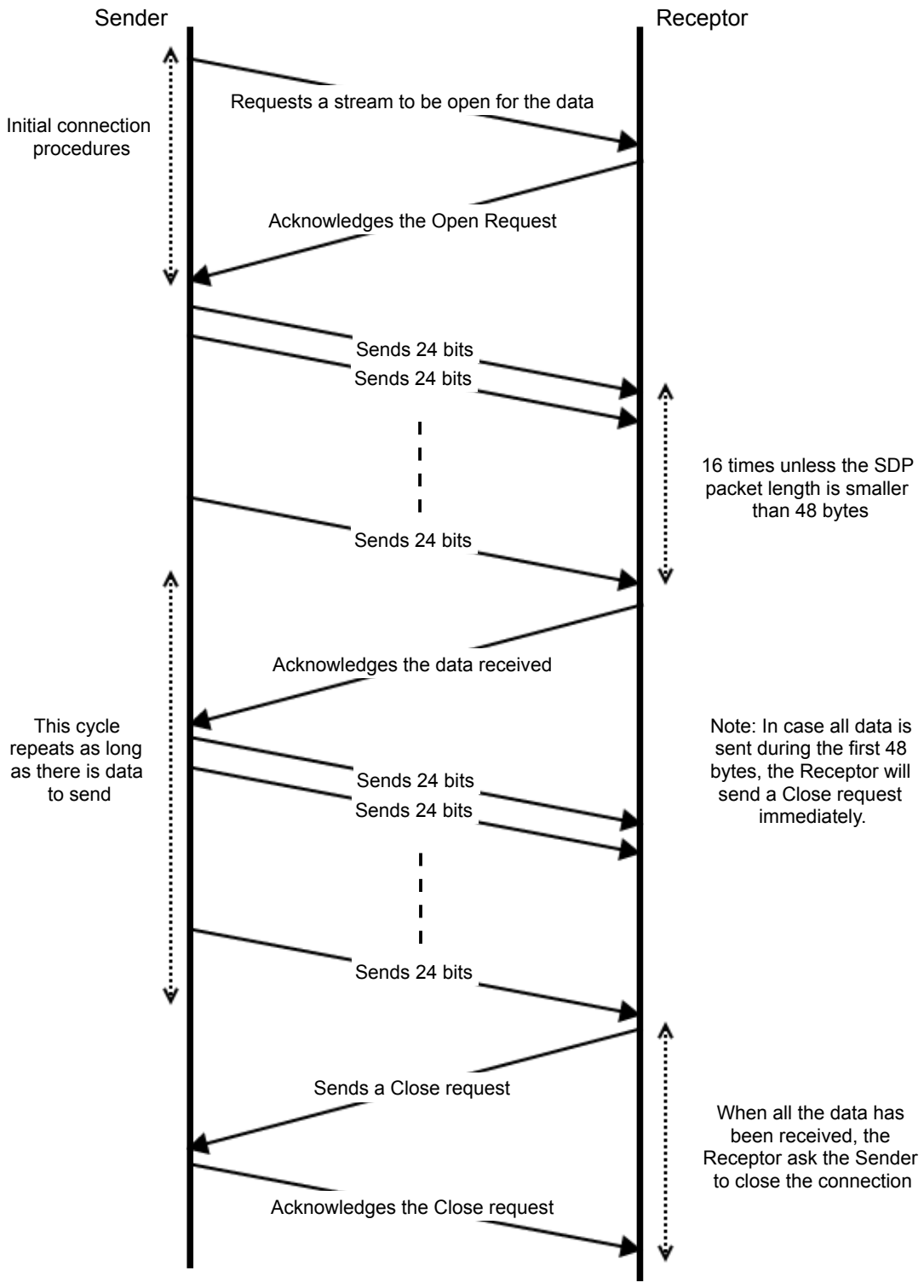


Figure 2.8: SDP over P2P protocol.

Protocol Step	Retries Count
Retries for Open requests	16
Retries for starting data transmission	4
Retries for resuming data transmission	4
Retries for Close requests	4

Table 2.6: Retries count for SDP over P2P.

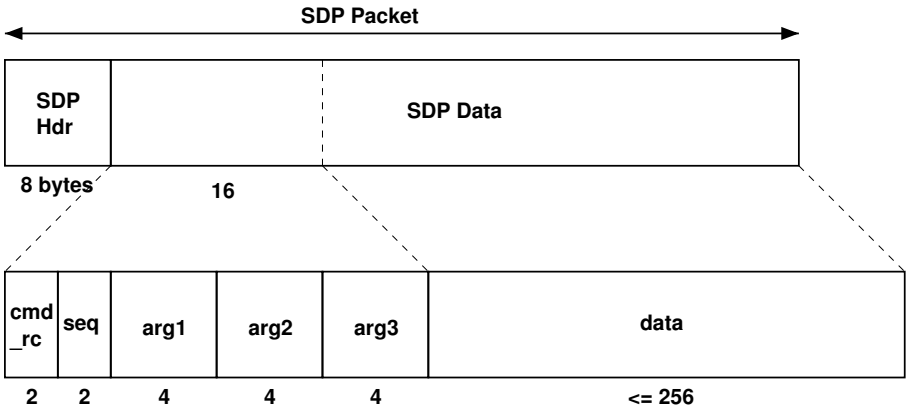


Figure 2.9: SCP packet [Tem11d].

Command Name	Action
CMD_VER	Retrieves the version of the kernel
CMD_READ	Allows the reading of memory from the core's address space up to 256 bytes. The read operation can be requested in byte, halfwords and words.
CMD_WRITE	The equivalent of CMD_READ for writing operations
CMD_RUN	Low level command which instructs the core to start executing at a specific address, it is not usually used.
CMD_APLX	The usual way of starting an application. This instructs the kernel to process an APLX file, which must have been loaded to memory (using CMD_WRITE) beforehand.

Table 2.7: 5 SCP commands that both SC&MP and SARK implement.

2.1.3 SpiNNaker machines

The Advanced Processor Technologies Research Group (APT), that has developed the SpiNNaker system, has also developed several SpiNNaker machines, each with different number of SpiNNaker chips. Their names are formatted as 10N where N is an integer number which tells the user that the machine has approximately 10^N cores. The 102 and 103 machines are single printed circuit board (PCB), larger machines are cabinets and/or racks of the 103 machine [Gro13].

102 machine

The 102 machines is a 4 node board with 72 ARM cores, with 64 usually deployed as Application cores, 4 Monitor processor and 4 spares cores. It is the smallest SpiNNaker machine currently available measuring 105×95mm and weighting around 50 grams. It has two connectors that expose the SpiNNaker's inter-chip link interface and a 100Mbps Ethernet standard plug, RJ-45. This machines requires a 5V 1A supply and it is depicted in Figure 2.10.

103 machine

The 103 machine has 48 nodes, which leads to 864 ARM processors. From this total, 48 are used as Monitors processors and 48 as spares processors which leaves 768 cores to be used as application cores. There are six 3.1Gbps high-speed serial interfaces, which use SATA cables though not necessarily a SATA protocol. These interfaces are used to build the larger machines, but they can also be used as general purpose high-speed input/output communications by configuring the on-board FPGAs. These machines require a 12V 6A supply and it is depicted in Figure 2.11.

104, 105 and 106 machine

The larger machines are increasingly bigger sets of 103 machine linked together through the SATA links available on those boards. Figure 2.12 depicts the amount of cores available with each machine, their relative sizes and their power consumption.

2.2 Application Loading

Besides the Node-Boot described below, a SpiNNaker system has no non-volatile storage from where it can load the software that will be running, so a host machine, usually a desktop or laptop, is needed. The connection between the host and the SpiNNaker is usually done with an Ethernet cable since every SpiNNaker chip has an Ethernet controller hardware on-board, though a direct connection is only possible when this is connected to an external transceiver responsible for connecting the link layer to the physical layer of the OSI model,hereafter many times addressed as PHYceiver, and a SerialROM chip with a valid IP and MAC address.

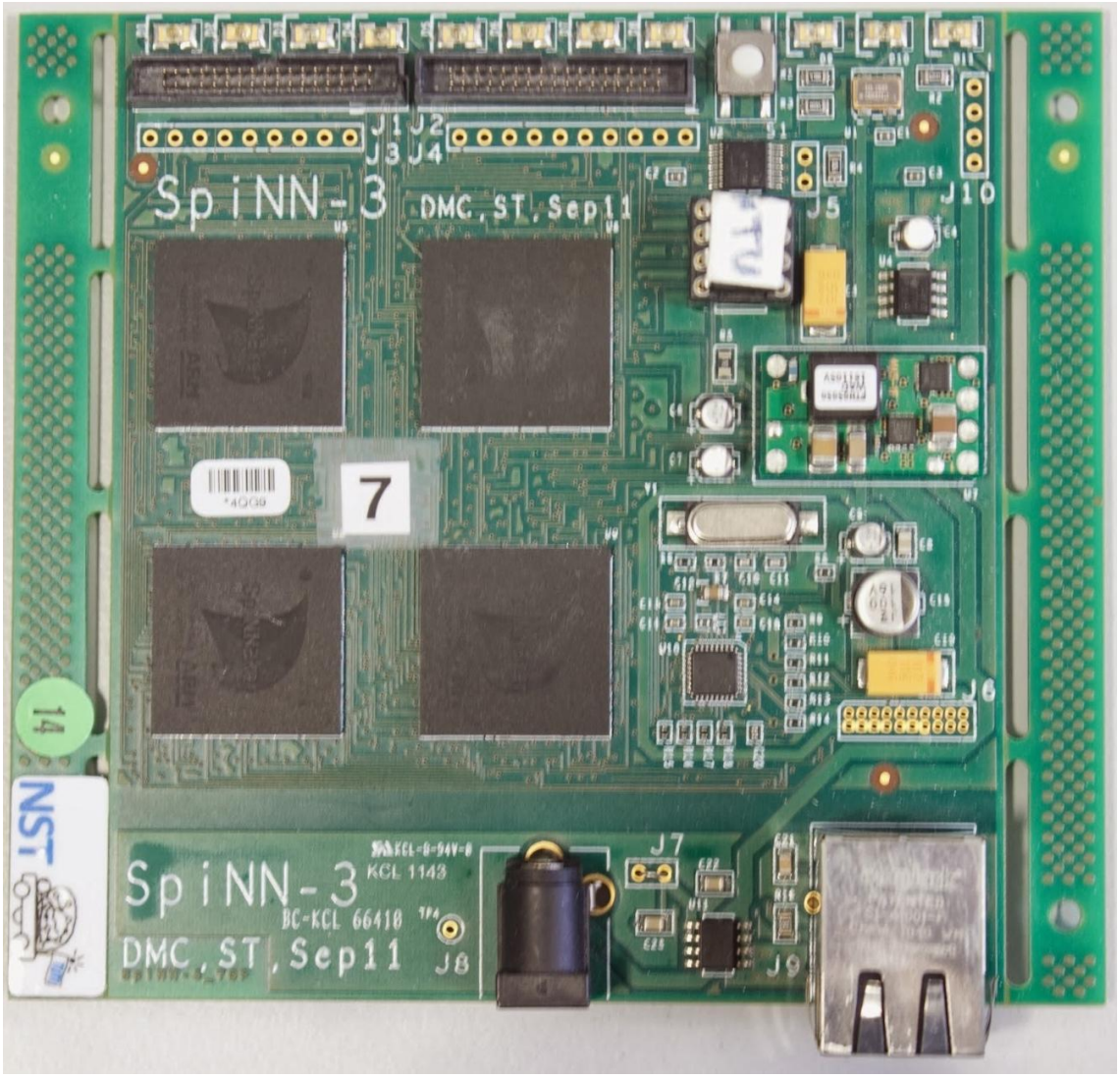


Figure 2.10: 102 Machine [Gro13].

2.2.1 Boot sequence

There are 3 stages on the SpiNNaker booting sequence:

- The *Node-Boot* phase which executes the code from a read-only on-chip memory, *BootROM*. This code was designed to be as simple as possible to minimize the probability of bugs since it is not possible to correct them after production. It is also responsible for the initial chip testing and initialisation, the election of a Monitor Processor and having the node ready to receive the second stage image.

- The *System-Boot* phase where a boot image is received on the Ethernet connected SpiNNaker node and is propagated to its neighbours until every Monitor Processor in the system is running the second stage image. This propagation of the boot image through the system

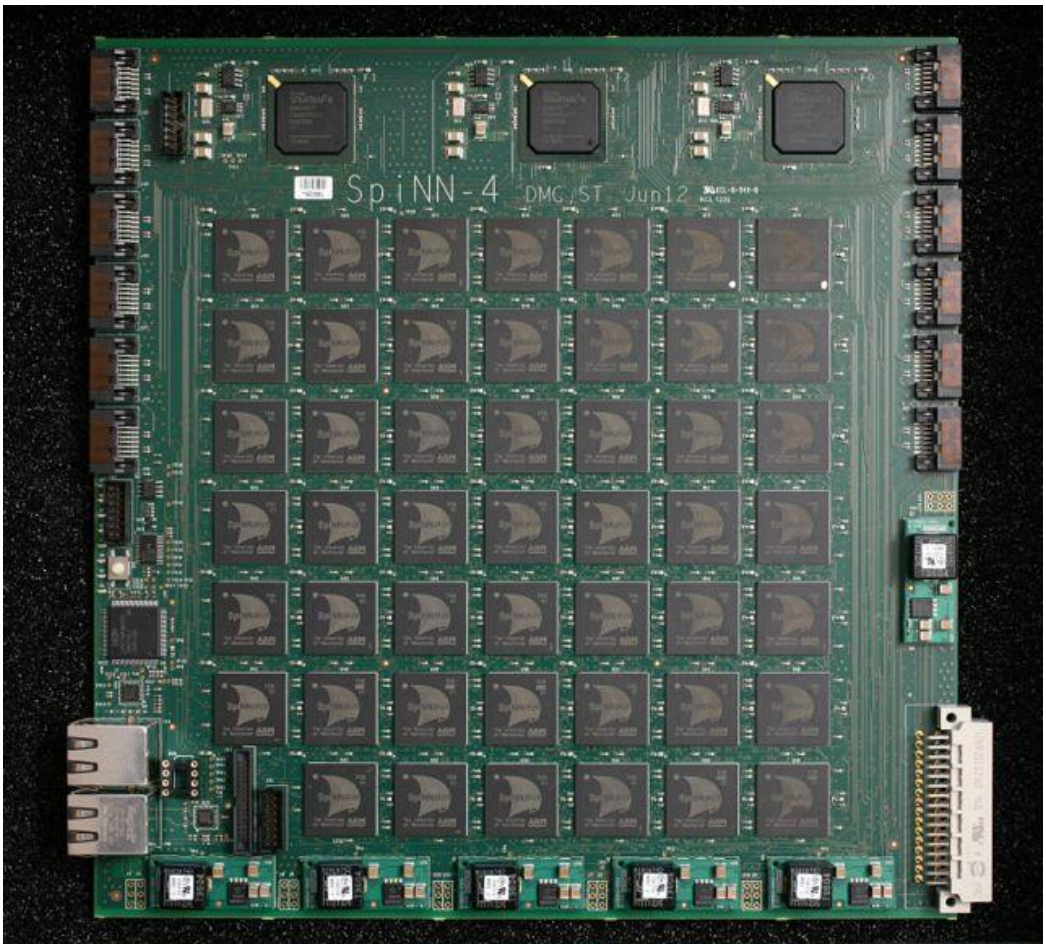


Figure 2.11: 103 Machine [Fur13].

is known as *flood-fill*. Following this, every working core in each chip is assigned a Virtual CPU number as described above.

- The final stage is the *Application-Load* where the application software is loaded from the host machine to the chosen cores, the application data uploaded to the shared SDRAM and the route tables populated.

Node-Boot

This is the first phase of the boot and the one where the initial checks are performed to check on the status of the memories and the peripherals, any failures at this stage will lead to the shutdown of the core as may be seen in Table 2.8. The very first step of the boot sequence is to check using the external General Purpose Input/Output (GPIO) pin 7 whether a manufacturing test should be run or if the default BootROM should be executed. Assuming the latter, the cause of the boot is probed at the Register 12 of the System Controller.

If the reset was caused by the Watchdog then it is detected if an ITCM Validation Block (IVB) has been set up to detect whether the instruction memory is intact and has not been corrupted by

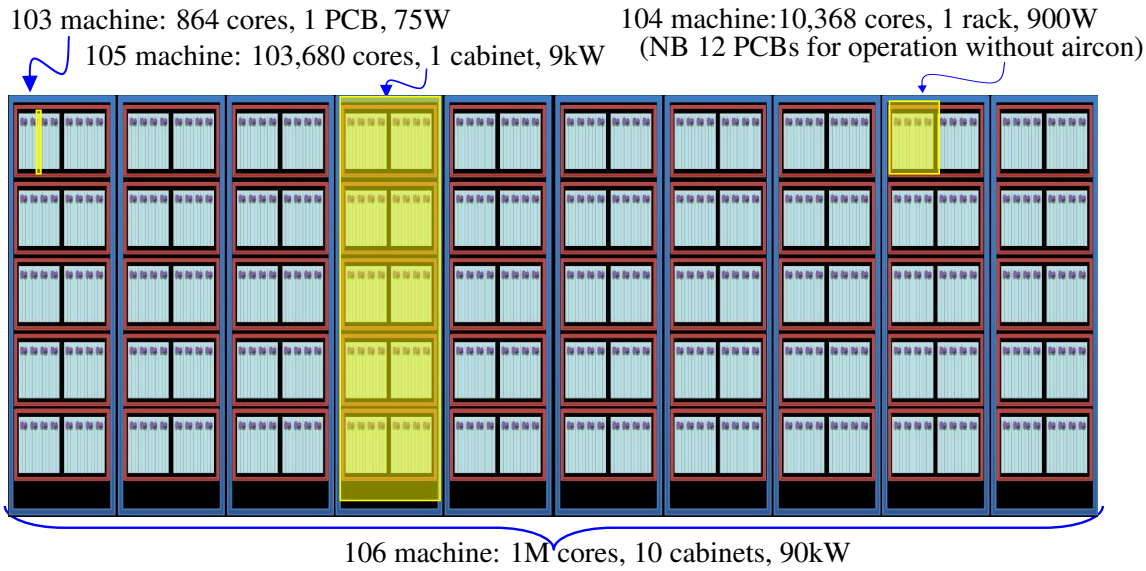


Figure 2.12: SpiNNaker Machines [Fur13].

Peripheral	Method	Executing Core	Failure response
	At Power-on		
ITCM	RAM test	All	Shutdown Core
DTCM	RAM test	All	Shutdown Core
	After Scatter loading		
Comms controller	Register test	All	Shutdown Core
DMA controller	Register test	All	Shutdown Core
Timer	Register test	All	Shutdown Core
VIC	Register test	All	Shutdown Core
	After Monitor election		
Previous Monitor	Check Register	All	Shutdown Core
SystemRAM	RAM test	Monitor	Shutdown MP Core
Router	Register test	Monitor	Shutdown MP Core
Watchdog	Register test	Monitor	Shutdown MP Core
PL340	Register test	Monitor	Record, continue
SDRAM	RAM test	Monitor	Record, continue
	Exceptions		
Exception	Hi Vectors	All	Shutdown Core
Exception	Low Vectors	All	Shutdown Core

Table 2.8: Ordered list of power-on self-tests performed during Node-Boot [Gro11a].

any software malfunction, and to resume execution immediately. This capability is useful in cases when the simulation is already running in the system. In this case the rest of the chips would not have code to restore the reset node and the routing paths would be interrupted. The fault may have been only a glitch or a transient fault which could mean that all the operating environment may yet be in memory. The IVB is a series of checksums that allows the node to be sure that its instruction memory was intact to start executing right away, otherwise it would proceed through with the normal boot.

If it was a non-watchdog reset, as for example a power on reset, then one processor is chosen as Boot processor through external GPIO pins. This core checks for the presence of an external SerialROM chip which usually provides the MAC and IP addressing information though it may be optionally be used to exit the Node-Boot sequence early by loading and executing an image hosted in this chip. After this point, all core clocks are boosted to 160 MHz to accelerate the boot sequence since up until this point all the clocks were running at 10 MHz.

In order to initialize other peripherals, a Monitor processor must be elected first. The election is performed by a hardware mutex in the System Controller implemented as a read sensitive register. The first processor that reads back the register is selected as a Monitor. The cause of the reset is again important, if it was a soft-reset, caused by the watchdog or an intervention triggered due to an unexpected failure, it could have been caused by an error within the Monitor processor so if this core was a Monitor before it will shut itself down so that other cores may be elected, otherwise the history of Monitor Processors is cleared so that every core has a chance to be elected. The newly selected processor marks itself in the bit-wise "Monitor History" System RAM location which will be used in the soft-reset scenarios. Other processors will then become Application processors and will wait for the Monitor to finish the remaining initialization procedures.

At this stage the remaining peripherals to be initialized are the Router, the Watchdog, the Ethernet and the GPIO pins. The router tables are populated with blank entries meaning that the P2P packets will be dropped and it will only transmit multicast packets with the default routing. The Watchdog is set to expire every 1.25 seconds and the chip will reset after the second timeout of the watchdog timer. This should not happen during normal operation as it should be refreshed every 5 ms. For the Ethernet to work, an external PHYceiver must be provided and a SerialROM with the IP and MAC addresses information, it will only be initialized if both are present and functional. The last step before signalling to the Application Processors that they may continue the booting process is the initialization of some GPIO pins where there are usually some LEDs connected. These are useful to denote the stage of the boot process since the frequency of its flashes is different on every stage. One important thing to notice, is that the failure to initialize correctly the router or the watchdog will lead to the shut down of the Monitor processor and to the restart of the SpiNNaker chip, where a different Monitor will be chosen.

The final step before waiting for the reception of the System Boot image is the initializations of the processors' timers and their Vector Interrupt Controller (VIC). The timer 1 is set to tick every millisecond in all cores but the VIC configuration depends on whether or not the processor is the Monitor processor. The Application processors only take action when the System Controller

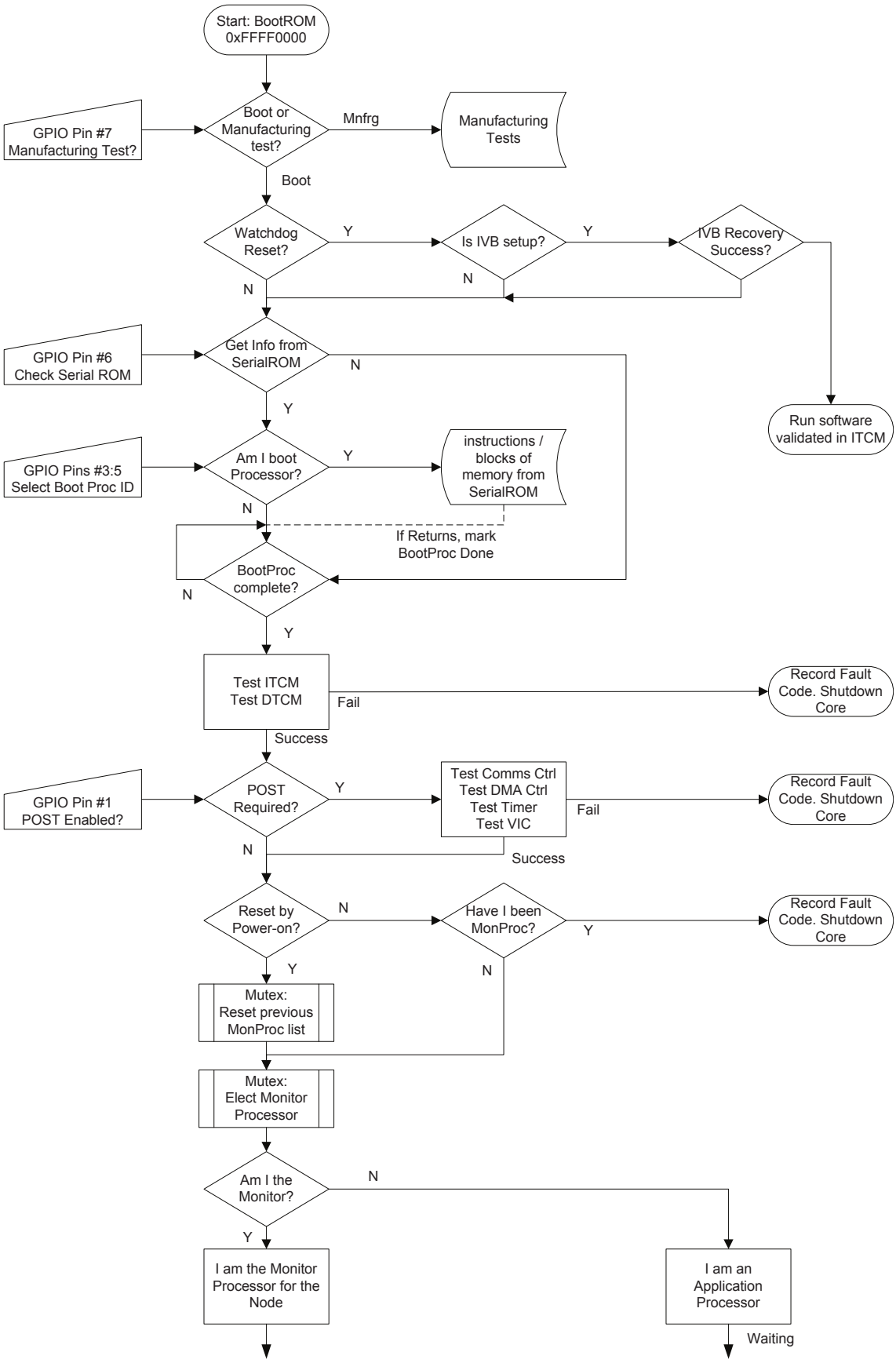


Figure 2.13: Node-Boot process until the choosing of the Monitor processor[Gro11a].

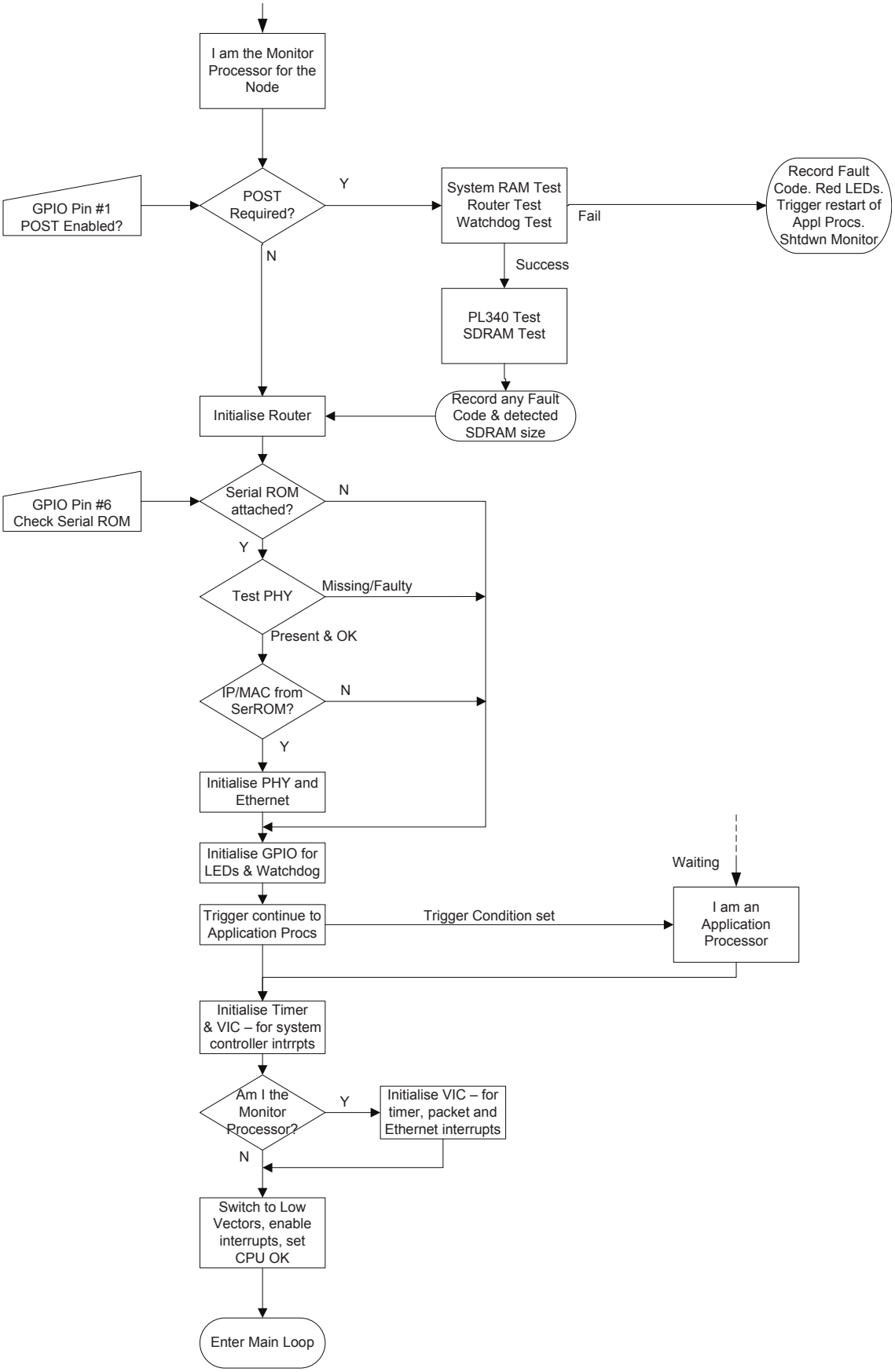


Figure 2.12: Node-Boot process after the choosing of the Monitor processor[Gro11a].

informs them of a message to be processed in the mailbox while the Monitor core takes care of the Timer interrupts, communications interrupt from the inter-chip links and the Ethernet interrupts if it is enabled in the current chip.

After all the tests and initialization procedures, every processor enters a main loop where it is in a low-power state waiting for interrupts. As soon as the interrupt is served, the processor goes back to this low-power state. The Monitor Processor is responsible for refreshing the Watchdog timer every 5 ms and it is now in a "listening" mode waiting for the System-Boot image to arrive for the boot process to continue. This process is shown in Figure 2.12.

System-Boot

The code which is responsible for the Node-Boot is loaded from a Read Only Memory (ROM) where it was decided to keep it as simple as possible to reduce the probabilities of serious bugs in the code base. As a consequence of this design decision, a second stage in the booting sequence is needed where the kernel is pushed to the system by the Host System using either the Ethernet port if the chip had an external Phyceiver and a SerialROM with valid addressing information or through the inter-chip links.

Every chip in the system has a Monitor processor which is waiting for this image. This image can perform further checks and verifications and it can prepare the system to receive the application code which will be run by the Application cores.

Since the Host System is usually connected to the SpiNNaker system through an Ethernet connection, the second stage image is pushed to the system UDP packets. This process is named Host Boot. As soon as the image is assembled, the node starts transmitting the image through all interchip links using NN packets. This second kind of booting is named Inter-chip Boot. This process of transmitting the second stage image is named fill flood and it is the mechanism that allows through a single source boot the entire system. The reception of this image is implemented through a fairly simple state machine which is represented in Figure 2.13.

Host Boot

The Host system uses a very simple 3 packet scheme to push the image to the SpiNNaker system. There are:

- Flood-Fill Start,
- Flood-Fill Block,
- Flood-Fill Control.

The Flood-Fill Start packet signals to the SpiNNaker chip that it should get itself ready to receive an image of up to 32 KBytes split up into the number of blocks indicated in the third operator of the packet. The image is assembled in the top half of the Monitor Processor's DTCM, and there is a reception array, which is initialised with zeros, with an entry for each Block ID expected.

The Flood-Fill Block packets contain the split image and are sent block by block in sequential order. The Block IDs start from zero and the block size is indicated in words. Every block should

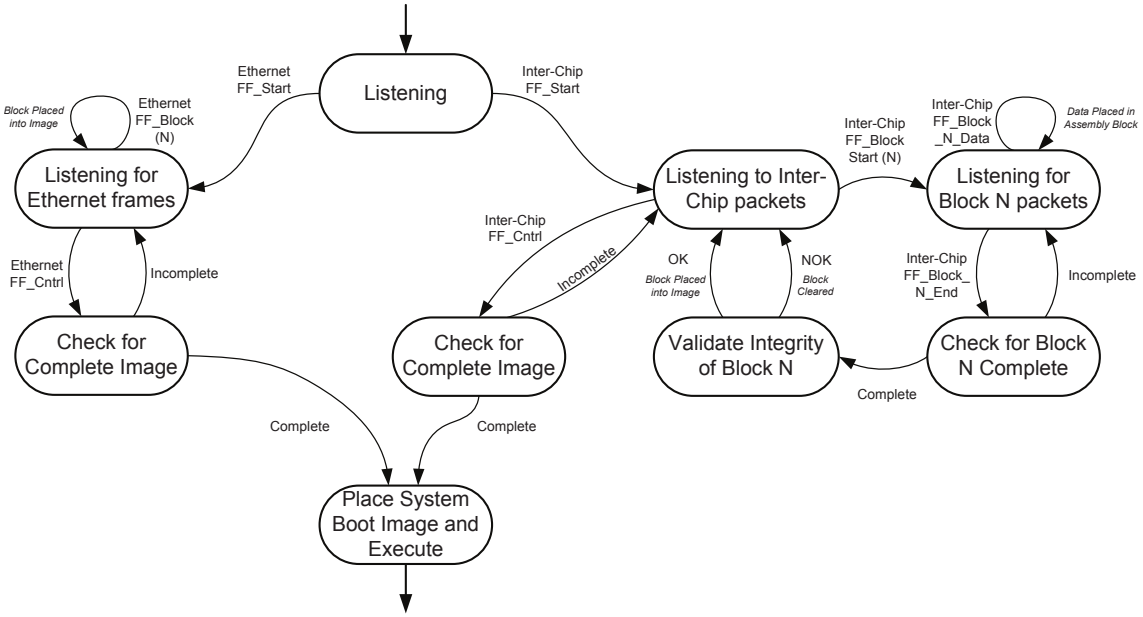


Figure 2.13: State machine for the reception of the System-Boot image [Gro11a].

be equal in size since the block size is used as an offset in the assembly of the image, if the last block is smaller than the previous then it should be padded by the host system until it has the selected size. Looking at the Block size and Block ID fields which are both 1 byte long, it is clear that the maximum value (256 x 256 x 4 = 262144) can be much larger than the allowed 32 KB which nevertheless should always be respected. In order to reduce overhead, the block size should be divisible by the maximum image size and as large as possible. For production use, up to 32 block of 256 words is recommended as it has the lowest overhead possible. Every block that is successfully received is copied to the appropriate position in the image and the reception array updated with an indication that this block has been received. The detail format of the packets can be seen in Figure 2.14.

After all data blocks have been sent, the host transmits a Flood-Fill End packet. The SpiNNaker chip will then check if all blocks are in place, if they are not it keeps waiting for the missing Flood-Fill Block packets. However, this is uncommon so the assembled image is copied to the beginning of ITCM (address 0x0) and starts executing from the start address specified in the packet. This is usually 0x0.

There are a number of considerations that the host system should take while transmitting the image. The byte order to be used at this stage is network order also known as big endian order, the SpiNNaker chip will take care of switching the byte order when it receives the packet. Additionally in order to prevent missing packets while transmitting the image, there should a be a inter-packet delay, it is suggested a rate of 1 ms per block which has been tested successfully with the chip.

These boot packets are sent in UDP datagrams that allow the SpiNNaker system to be booted remotely but it also means that there is not a read receipt nor any guarantee of packet reception. Since the SpiNNaker chip is a passive receptor, they are unable to request the missing blocks

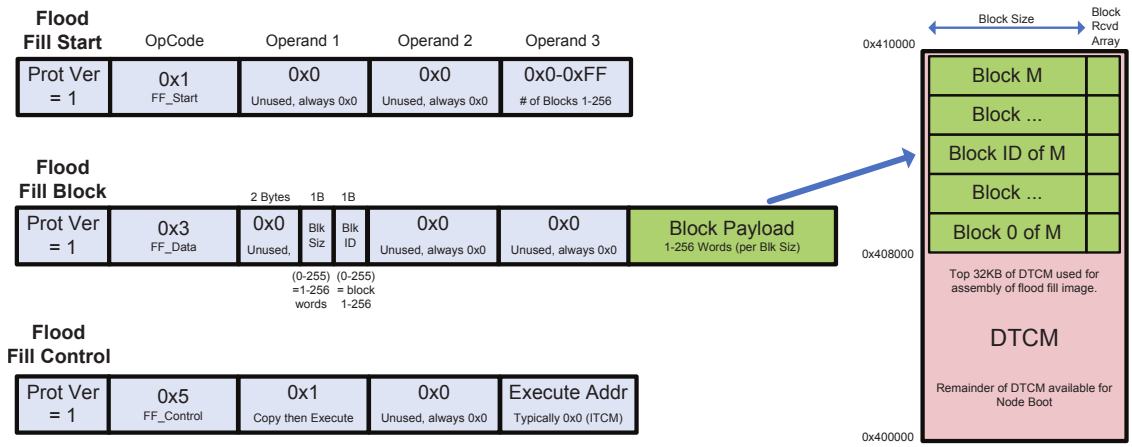


Figure 2.14: Packet scheme used by the Host System to push the second stage image to a SpiNNaker chip [Gro11a].

to be re-transmitted. Therefore the host system does not know which packets may have gone missing so it must always retransmit the full image. It is suggested that the SpiNNaker chip sends a message signalling that it has started executing the image so that the host may stop the transmission. Another possibility is a limit on the number of retransmissions.

Inter-chip Boot

The Inter-chip boot is the transmission of the System-Boot image through SpiNNaker packets, in this particular case nearest-neighbour packets. This transmission is started very early in the System-Boot execution and is done on all six inter-chip connections. The figure 2.15 shows the packet model used. Since the SpiNNaker packets only allow 32 bits payloads, the model had to be extended from the Host Boot version to five different packets. There are:

- Flood-Fill Start,
- Flood-Fill Block Start,
- Flood-Fill Block Data,
- Flood-Fill Block End,
- Flood-Fill Control.

The Flood-Fill Start packet triggers a very similar behaviour to the analogous packet in the Host Boot process. It informs the receiving SpiNNaker chip that it is going to receive an image of up to 32 KB divided in the number of blocks specified in the packet payload. As it happens with the Host Boot process, the image will be assembled in the top half of the DTCM and a reception array is initialized with an entry per expected Block ID.

The Flood-Fill Block Start packet has the Block ID that is about to be transmitted and its size. The block size should be stable during the transmission, including the final one, as it is the case with the Host Boot. The recommendation in this case is to send always the largest block possible,

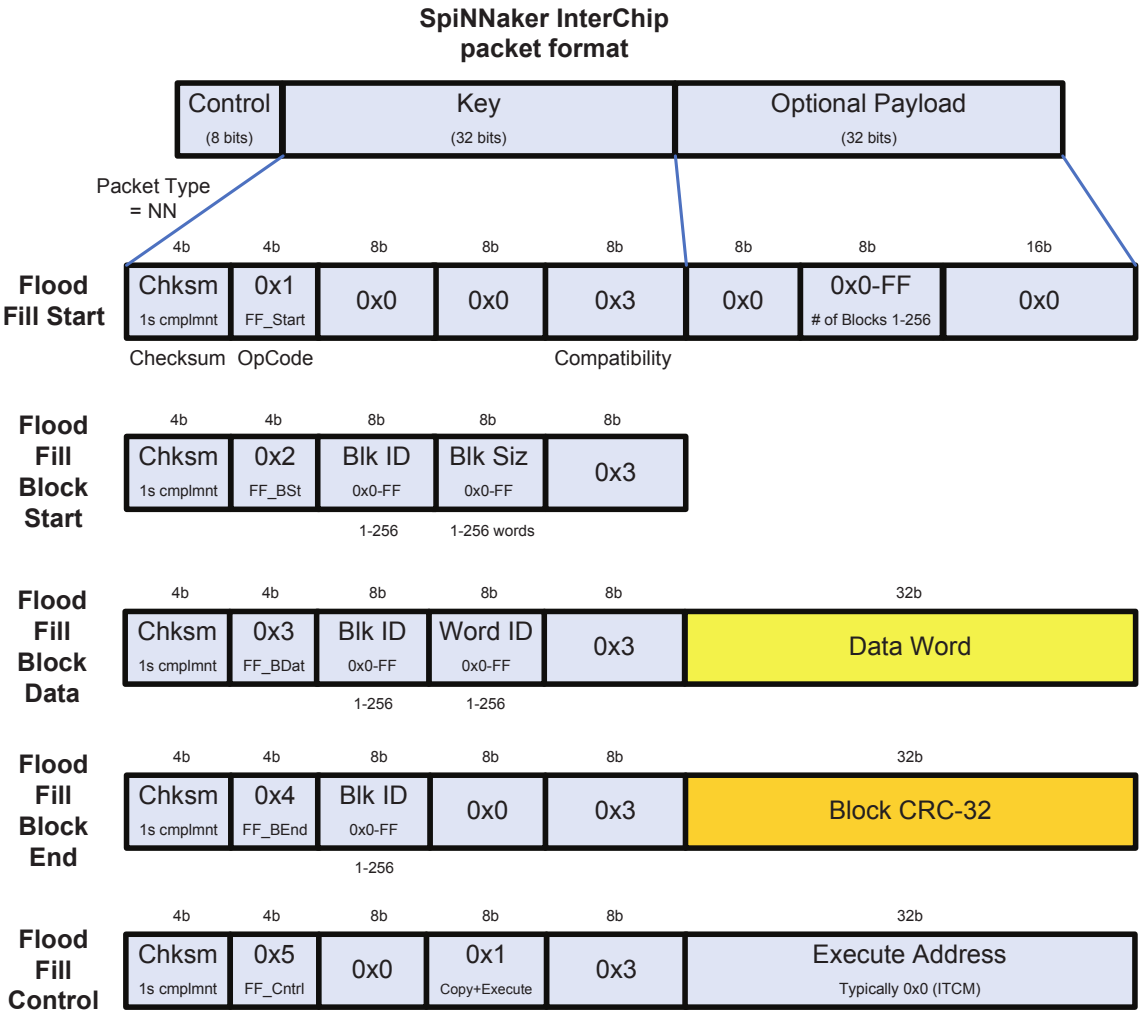


Figure 2.15: Packet scheme used by the SpiNNaker chip to push the second stage image to a neighbour chip [Gro11a].

256 words, and add padding if the last one requires it. This allows the reduction of overhead related with the transmission of packets marking the start and end of blocks. A word reception status array will also be initialized for this Block with an empty entry for each Word ID. The block is assembled in a 257 word block of data present in SystemRAM. The extra word makes room for the CRC that is sent at the end of each block.

The Flood-Fill Block Data packets are the ones responsible for actually carrying the image data 32 bits at a time. They also carry the Word ID and Block ID, if the block ID does not match the block that is currently being received then the packet is dropped since only one block is populated at a time. The word ID is used to place the word in its correct place in the SystemRAM's block of data and to mark on the word reception status array that it has been received.

The Flood-Fill Block End packet marks the end of a block transmission. It triggers a check on the word reception status array, if a word is missing then the process stops here waiting for the missing words to be received. However, this is not a typical situation and the validation step can start. Only after validating a received block, it will be copied to the its place in the top half

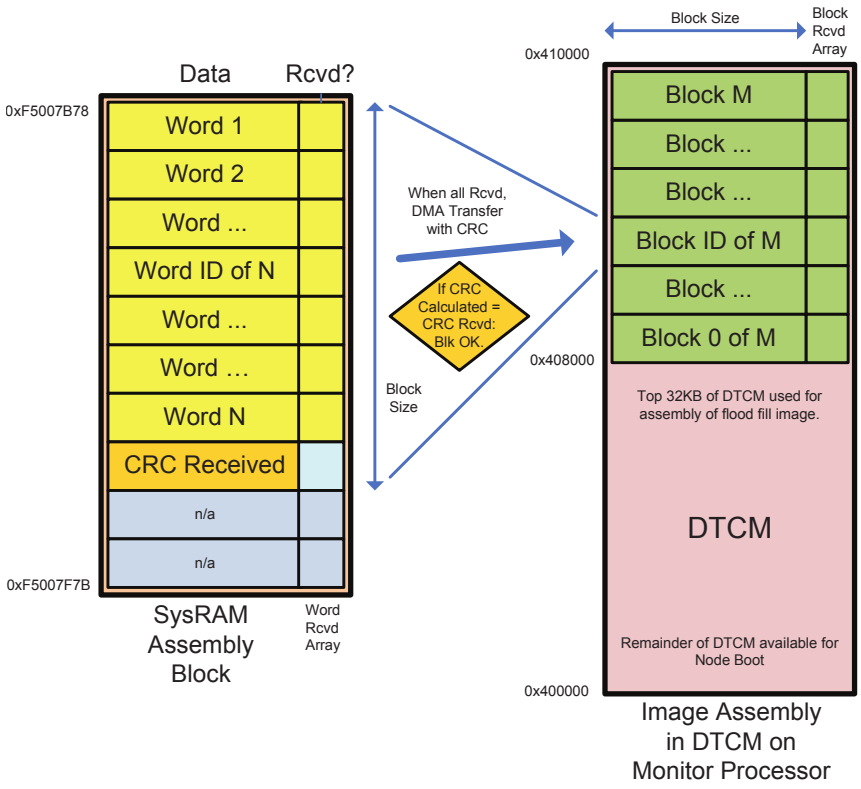


Figure 2.16: Word array for a Flood-Fill Block [Gro11a].

of the DTCM, shown in Figure 2.16. This explicit validation is needed as opposed to the Host Boot scenario where the implicit validation is provided by the CRC value present at the end of the Ethernet frame. In this case, the calculated CRC-32 value is the payload which is placed at the end of the block as shown in Figure 2.16. The CRC check occurs while performing a DMA from the block assembled in SystemRAM into the appropriate position in the DTCM image. If it fails, the block is discarded since it is not possible to calculate which word was received with errors. There is a 4-bit 1s complement packet checksum on every NN packet but it is not particular resilient which means it can still let bad packets go through. After this packet, there should be either a Flood-Fill Block Start packet to restart the process for another block or a Flood-Fill Control packet for the final step.

The Flood-Fill Control packet is very similar to the one present in the Host Boot just like the Flood-Fill Start packet was. After receiving it, the SpiNNaker chip checks if all the blocks have been received, in case there is a block missing, it will stop and wait for missing blocks. It will usually have every block already so the system will copy the image from DTCM to the address 0x0 of ITCM and start executing from the address indicated in payload of the packet.

2.2.2 Application Load and Execute (APLX) File Format

After the second stage boot, the SpiNNaker system is capable of understanding and replying to SCP commands which eases the task of loading application code to the SpiNNaker chips. Cur-

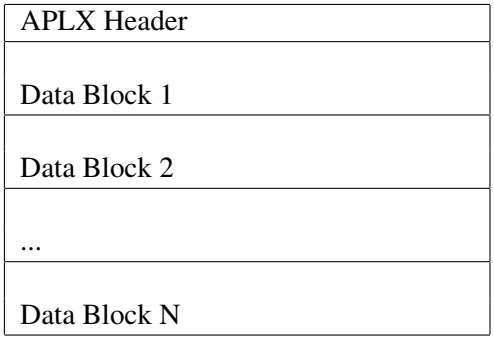


Figure 2.17: Typical APLX file structure.

rently the toolchain resorts to using the CMD_WRITE to upload the binary code to a known location and then instructs the Monitor processor using another SCP command to inform the selected cores that they can start executing. The current implementation of this Monitor processor command simply sends a CMD_APLX to the selected Application cores. The latter command causes the kernel to process an APLX file that has been loaded to memory beforehand, in this case, through the use of the CMD_WRITE.

The APLX format is a simple way of loading applications on to a SpiNNaker chip. Its role is similar to the Executable and Linkable Format (ELF) that is used for executables, object code and shared libraries on Unix systems. The APLX file format allows the loading and execution of C programs that need to be *scatter loaded*, i.e., place parts of the executable in different addresses. There are 3 parts of memory that must be initialised for a typical C program. The first one is the program binary code which must be placed in a executable address, in the SpiNNaker's case this means in the ITCM, usually at the address 0x0. The second part is static or global variable which were initialised with a value different from zero. These are usually placed at the DTCM. The third part is the rest of the global variable which must be initialised to zero as stated in the C standard [ISO99]. These are also usually placed at the DTCM.

The APLX file has an header block that determines how the data is to be loaded into the SpiNNaker's memory. A typical file has a structure like the one presented in Figure 2.17, an APLX header block followed by one or more data blocks [Tem11b].

The structure of the header block is very simple as it is a set of commands, each command is 4 words long. A command has 4 fields, each one word long, the first is command identifier and the following 3 are possible arguments that even if not used are still present, in the Figure 2.9.

Currently, there are 4 commands defined and a marker for the end of the header block. These are :

- APLX_ACOPY
- APLX_RCOPY
- APLX_FILL
- APLX_EXEC

APLX Command 1	Command identifier
APLX Command 2	Argument 1
...	Argument 2
APLX Command N	Argument 3

Table 2.9: APLX header and APLX Command structure

- APLX_END

The APLX_ACOPY command is responsible for copying data from an absolute address to another. The arguments are the source and destination addresses and the length of the data block to be copied. The length is specified in bytes but the operation is done word by word so the length is rounded to a multiple of 4 bytes, currently the implementation rounds it up to a multiple of 32 bytes.

The APLX_RCOPY command is very similar to the APLX_ACOPY command except the source address argument is a relative to the start of this APLX command block. The destination address is still an absolute value and the copy behaviour is still the same.

The APLX_FILL command sets a memory section to specified value in a argument. Just like the previous commands there is a length in bytes that is rounded up to a multiple of 32 bytes.

The APLX_EXEC command is used to start execution at an address specified in the first argument. This address is copied to an ARM register and the BLX instruction is used to branch. If the code being run preserves the link register on entry, then it is possible to return to the APLX loader and continue to other blocks.

The APLX_END is a marker that is used to limit the APLX header which means that after this marker, the loading process is complete. If there is a non returning APLX_EXEC command before, this marker can be omitted since the loader will not continue execution. All the arguments are ignored and unspecified for this *command*. For more details on all these commands, the reader should refer to the application note [Tem11b].

Taking in consideration the requirements of a C program initialization, it is clear that these commands are sufficient since they allow for data to be copied and initialised as it is necessary. Therefore an APLX file with a C program will usually have the header block and two data blocks, one for the program code and the other for global variables that have been explicitly initialised. The header will typically contain 4 commands, two APLX_RCOPY for each of the data blocks; a APLX_FILL for the rest of the global variable which are usually placed together in memory and finally a APLX_EXEC to start execution.

As a final note, it is possible to create self-extracting APLX files. Usually the SpiNNaker kernel takes cares of the loading process but it is possible to prepend unpacking code to the file and process the file by branching to the start of the prepended file.

2.2.3 SpiNNaker Control & Monitor Program (SC&MP)

SC&MP is the kernel that runs on the Monitor Processor and it is responsible for all management tasks. During the booting process, it initializes several structures it uses for the management tasks. It also sets up the timer to run every millisecond and test the interchip link to probe how many active neighbours has using peek packets which do not require processor intervention.

After all the initialisations procedures, it is mainly responsible for routing of the SDP packets which include their transmission to other spinnaker chips using the SDP over P2P protocol or to the application cores. It also implements a larger amount of SCP commands when compared to SARK as for instance the IPTag command which allows the creation and removal of permanent IPTags or commands that allow the manipulation of the interchip link interfaces. It also processes SpiNNaker packets that it receives, specially P2P packets if they carry SDP data or NN packets which are used during the bootstrapping procedures to propagate and set the P2P addresses of all nodes in the grid as well as the P2P routing tables.

2.2.4 SpiNNaker Application Runtime Kernel (SARK)

SARK is a very low-level kernel that runs on the Application Cores. Its source code is provided to the Application developers since it is linked together with the application code written in C during the compilation process. It is mainly responsible for two functions:

- Initialization of the ARM core, setting up the stacks, and some peripherals like for example the timer.
- Providing low level routines for CPU control and hardware management, such as enabling and disabling interrupts, memory manipulation and management, random number generation, SDP messaging which is also used to communicate with the Monitor Processor, packet transmission and event management. The low levels routines that SARK provides aim to substitute parts of the C standard library which are usually provided by operating systems like memory allocation and random number generation. The rest of the routines are specific to the SpiNNaker operation like the communication or environment routines.

The SpiNNaker programming model is event driven, the processor is in a low power state waiting for interrupts that signal events. There are currently 5 available events to the applications:

- MC packet received which is triggered by the successful reception of a multicast packet;
- DMA transfer completed which is triggered by the successful completion of DMA transfer;
- Timer tick which is triggered by the passage of a previously specified period of time;
- SDP packet received which is triggered by the reception of a SDP packet;
- User event which is triggered by a software interrupt.

Event	First Argument	Second Argument
MC packet received	Key	Payload
DMA transfer done	Transfer ID	Tag
Timer Tick	Simulation time	None
SDP packet received	Pointer to mailbox	Destination Port
User event	Argument 0	Argument 1

Table 2.10: Event Callbacks Arguments [Gro11a].

Figure 2.18 shows the dispatch model for the event callback which the application registers for each event with a certain priority. There are two main types of callbacks, non-queueable and queueable. When the relevant event occurs, the scheduler executes the callback immediately in case of a non-queueable callback or places it in a queue according to its priority in case of a queueable callback. After the scheduler finished, control returns to the dispatcher which will execute all the queueable callback in the queue according to their priority. A non-queueable callback may pre-empt a queueable callback at any time since when the corresponding event occurs control will be returned to the scheduler which will execute the non-queueable callback immediately. When there are no more pending callbacks in the queues, the dispatcher will enter a low power state mode until there is an event. There is also a preeminent callback which is selected by the application developers to have the highest priority and it is capable of pre-empting both non-queueable and queueable callbacks. The current implementation associates this preeminent callback with a fast interrupt request (FIQ) while the non-queueable callbacks are associated with normal interrupt requests (IRQ). The callbacks are functions with two unsigned integer arguments and no return value, the meaning of the arguments for each of the available callback is presented in Table 2.10.

The non-queueable callbacks are available to the application programmer as a method of pre-empting long running tasks with high priority tasks. They should be used sparingly since they are only pre-empted by the preeminent callback and may starve the queueable callbacks. Critical sections are also available to prevent pre-emption during access to shared resources, there are routines to disable IRQs or to disable both FIQs and IRQs. Disabling both interrupts requests may lead to priority inversion. Further details about the location of the source code and the API are presented in appendix D.

2.2.5 Toolchain

The toolchain used for SpiNNaker machines depends on the chosen programming language since C, ARM Assembly or Python can be used [Gro11a]. Either way, some tools like *ybug* are common for both approaches since they are used to upload code and data to the SpiNNaker systems.

Ybug and Tubotron

In order to deploy applications to a SpiNNaker system a custom program must be used, named *ybug*. *ybug* is a computer program that runs on the host system and provides an interactive text-

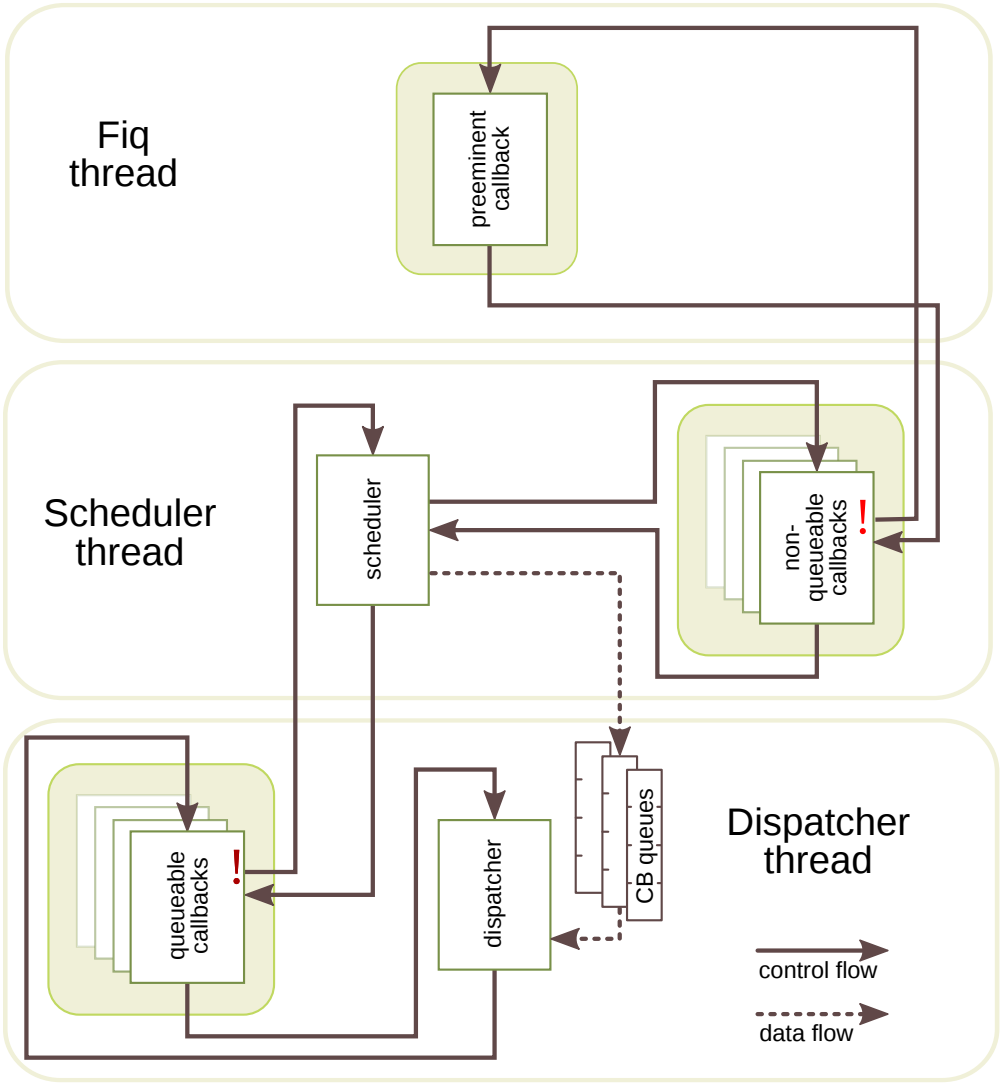


Figure 2.18: SpiNNaker programming framework [Gro11a].

based interface to a SpiNNaker system. It uses UDP datagrams to communicate with the SpiNNaker system so it can be used to control remote systems. Since the bootstrapping process is also based on the UDP protocol, *ybug* is also responsible for initializing the system.

ybug is written in Perl and it is usually run under Linux, it started as a simple program meant to communicate with early SpiNNaker systems but it has not been superseded so far. It supports a large amount of commands that allow the SpiNNaker system to be bootstrapped, loaded with data and binary code, and to be debugged. The debugging capabilities are based on low-level features that have the ability to inspect and change memory in any SpiNNaker chip that has been bootstrapped.

TuboTron is a tool that display debug information outputted by a SpiNNaker chip. SARK has some routines that allows printing to a debug buffer that are sent to a specific UDP port that Tubotron is listening on. It displays a separate window for each core.

These two tools are essential for the use of a SpiNNaker system.

Low level Programming

For most low level programming, C will probably be the language of choice with small sections of assembly code for critical performance areas [Tem11a]. The recommended and tested compilers are the RVDS 4.0 release of the ARM RealView Development System and GCC 4.5.2 from Code Sourcery, available at http://www.codesourcery.com/sgpp/lite/arm/portal/release1802.

The SARK source code comes with an example folder where there is a GNU Make Makefile for both of these compilers. This compilation process will take the source code provided and SARK and produce an APLX file suitable to be uploaded to a SpiNNaker system.

High level Programming

Besides C programming, other high level technologies are also available. Figure 2.19 shows the development route for a neural networks simulation. The model can de described using standard simulation languages/frameworks, such as PyNN [DBE+09] or Lens [Roh98]. The Partitioning And Configuration MANager (PACMAN) is then responsible for transforming the high-level representation into a physical on-chip implementation, the details on this process through which this conversion happens are described in [Gro11a].

2.3 Summary

In this section, the hardware and basic software architectures of the SpiNNaker system were introduced, with the purpose of laying the foundations for the subsequent chapters. In conclusion, it may be inferred that the SpiNNaker system is a fully equipped with specific means to operate and host applications developed in C.

However the SpiNNaker system is a fairly complex system which is still in constant evolution, specially the APIs that SARK provides which are still the target of research in order to improve and facilitate the deployment of application.

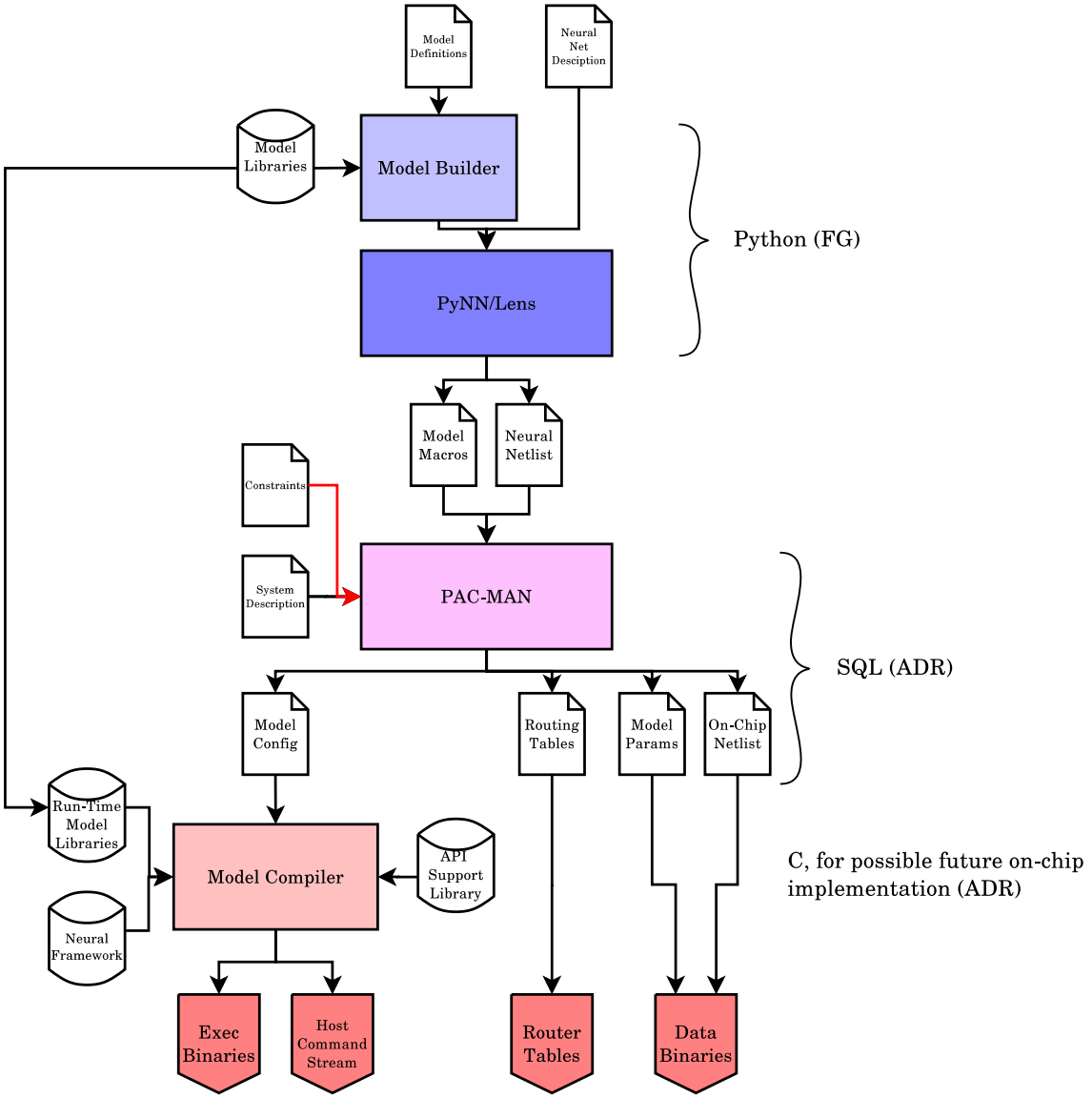


Figure 2.19: SpiNNaker neural networks simulation development route [Gro11a].

Chapter 3

SpiNNaker Chip Computing Module

The low power consumption and small size of a SpiNNaker chip makes it an interesting candidate to have on-board of a small robot as a powerful computing resource. The projected solution should be as small as possible, it should also allow to be connected to sensors and actuators through a serial connection. In order to solve the stated problem, a solution based on custom hardware and software was devised.

A detailed description of the SpiNNaker Chip Computing Module will be presented in this chapter, starting by the developed hardware and continuing through the software layers present on the developed board and on the associated workstation. The order of description follows roughly the time line of the development work.

3.1 General Architecture

Traditional SpiNNaker systems rely on the host system to provide the input data which limits the possibilities of interfacing the SpiNNaker chip with the external environment. In order to overcome this, a microcontroller was added to be responsible for communicating with the outside world. Figure 3.1 presents the general architecture for the developed solution which involves the design of a PCB with a single SpiNNaker chip and a microcontroller, the software to drive the microcontroller and an application that runs on the host system to communicate with the board. Since one of the main goals of this solution is its portability, there was an effort to keep the hardware as simple as possible having only the necessary components to achieve the project's goals.

There were several requirements that determined the usage of this architecture. Backwards compatibility with the normal tools used for the bigger SpiNNaker machines, *ybug* and *tubotron*, was an important aspect that lead to the creation of the Host system application. Having this compatibility allows the reuse of know-how related to these tools and increases the value of the developed system for future users.

The addition of the microcontroller allowed for the replacement of the Ethernet connection with a simpler albeit slower communication protocol, RS-232, at a baud rate of 12 MBps which

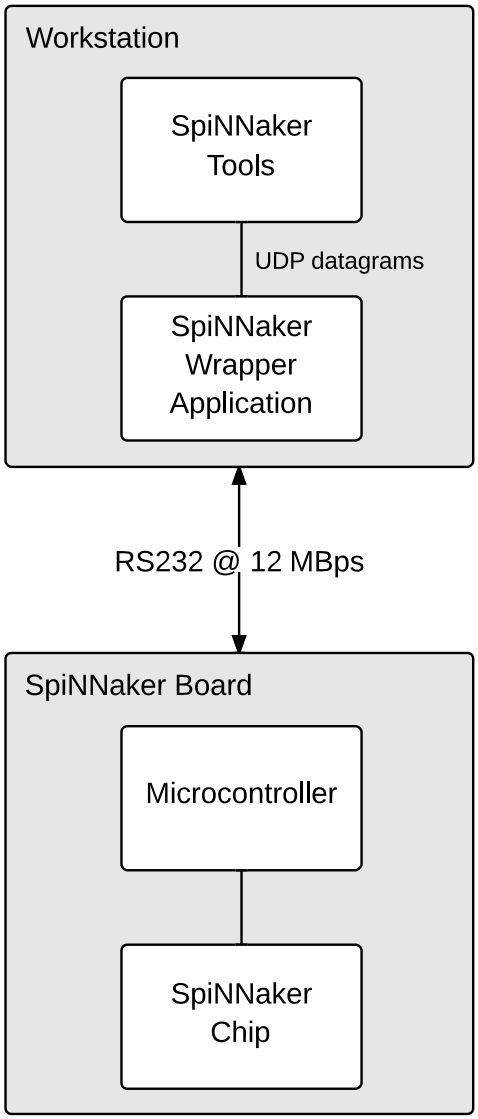


Figure 3.1: General Architecture of the developed solution.

translates into 1.2 Megabytes/s each way. This microcontroller is also essential for booting the SpiNNaker chip without a host system, as it can store the image that was previously transmitted. The microcontroller and the application in the workstation work together to pretend they are another SpiNNaker chip which is connected to the Ethernet, having their own P2P address and position in the grid.

3.2 Hardware

The first stage of development involved the preparation and design of the board with the single SpiNNaker chip and the chosen microcontroller. Figure 3.2 shows a possible diagram connection

between the two chips using the Ethernet capabilities of the SpiNNaker and by selecting a microcontroller with similar Ethernet capabilities. The SerialROM chip and the external PHYceiver are necessary in order for the SpiNNaker to activate its Ethernet connection, which would mean that another PHYceiver would be needed since there are few microcontrollers with integrated PHYceivers. A direct connection was also considered between the two Media Access Control (MAC) interfaces according to [SMS07] but this approach raises other design considerations that would have been troublesome to test since the turnout time for the production of a board was around three weeks, disregarding the fact that during the first stage of booting the SpiNNaker chip actively checks for the presence of the PHYceiver. This could be overcomed through the use of the code loading capabilities with the SerialROM chip but the gains from this effort would be null. Taking this approach would decrease the amount of work needed from the software's point of view, since it would only be required to copy the UDP datagrams through the serial port and then transmit them using the microncontroller's Ethernet connection, but it would greatly increase the effort from the hardware perspective. This solution would also come with increased power consumption since typical low power PHYceivers consume around 400 mW meaning that having two of these chips would translate to a consumption similar to the SpiNNaker chip itself. This consequence alone would cause failure in one of the core principals guidelines of the project, low power consumption, so this solution was not appropriated and it was abandoned.

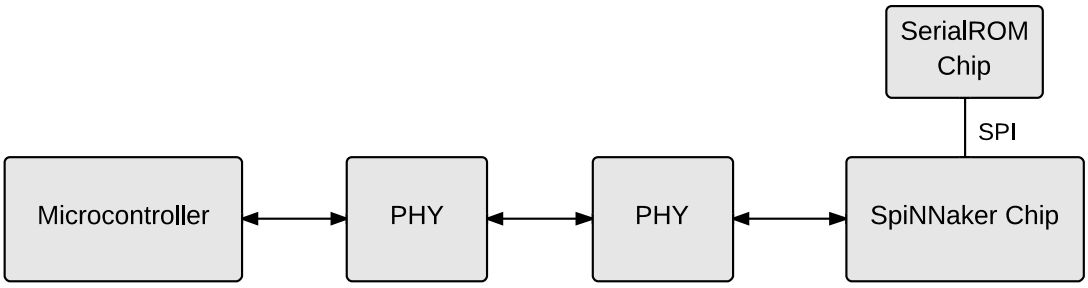


Figure 3.2: Possible connection between the microcontroller and the SpiNNaker chip.

Having discarded this previous approach, the selected one was to connect one of the six inter-chip connections present in the SpiNNaker chip to GPIO pins on the microcontroller and implement the protocol on software. By selecting this way, the hardware design was simplified with a significant increase in the complexity of the software design which is preferable since the testing loop is much faster when compared to hardware testing. Additionally, this also demands that the selected microcontroller has a good performance since it will be necessary to implement a communication protocol which is implemented in hardware from the SpiNNaker side, and it would be necessary to reduce as much as possible stalls that the microcontroller might introduce.

For the design of the schematics and the layout of the board, an Electronic Design Automation (EDA) software utility was used. There are multiple offerings in the market from a variety of companies such as Synopsys, Cadence or Mentor Graphics whose products are fairly complete although complex. The selected tool was the Easily Applicable Graphical Layout Editor also

known as EAGLE. EAGLE is a simpler tool when compared to proposals from larger companies but it is fairly user friendly and widely used in the hobbyist community thanks to its friendly freeware licensing for non-profit applications. It is also available for other operating systems other than Windows such as Mac OS X or Linux which is uncommon among the proprietary EDA applications.

In order to use a PCB layout application, the footprint data for any used component must be present. For simple missing components with few pads, the design of this footprint is viable and it is not too time consuming. On the other hand, the SpiNNaker chip has a 300 pin Ball Grid Array (BGA) package which besides being troublesome to assemble and solder, makes the manual construction of its footprint dangerous because any errors at this stage would be difficult to correct at a later stage without a significant time expenditure. To overcome this problem, a computer program named AutoBGA [CV11] was used. This application allows the generation of footprint data through the use of a image captured from the package drawing present in the data sheet. The parameters needed for this generation, depicted in Figure 3.3, are all fairly simple to measure and although the automatic recognition of the pads' locations is not very reliable, the user interface allows the user to place the missing pads in a grid.

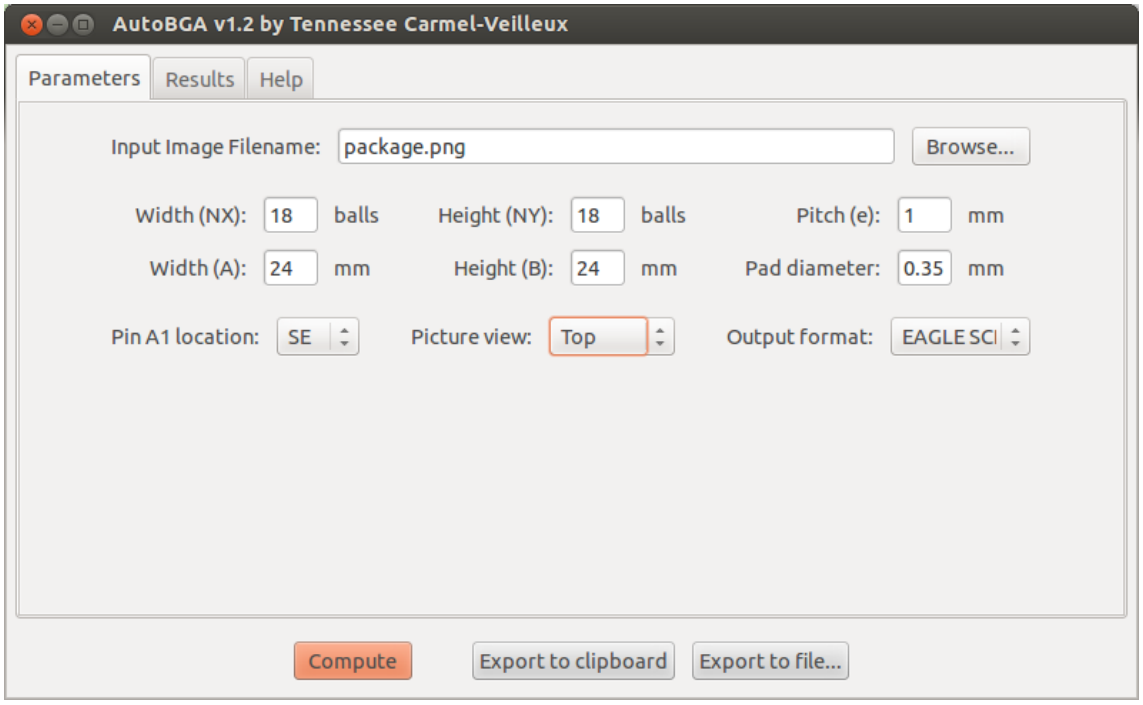


Figure 3.3: AutoBGA user interface for the parameter configuration.

For other components like for instance, capacitors and inductors, the footprint data was drawn using the tool that Eagles provides for the creation of new components. During the development of the hardware, a library with several new components was created to be used on the schematics and on the PCB layout.

3.2.1 Components Selection

Having decided that the connection between the microcontroller and the SpiNNaker chip would be fairly simple as it would be a direct connection between GPIO pins and one of the interchip I/O links, the rest of the board capabilities can be defined in order to choose the components to be used.

A critical component to be chosen was the microcontroller. Up until now, a few generic requirements had been determined: low power consumption, fast performance, enough GPIO pins to drive a 2-of-7 connection and a serial port. The ability to quickly interface with the SpiNNaker chip was essential so another requirement was added, a dual core chip so that output and specially input could be handled in parallel for maximum throughput. A lot of attention was paid to this requirement due to the fact the router on the SpiNNaker chip may drop packets if the receiving chip seems blocked or slow even though the communication protocol relies on an asynchronous self-timed circuit.

The NXP LPC4337 [NXP13a] is a dual core microcontroller that complies with all the requirements, it has a 32 bit ARM Cortex M4 core running up to 204 MHz and a 32 bit ARM Cortex M0 core that can run up to 204 Mhz. The chosen configuration also uses two separate banks of flash, each with 512 Kilobytes, and RAM memory, one with 64 and another with 40 Kilobytes.

The SpiNNaker chip requires two different voltage supplies, one at 1.2V capable of 1A and another at 1.8V capable of 250mA. The complete board was to be fed a 3.7V Lithium cell battery which means that at least two step-down converters would be needed. The initial choice to be made was between a linear regulator and a buck converter for feeding the supply needs of the SpiNNaker chip. A linear regulator is generally a good choice when the difference between the output voltage and the input voltage is small since it dissipates the excess energy as heat.

$$\begin{aligned} \text{Dissipated power at 1.2V} &= (3.7\text{V} - 1.2\text{V}) \times 1\text{A} = 2.5\text{W} \\ \text{Dissipated power at 1.8V} &= (3.7\text{V} - 1.8\text{V}) \times 0.25\text{A} = 0.475\text{W} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \text{Efficiency at 1.2V} &= \frac{1.2\text{V} \times 1\text{A}}{3.7\text{V} \times 1\text{A}} \approx 32\% \\ \text{Efficiency at 1.8V} &= \frac{1.8\text{V} \times 0.25\text{A}}{3.7\text{V} \times 0.25\text{A}} \approx 49\% \end{aligned} \tag{3.2}$$

As it can be seen in the calculation above, linear regulators are extremely inefficient for this use case in absolute terms and also relatively to the buck converter since it can easily obtain efficiencies in the order of 80-90%. There are some trade-offs which include the added cost and space but these are worth it when compared to the efficiency gains. Additionally the increased analog noise that these converters might introduce is not relevant since the rest of the circuit is based on digital signals and this noise can be minimized with careful layout and a correct selection of the output filter. Considering all these requirements, two buck converters from Texas

Instruments were chosen the TPS62202 [Ins06] for the 1.8V rail and the LM2852 [Ins13] for the 1.2V rail. For both, the values of the output inductor and the output capacitor were well defined in their data sheets and their recommendations were followed.

The final major component left to be chosen is the power regulator for the microcontroller, the LPC4337 requires a 3.3V supply capable of 200mA.

$$\begin{aligned} \text{Dissipated power at 3.3V} &= (3.7\text{V} - 3.3\text{V}) \times 0.2\text{A} = 0.08\text{W} \\ \text{Efficiency at 3.3V} &= \frac{3.3\text{V} \times 0.20\text{A}}{3.7\text{V} \times 0.20\text{A}} \approx 89\% \end{aligned} \tag{3.3}$$

In this specific case the efficiency of a linear regulator is on the same level of a buck converter although this alternative is smaller and cheaper. The chosen component was the TPS73033 [Ins11] that can output 3.3V at 200mA.

3.2.2 Power Dissipation

A typical concern when designing a PCB with several power supplies and regulators is the power dissipation present in the various regulators that are in place and which may require a heat sink in order to perform within its operational ratings. The maximum power dissipation limit is determined using the equation 3.4:

$$P_{D(max)} = \frac{T_{J(max)} - T_A}{R_{\Theta JA}} \tag{3.4}$$

Where:

- $T_{J(max)}$ is the maximum allowable junction temperature which is usually considered to be 125° C;
- T_A is the ambient temperature which will be considered to be at 85° which is a very conservative estimate;
- $R_{\Theta JA}$ is the thermal resistance junction-to-ambient for the package which depends on the chip.

While the power dissipation is determined using the equation 3.5:

$$P_D = V_O \times I_O \times \frac{1-\eta}{\eta} \tag{3.5}$$

Where:

- V_O is the output voltage;
- I_O is the output current;
- η is the efficiency of the regulator at previous output values.

Regulator	Dissipated Power (W)	Maximum Power Dissipation (85° C)
TPS73033	0.080	0.225
TPS62202	0.061	0.160
LM2852	0.514	1.053

Table 3.1: Power dissipation for the various regulators.

The results from these calculations are compiled in a Table 3.1, the values used were retrieved from the regulators' data sheets. As it can be seen from the table, there is plenty of headroom for each regulator in terms of power dissipation meaning that a heat sink is not necessary even though these regulators were used in a compact design since the estimate for the ambient temperature was very conservative for its future usage scenario.

3.2.3 Layout Concerns

Even though, the entire circuit relies on the digital signals there were still several guidelines that were followed during the layout process in order to improve the signal quality and to increase errors margins [Zum08].

Ground Plane

A ground plane is the use of one layer of a multilayer PCB, or in this case one side since it was a double-sided one, as a continuous sheet of copper which is then used as ground. The large amount of metal will have a resistance as low as possible and also an inductance as low as possible because its large flattened conductor pattern. With these features, it can offer the best possible conduction which minimizes voltage differences across reference points.

Decoupling

The SpiNNaker chip has highly variable current demands specially during the self-test phase that occurs right after power-on. In order to account for behavior several decoupling capacitors have been in the neighbourhood of the multiple supply pins to which were connected through wide traces to minimize the line inductance and resistance [Inc09]. The values for these decoupling capacitors were obtained from the collaboration with researchers from the University of Manchester.

3.2.4 PCB Test Board

Before committing to the final design, a simpler version without the SpiNNaker chip was developed in order to start developing the software without having to debug hardware problems. This test version featured a 34 pin connector that was used to drive the 102 SpiNNaker machine as seen in Figure 2.10. This connector has sixteen pins that connect to one of the inter-chip links of a SpiNNaker chip while the rest of the pins are connected to ground. This test board allowed the software to be fairly mature before building the larger version.

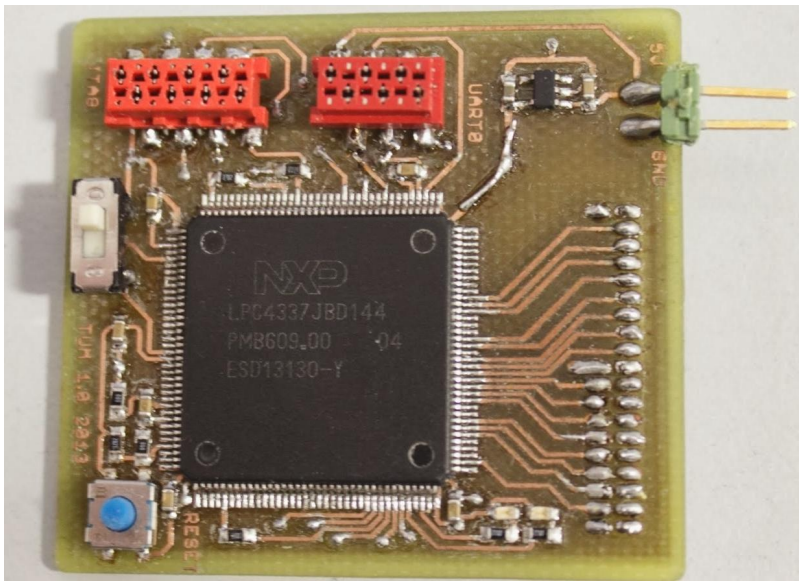


Figure 3.4: PCB Test Board.

3.2.5 PCB Final Board

The final design added the missing SpiNNaker chip and the power supplies circuits that the SpiNNaker requires. Figure 3.5 shows a picture of the final board with all the components, except for the 34 pin connector, soldered.

3.2.6 SpiNNaker Extension Board

The use of the 34 pin connector allows the SpiNNaker Computing Module to be extended with the SpiNNaker 102 machine but using this larger board defeats the purpose of having a small board. In order to overcome this limitation, an extension board with a single SpiNNaker chip with two 34 pin connectors which makes it possible to build a chain with up to 256 chips. Appendix A has the schematic and the layout for this board, and the other two previously mentioned.

3.3 Microcontroller Firmware

This layer of the developed solution is the one responsible for communicating with the SpiNNaker chip, which includes the responsibility of implementing the 2-of-7 protocol and the SDP over P2P protocol, and communicating with the Host system. The Figure 3.6 shows a high level view for the architecture used in the microcontroller software. There are two core, one ARM Cortex M0 which is a low power simple 32 bit core and one ARM Cortex M4 which is a much more powerful core with its own Floating Point Unit (FPU). The M0 core was used exclusive to process the input from the SpiNNaker while the M4 core was in charge for processing the packets received, communicating with the host system and outputting packets to the SpiNNaker. This cooperation is possible thanks to the access that both cores have to most of the peripherals and memory banks

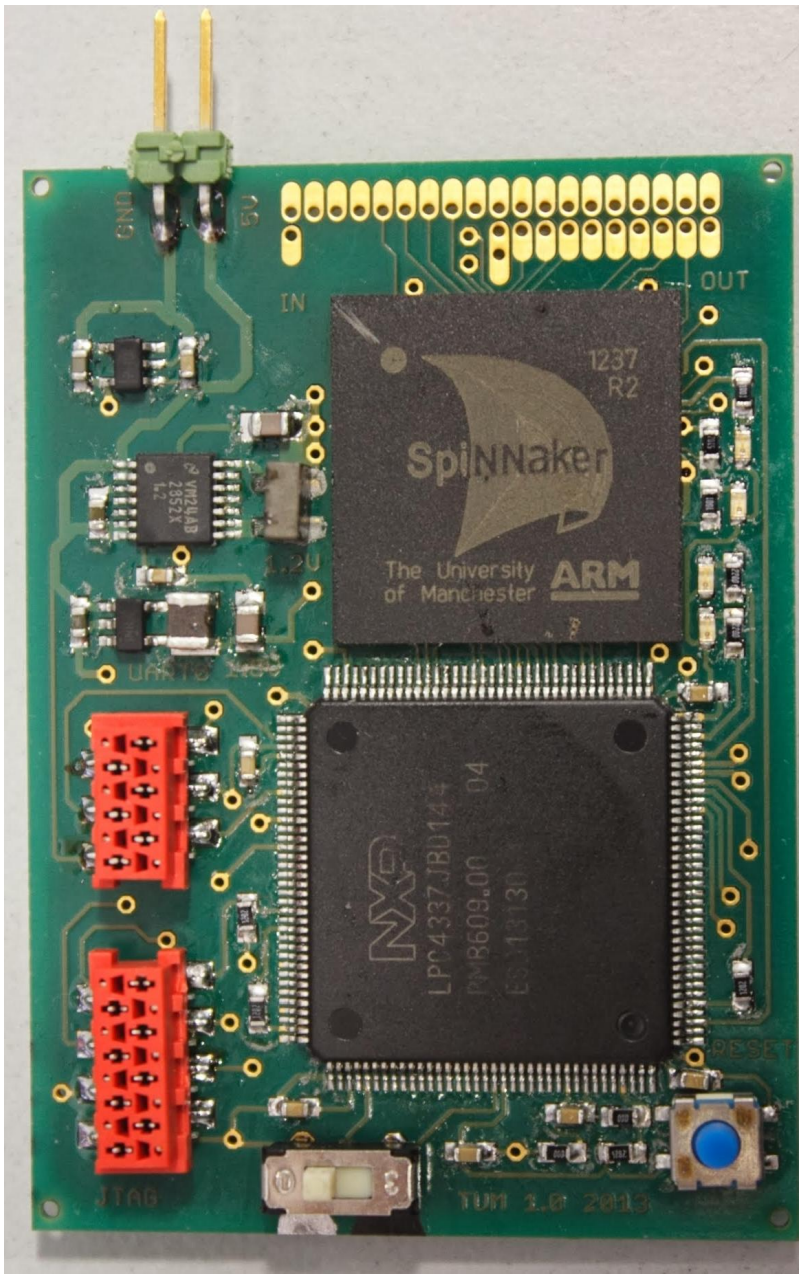


Figure 3.5: The SpiNNaker Computing Module.

through the use of a matrix connection layer present in the LPC4337. Additionally, each core runs its own binary code with no shared code, the compilation process for each core was separate and there are placed in different flash banks for increased performance. The main reason to avoid sharing binary code between the two cores comes from the fact that the M4 core has a larger instruction set that the M0 does not support which means that any shared binary code would not take advantage of the extended features that the Cortex M4 has.

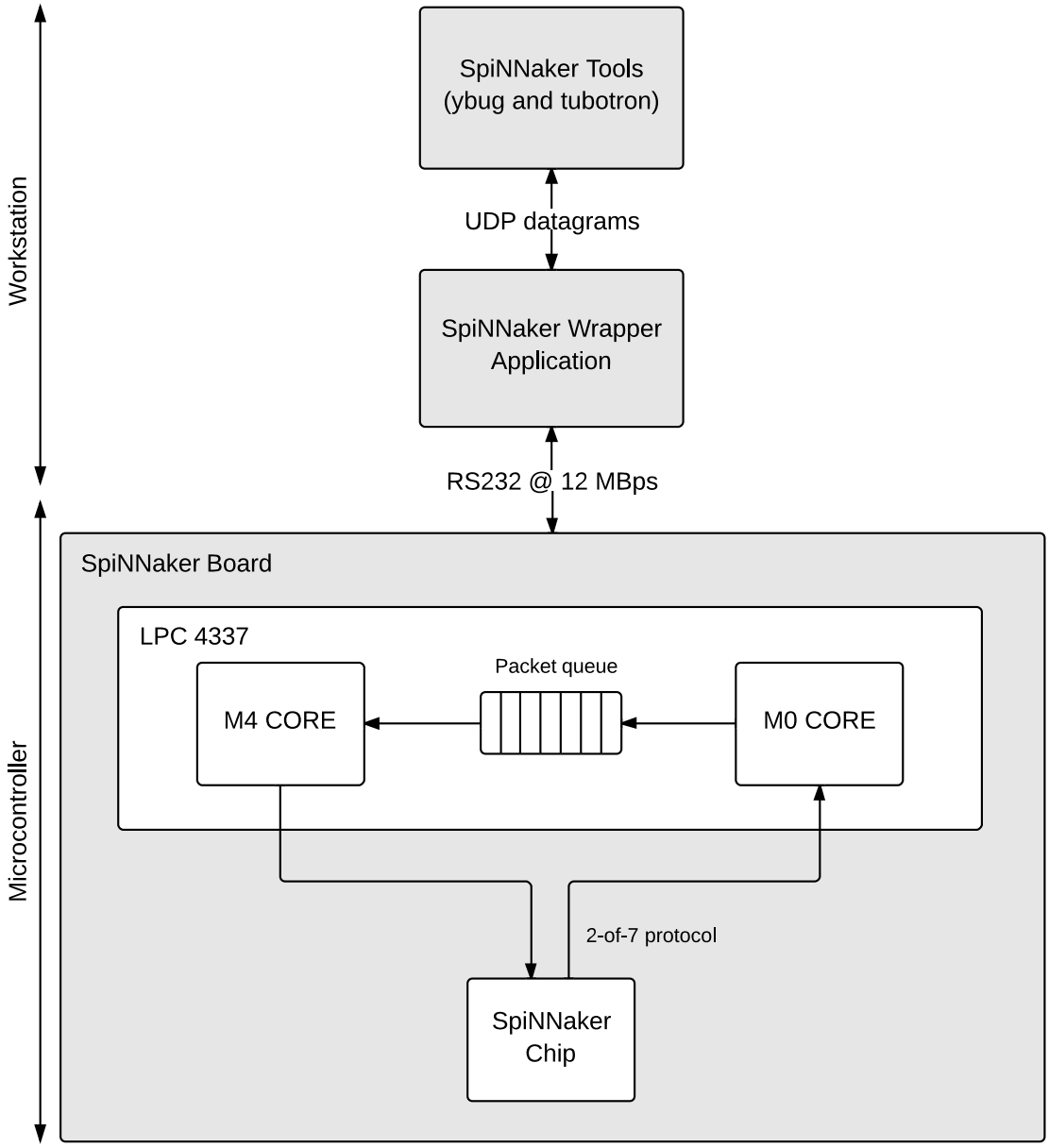


Figure 3.6: Microcontroller Firmware Architecture.

3.3.1 M0 Core

The 2-of-7 protocol relies on two unidirectional links, each with seven data wires and one acknowledgement wire. A four bit symbol is transmitted by toggling two of the data wires, the symbol coding is presented in the Table 2.3. The receptor should then acknowledge that it has decoded the symbol by toggling the acknowledgement wire. The SpiNNaker chip sends packets with either 72 bits or 40 bits (18 or 10 symbols) if the packet has a payload or not, followed by a End-of-Packet symbol. The M0 core function is simply to received symbols, assemble them into a packet and push that packet into a shared buffer with the M4 core for processing.

Processing Algorithm

Since the communications protocol is based on transitions and it relies on 7 parallel signals, it is not possible to map it in any way to the more standard serial protocols, neither rely on some custom peripherals that the LCP4337 contains that were meant to help encode custom serial protocols which means that all processing must be done in software. The current algorithm is based on a look-up table to decode the symbol and relies on the XOR operation to retrieve the transitions. The use of a look-up table is possible due to the fact the 7 wire transitions led to 128 different combinations.

During the initial stage, the M0 Core initializes the look-up table with the valid symbols for the correct positions and the rest with a known invalid value. It also reads the GPIO input state for all the seven input wires, saving its state in a local variable, and sets the acknowledgement to signal to high as it is specified [Tem12]. Having completed all the initialization configurations it enters the main processing loop where the first step is to check if there is room in the packet queue. This verification is blocking meaning that the microprocessor will be locked in a tight loop until this condition is verified. It then proceeds to read the current state of the GPIOs pins and check if there were any changes, if there were not it will repeat this step until there are. If there were changes it retrieves the XOR between the current and previous status, and using this result it fetches the symbol from the look-up table. If the symbol is valid it is stored in the correct position in the packet, if it is invalid it is dropped and the system stays on the same step. Finally it toggles the acknowledgement wire and it updates a local variable with the current state of the GPIO input lines. This process repeats until a full packet is received from which the system jumps to the initial verification of space in the packet queue. In order to prevent needlessly copies the received packet is written directly to the queue since the M4 core will not read the current writing position until it has progressed to the next packet.

The coding style adopted to implement this algorithm can be described as *assembly like* C programming since it features heavy usage of labels and *goto* which are usually looked down upon in standard coding practices. This style of coding came from the need to help the compiler improve its code generation. Most of the improvements to the code were made by analysing the compiler's output and reworking areas that could be improved.

Packet Queue and Memory Layout

An important element of this processing algorithm is the packet queue which must be shared with the two cores. NXP provides a number of libraries and Application Programmable Interfaces (APIs) for interprocessor communication (IPC)[NXP12] after a careful analysis of its implementation, there were discarded since they had too much overhead for the simple communication model that it was required. As a result special care was given to the memory layout of the system. The LPC4337 has two RAM banks with different sizes, one with 64 Kilobytes and another with 40 Kilobytes. Traditionally the first would be exclusive to the M4 core and the second for the M0 core and indeed this was the solution employed for this case.

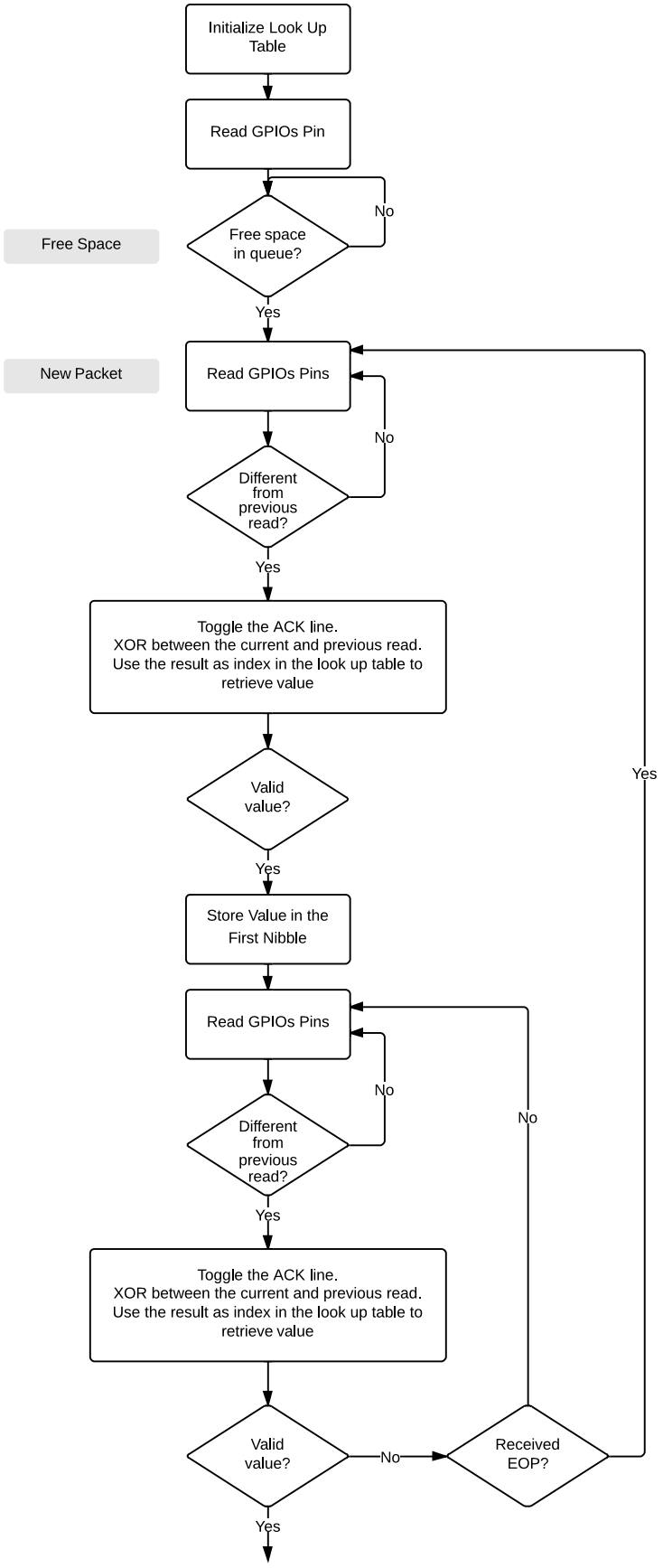


Figure 3.7: Packet Input Reading Algorithm.

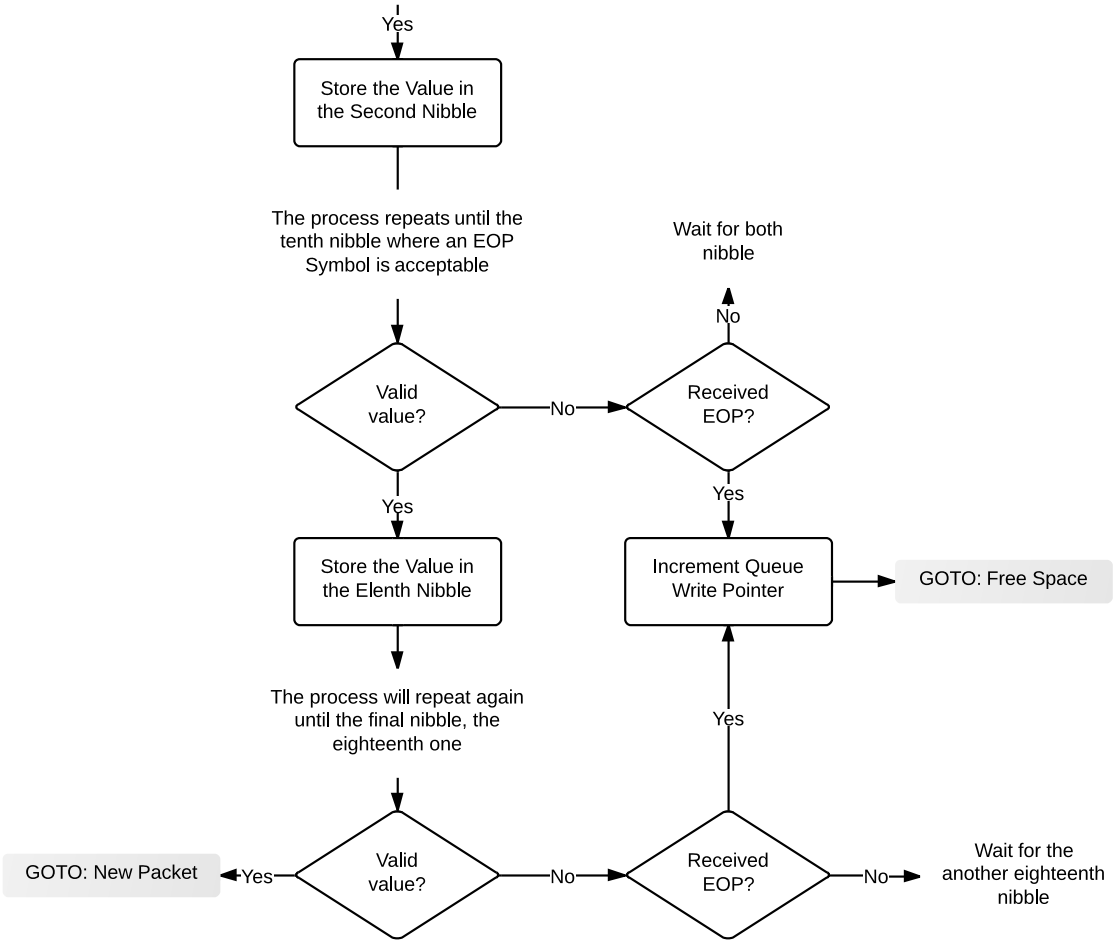


Figure 3.6: Packet Input Reading Algorithm.

In order to maximize the queue's length, the majority of the 40K bank was reserved for its use. Each packet takes at most nine bytes, one for the control header, four for the routing key and four optional more for the payload. The queue is simply an array with the largest amount of packets possible with the constraint that the number of elements must be a power of two in order to do the wrap-around operation using a simple AND operation. The fact each position takes full 9 bytes when four are optional may seem unreasonable but in practice the large majority of packet require a payload. For the 40K bank this lead to 4096 elements which take up 36864 bytes of space. The queue also required two more integers with the current write and read position. These three elements position in memory, the two integers and the array, was hard coded and added to both cores linker file [NXP13c]. With this technique, it was possible to share a structure between the two cores with independent code bases without any overhead at all. The rest of the memory in the 40K bank was used for the binary of the M0 core, for its stack and global variables, while the 64K bank was reserved for the M4 core binary.

3.3.2 M4 Core

The M4 Core is responsible for the rest of the system management which includes various tasks such as processing the packets read by the M0 core, communicating with the host system and transmitting packets to the SpiNNaker. The main processing loop executes a variety of tasks sequentially, meaning that each task never blocks at any point. In order to follow this very important guideline , a variety of state machines were employed in several subsystems. They are four main tasks:

- Host communication task which is responsible for receiving commands and data from the host system and processing it accordingly;
- Packet processing task which is in charge of reading the packets received from the SpiNNaker system and taking actions as required such as moving it to the serial port output buffer or replying with another packet in case of poke packets for example;
- SDP packet handling task that continues the transmission of a SDP packet and checks for any timeouts that may have occurred while receiving or transmitting SDP data;
- Serial output task which simply empties the serial output buffer through the serial port as fast as it is possible.

The following subsections will discuss the implementation of the above tasks.

Host communication task

This tasks only runs if there is a new byte to be processed from the serial port otherwise it will exit immediately. This task is basically a state machine, depicted in Figure 3.7, that has a certain amount of commands and their processing. Currently the commands available are the reception of a boot image, the transmission of a SDP packet, the transmission of a SpiNNaker packet for which each type of packet and whether or not it has a payload is a different command, and a P2P address redefinition command. As soon as the data needed for acting the command is received it will be executed. This means the transmission of the boot image or a SpiNNaker packet will be done at the final step of the state machine processing branch. As for the SDP transmission the initial packet is sent but the rest of the process is left for another tasks since it requires replies from the SpiNNaker chip according the SDP over P2P protocol.

Packet processing task

This task is responsible for looking at the packets received from the SpiNNaker system and act accordingly, it processes a packet at a time to avoid starvation of the other tasks. The rule of thumb while processing these packets is if the firmware does not know how to proceed it will simply copy it to the system output buffer which will then be transmitted to the Host system. In case this output buffer is full, then the packet is not release, i.e., the read position integer is not incremented. This

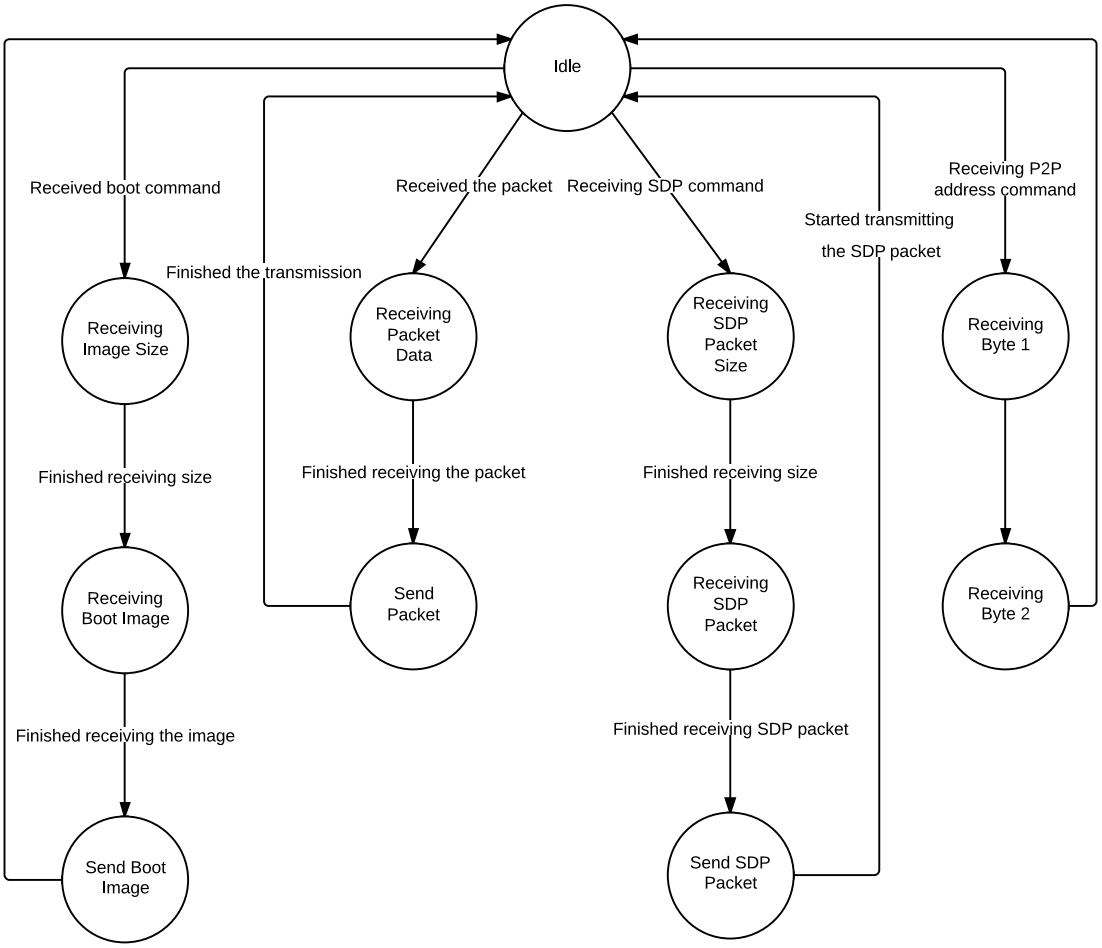


Figure 3.7: Host communication task state machine.

means that if the output buffer is full caused by a burst of unknown packets, processable packets may be in the queue for an unreasonable amount of time depending on the baud rate being used. The solution found to this problem was the usage of a *shadow pointer* which basically is an extra local integer that tracked the position of processed packets and an array that indicated whether or not a packet should be copied to the output buffer. Basically a packet is processed by checking its type and if can trigger any other action such as a reply in case of a poke packet or set a flag for the SDP packet handling tasks. In this case, the packet is not to be transmitted back to the Host system so it can be released immediately. If no recognizable action is identified, then it is marked to be transmitted which depending on the free space available on the output buffer may either occur immediately or in following executions of this task. Nevertheless, packet processing continues one packet at a time if there are any left to be processed although the queue read pointer will only move after a packet has been successfully copied to the output buffer. This technique allows the system to reduce latency without discarding packets during a burst reception and it proved essential to keep the system active during the flood-fill phase which is a specially critical phase with a large amount of packets being received in a short period of time.

SDP packet handling task

This tasks implements the SDP over P2P protocol present in Figure 2.8. The reception implementation is basically the checking and handling of timeouts since the packet processing task handles the packets received atomically, and copying the final SDP packet to the output buffer. It uses the state machine presented in Figure 3.9 to control this behaviour. On the other hand, the bulk of the transmission is handled inside this task leaving only the acknowledgement packets handling to the packet processing task. It features the state machine featured in Figure 3.8 that resembles the several steps of the protocol. The initial step, i.e. the open channel request, the request packet is sent inside of the Host communication task. This task will also check for timeouts and the need for retries until the packet processing task sets a flag allowing the process to continue with the data transmission. This means that each transition in this state machine is regulated by packet processing task with the reception of acknowledgement packets since timeouts or retries will either maintain the same state or completely reset the process and drop the SDP packet.

Serial output task

This tasks is simple responsible for moving bytes from the output buffer to the serial port peripheral as fast as possible.

Packet Transmission routines

The packet transmission routines are the functions responsible for sending a SpiNNaker packet to the SpiNNaker chip. They received as arguments the control header byte, the four bytes routing key and the optional four bytes payload and calculate the parity bit and fill it in the control byte before transmission. The implementation of the nibble transmission was the target of several optimizations and tunings since this is a basic building block of the system. The final implementation relied on capability of the C programming language to have a function array, each different nibble was implemented in a separate function, and the sixteen possibilities were packed in a function array. While transmitting a byte, it was simply a matter of using each nibble as a position for the function array table. The advantage of this approach is that every nibble takes the same time to be transmitted as opposed to the initial implementation while a *0* would be faster than a *15* depending on the order of the *compare* operations. Additionally the GPIO peripheral present in the LPC4337 proved to be essential in improving the performance since it had the capabilities to toggle an output using a single instruction meaning it was not needed to keep track of the current status of the output and act accordingly.

Another optimization that may seem a bit dubious at first look is checking the acknowledgement signal only after the EOP symbol. Since an even number of symbols is transmitted while sending a packet up until the EOP packet symbol, the state of the acknowledgement signal should be the same as it was before transmitting the packet assuming it toggled correctly for every symbol. This allows the check only to be done at the end which should be sufficient to detected errors with the transmission. This improvement proved to be essential in improving the throughput of

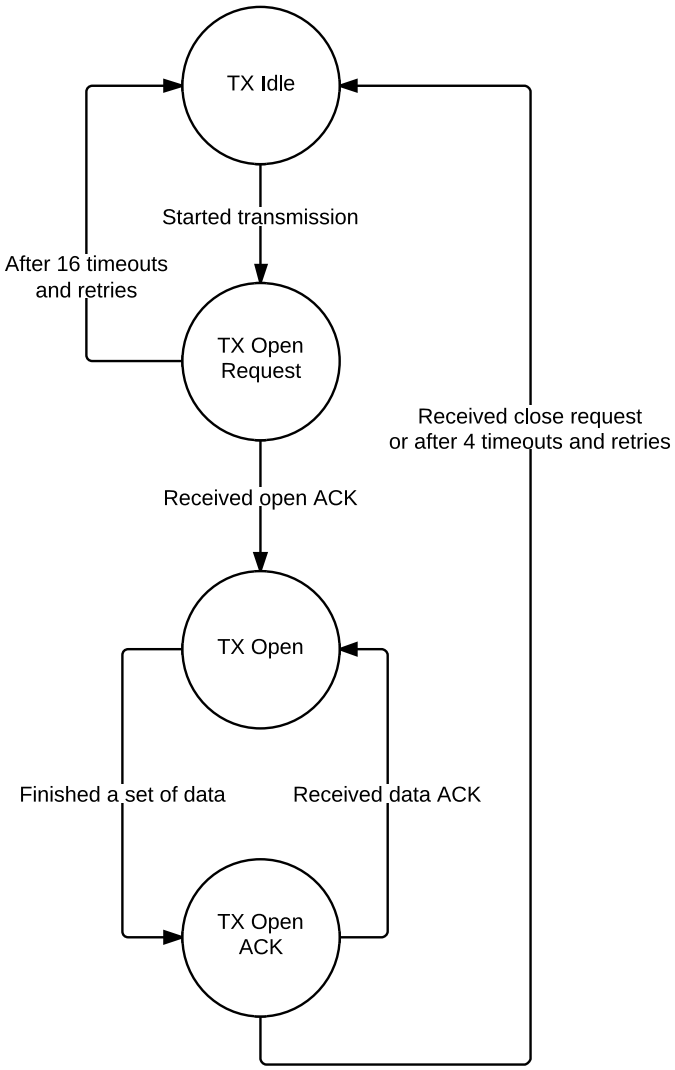


Figure 3.8: SDP Packet Transmission State Machine.

symbol and packet transmission leading to more than 60 % of reduction in the total transmission time.

3.4 Workstation Application

The upper layer of the developed solution is an application that runs on the host system. This application is responsible for trapping UDP packets meant for a standard SpiNNaker system sent by *ybug* and relay them to the PCB using a serial port. This program was written using the Java programming language [DD11] and the RXTX Serial Port Library [Jar06]. The Figure 3.10 presents a high-level architecture of the program.

The application opens an UDP socket on the same ports that the SpiNNaker usually uses, though the user must first select the serial port and baud rate since a connection is only opened if

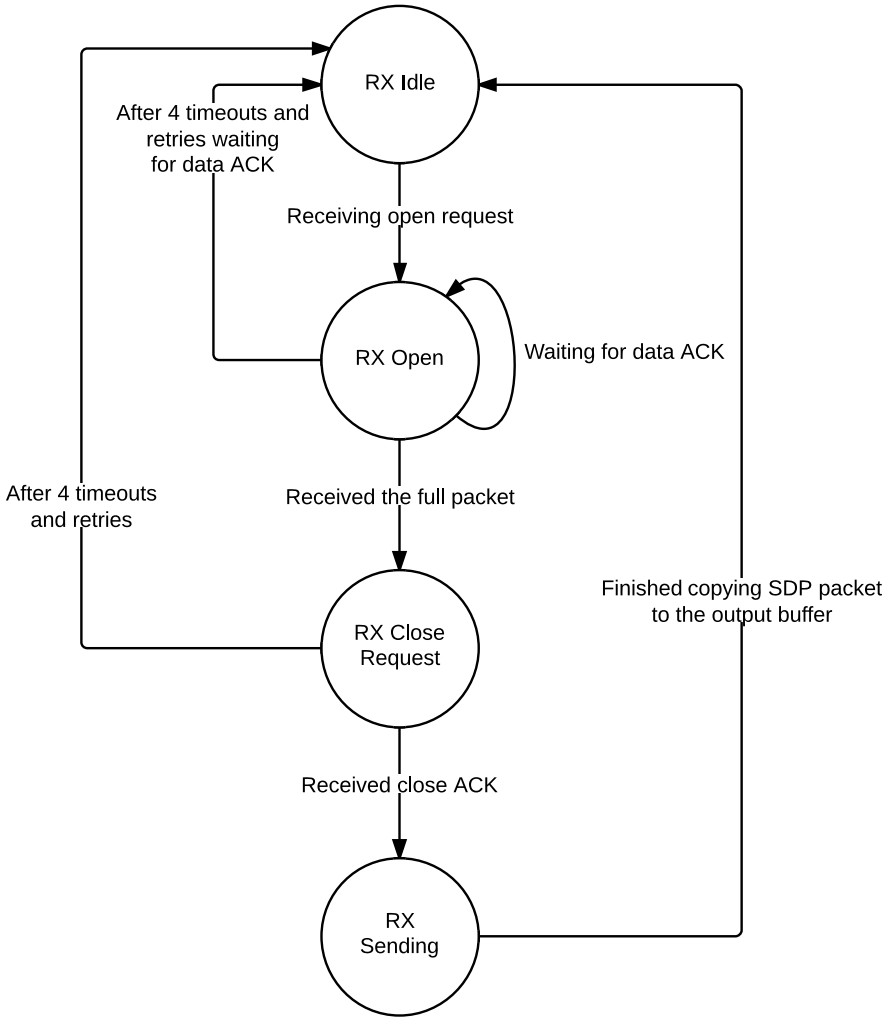


Figure 3.9: SDP Packet Reception State Machine.

the program can relay the data it receives. When a connection is opened, several new threads are launched. These are:

- Boot Image Transcoder which is responsible for capturing UDP datagrams with pieces of the boot image and reassembling it with little endianess before transmitting it to the microcontroller, after finishing the transmission of the boot image it also configures the P2P address of the microcontroller and sets up the P2P grid using NN packets. Additionally it also configures the Router to increase the number of the wait cycles to ensure the microcontroller does not misses packets during the burst that may come from the SDP over P2P protocol.

- SDP Packet Transmitter which takes the SDP packets and places them in a queue to be analysed. Another thread is responsible for removing these packets from the queue and sending them to the microcontroller at a controlled rate in order to prevent rejected packets since there is limited buffering on the microcontroller size as opposed to the workstation

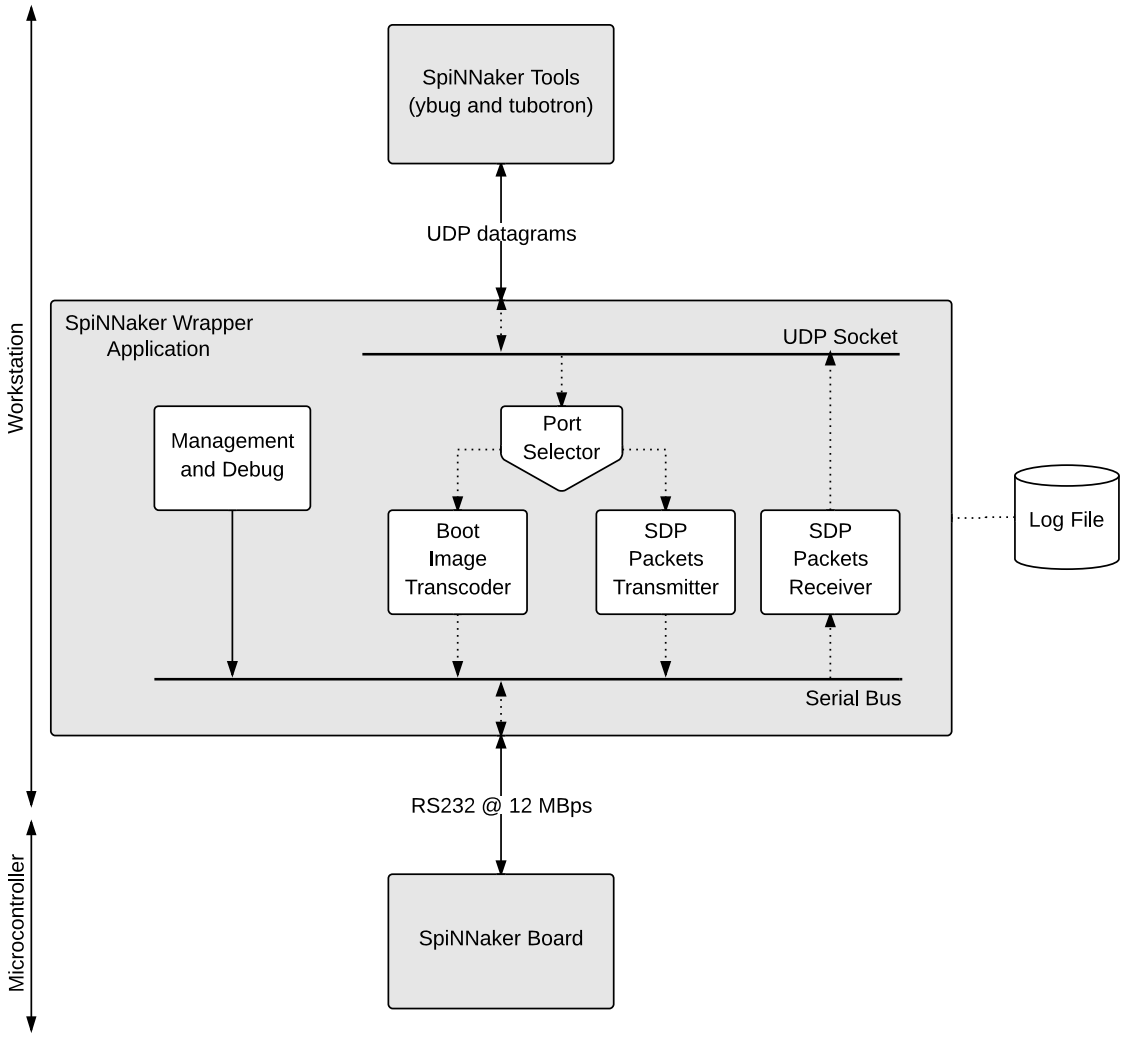


Figure 3.10: SpiNNaker Wrapper Application Architecture.

whose buffering capabilities are much larger. This thread also checks if the SDP packet contains a command which may be directed to the SpiNNaker that the application pretends to be like an IPTag command.

- SDP Packets Receiver which its main objective is to take SDP packets received through the serial port and send them through the UDP socket using the IPTag field present in the SDP packet header. It is also responsible for receiving email packets and printing those in the log in order for the user to analyse their content.

IPTag Commands

The application and the microcontroller are responsible for implementing the communications capabilities of a SpiNNaker chip with an Ethernet connection. In order to achieve this, they have their own P2P address which can be targeted using P2P packets or SDP packets which means that it may also receive SCP commands. Most commands are ignored so that the user using *ybug*

remotely may realise it is communicating with a real SpiNNaker. However, IPTag commands are accepted and replied to because they are essential in the functioning of *Tubotron*, the debug output tool, and the general routing of SDP packets. The implementation in the application mimics the SpiNNaker's kernel implementation but with increased size capabilities since it uses the full length of the field, eight bits, as opposed to the kernel implementation which only uses four bits.

3.4.1 User Interface

The user interface is divided in three main areas, the connection manager, the packet manager, and the log area, as shown in Figure 3.11. The connection manager features the necessary tools for the user to select the serial port and the baud rate for the connection. At any point, the user may disconnect and choose different settings. This interface brings flexibility and portability since the name of the chosen serial port is not hard coded which allows the program to execute flawlessly in other operating systems like Windows or Mac OS X.

The Packet Manager is an utility that was added during development of the microcontroller's firmware and it allowed to test various situations. It allows the user to manually transmit a SpiNNaker packet, by selecting the type and filling in the fields. During development, it was extremely useful to tune some configurations packets that must be sent and to check their real effect by examining the SpiNNaker memory by using *peek* packets.

The Log area shows a restricted amount of log messages that the application generates while running. The text it shows is selectable which the user can then copy, and scrollable meaning that any message shown since the start of the program can be seen until termination.

3.4.2 Wrapper Protocol

A very simple protocol was designed to communicate between the microcontroller and the application. The main motivation for simplicity was efficiency, any overhead must be avoided specially since the communication medium, RS-232, is not particularly fast. The protocol is based on two types of commands, one with fixed length and another with variable length. Every command is identified by the first byte, then if it is fixed length command the sender will output the needed missing bytes, if it has variable length it will send a number in ASCII followed by a newline character, this number is the length of the rest of the command that will then be transmitted. The receiver must wait for all these bytes before executing the command. The commands sent by the application to the microcontroller are shown in the Table 3.2, while Table 3.3 has the commands sent by microcontroller to the application. The differences between the SpiNNaker packets commands are noticeable, the reason for this approach is the reduction of overhead on one side of the communication since the microcontroller is capable of generating a suitable control header byte while when sending packets back to the host system from the SpiNNaker chip it is interesting to show the complete packet including the control byte.

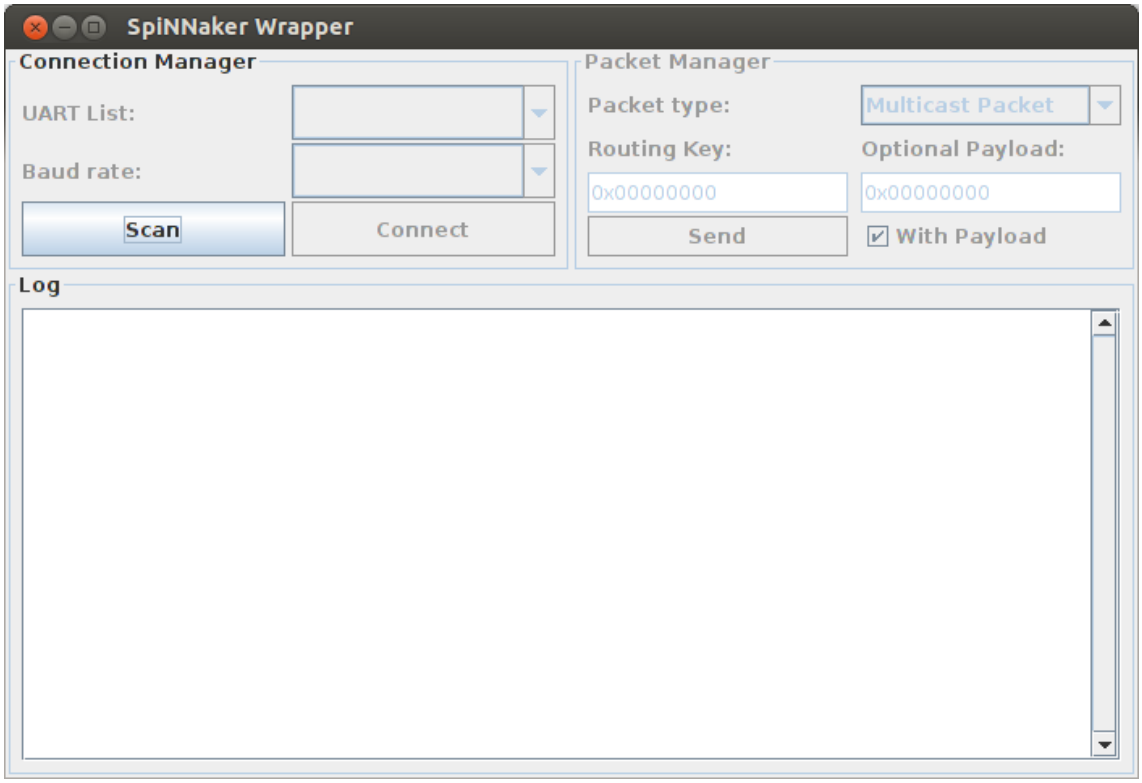


Figure 3.11: User Interface of the Workstation Application.

Command	Command Format	Length Type
Send Boot image	b<length of the boot image>'\n'<image>	Variable
Send SDP packet	s<length of the packet>'\n'<packet>	Variable
Set P2P Address	a<2 byte with address>	Fixed
Send Peek packet	e<4 bytes with address>	Fixed
Send Poke packet	E<8 bytes with address and payload>	Fixed
Send P2P packet	p<4 bytes with address>	Fixed
Send P2P packet with payload	P<8 bytes with address and payload>	Fixed
Send FR packet	f<4 bytes with payload>	Fixed
Send FR packet with payload	F<8 bytes with payload and another payload>	Fixed
Send NN packet	n<4 bytes with routing key>	Fixed
Send NN packet with payload	N<8 bytes with routing key and payload>	Fixed
Send MC packet	m<4 bytes with address>	Fixed
Send MC packet with payload	M<8 bytes with address and payload>	Fixed

Table 3.2: Commands send by the Application to the Microcontroller

Command	Command Format	Length Type
Send SDP packet	s<length of the packet><packet>	Variable
Send SpiNNaker packet	p<5 bytes with control byte and address>	Fixed
Send SpiNNaker packet with payload	P<9 bytes with control byte,address and payload>	Fixed
Show SpiNNaker packet sent from the microcontroller with payload (only used during debug)	S<9 bytes with control byte,address and payload>	Fixed

Table 3.3: Commands send by the Microcontroller to the Application

3.5 Evaluation

In order to characterize the developed computing module, some measurements were made. Both cores ran at 180 MHz during all tests. The main objective was to determine the input and output capacity in terms of SpiNNaker packets since higher levels protocols rely on these. The methodology used for these procedures was as follows:

- change the source code to set and clear a GPIO pin for the relevant section to be measured;
- use the switch present in the board to trigger the transmission of a peek or poke packet depending if the objective was to measure packets with or without payload.
- using an oscilloscope, measure the time taken between the changes in state of the previously selected GPIO pin.

The results are compiled in Table 3.4. The first two lines represent the time taken to transmit a packet, including the calculation of the parity bit and consequental addition to the header, while the following two lines represent the time taken only during the symbol transmission. There is no such difference for the input side since it is only reading symbols and placing them at the queue in the correct position. With these results, it is now possible to calculate the symbol transmission capacity. Each symbol takes around 90 ns to be transmitted which translates into a 11 MSymbol transmission rate. As for the input capacity, is around 43 % of the transmission rate at 4.78 MSymbol. These number are fairly small when comparing with capacity on the SpiNNaker side which is around 62.5 MSymbols in both direction. This large difference is easily explained by the fact the microcontroller implementation is entirely software based as opposed to the SpiNNaker which is implemented in dedicated hardware. Additionally, the difference between the transmission and the reception rate can be explained for the increased complexity when receiving and identifying the symbols as opposed to output process where there is no identification step, besides the differences of performance between the two cores. Another analysis which can be done is the fact that the transmission and reception capacity is much bigger than RS-232 at 12 Mbps, meaning that this communication channel is currently the bottleneck.

In terms of packet transmission rate, for packet with a payload, the system can achieve around 455K packets per second while for those without payload, the rate is about 770K packets per

Action	Time taken (µs)
Packet Transmission with payload	2.2
Packet Transmission without payload	1.3
Symbol transmission for packet with payload	1.8
Symbol transmission for packet without payload	1
Tasks Loop without running any action	0.55
Packet reception with payload	4.2
Packet reception without payload	2.3

Table 3.4: Performance Measurements for the transmission and reception of SpiNNaker packets.

second. As for the packet reception rate, 435K packets per second can be received if these do not include a payload, while for the packets with a payload, it is around 238K packets per second.

The final interesting number is the time taken per execution of the main loop which currently is 550ns. This number represents the time taken for every task to check if there is something to be done without actually performing some function.

3.6 Summary

The SpiNNaker Computing Module allows the use of a single SpiNNaker chip in a very small package without requiring an Ethernet connection. Through the addition of a microcontroller, which emulates an Ethernet connected SpiNNaker chip taking its own position in the 2D grid and with its own P2P address, the initial steps for a complete autonomous solution have been laid. The system is already capable of bootstrapping and communicating with the SpiNNaker chip at the various layers of its communications model which means it has comparable functionality to a standard SpiNNaker machine.

The SpiNNaker Extension board allows the system to be extended up to 256 SpiNNaker chip, although future iterations with more interchip connectors may extend this capability up to the full extent of the architecture, i.e., 65536 SpiNNaker chips.

The developed workstation application adds backwards compatibility with the currently available tools, such as *ybug* and *tubotron*, in order to allow the porting of previous development based on standard SpiNNaker machines to be ported to this new system without any effort besides some changes in the loading scripts to prevent trying to upload code and data to the *fake* SpiNNaker chip that the microcontroller represents.

Chapter 4

Case Study

This chapter presents the developed case study for the SpiNNaker Chip Computing Module which was presented in the previous sections. The selected example is based on the simulation of the aggregate motion of a flock of birds while using distributed rules for each bird. The implemented model for this simulation is traditionally named the Boids model.

This example was chosen because it requires intercommunication between the processing units in order to achieve the correct results and it is a non neural application which showcases the general purpose computing possibilities that this new solution allows. Additionally it provides an interesting visualisation that shows the increased computing power by comparing the frames per second between a pure computer implementation and one that uses a single SpiNNaker chip.

4.1 Boids Model

The Boids model is a distributed behavioural model [Rey87] for flocks of birds flying or fish schooling. It is similar to a particle system, a large set of individual particles, each with its own behaviour, but it has several crucial differences. Traditional particle system are used to model fire, smoke or water. Each particle is created, age, and die off. Unlike the objects the boids model describes, which have a geometrical shape and as a consequence an orientation, they are usually dot-like. Additionally, typical particles do not interact between themselves as opposed to bird which must do so in order to flock appropriately.

This model is often referred to as a prime example of Artificial Life [Bed03] as it illustrates a variety of its principles, such as *emergence* where complex behaviour comes from the local interaction of simple rules, and unpredictability since it is not possible to predict the direction of a bird's movement after two minutes, although they do not behave chaotically as one can easily predict their direction for the next second since it will approximately be the same.

The natural flock has certain behaviours that allows it to exist, e.g. the need for birds to stay close to the flock and the need to avoid collisions with other birds. The motivation for the latter can be easily understandable as opposed to the first behaviour where the motivations are not so clear,

though factors like protection from predators and advantages for social and matings activities may be responsible.

Therefore the simulated flocks will have some rules or behaviours that will generate the motion similar to the one produced by real ones. The basic behaviours are:

- Collision Avoidance: avoid collisions with nearby flockmates
- Velocity Matching: attempt to match velocity with nearby flockmates
- Flock Centring: attempt to stay close to nearby flockmates

Each behaviour produces an *acceleration* which is a contribution for a tunable weighted average. The relative strength of each rule will dictate the general behaviour of the flock, for example, if the flock centring behaviour has a very low impact then the flock will be very sparse while still following a common direction.

Since the model attempts to simulate the movement of birds, it must be based on a semi-realistic model of flight. It does not need to take in consideration all physical forces like aerodynamic drag or even gravity but it must limit the velocity and instantaneous accelerations to realistic values. These restrictions help modelling creatures with finite amounts of energy.

4.2 General architecture

The architecture for this implementation is presented in Figure 4.1. The SpiNNaker chip will compute the simulation and update at a rate of 30 frames per second the birds' position. The update will be sent to an application running on the Host system using SDP packets. A bird position is given by three coordinates since the simulation is on a three dimensional space. Each coordinate position takes two bytes since the simulated world space is limited to values well below the 32768 limit given by signed two byte value. This means that each bird takes up six bytes for the position and two more for an identifier which means that currently the maximum number of birds is 65536 which is a very large flock. The current maximum length for a SDP packet is 272 bytes which divided by 8 equals 34. This means that each SDP packet can carry up to 34 birds' positions.

4.2.1 SpiNNaker implementation

The SpiNNaker implementation was written in C and it uses the low level routines provided by SARK to achieve an efficient implementation. It is based on open sourced version, licensed under the GNU General Public License (GPL), by Conrad Parker named xboids [Par02] which was designed to run under X11 on Unix systems. A large numbers of modifications were required since the code used several dynamic memory allocations and floating point. The initial step was to port the code base to a fixed point math library, the one selected was the Fixed Point Math Library for C [Vor12], which is a header-only integer based fixed point library licensed under the Berkeley Software Distribution (BSD) license. After verifying that the behaviour of the flock was

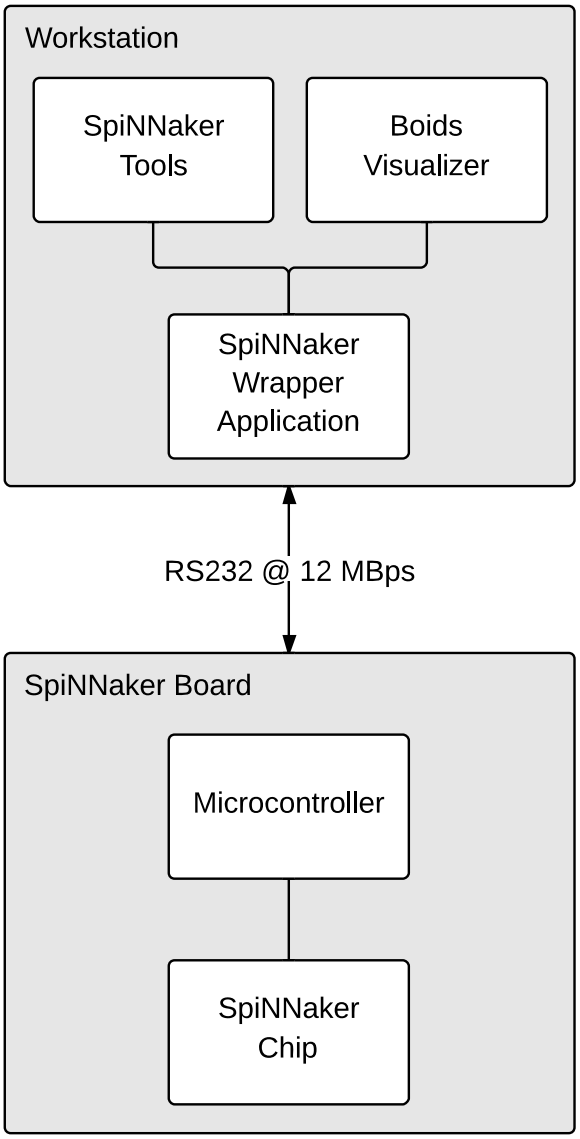


Figure 4.1: Architecture for the Boids simulation.

not different from the floating point version, the second step was to eliminate the dynamic memory allocation and replaced them with statically allocated structures and arrays. This two steps were performed on the computer version since it was easier to test and easy to compare the flock's simulation before and after the changes.

Having achieved this objective, the missing step was to create a SpiNNaker version. The first iteration focused only on having a single flock per core without any communication. This version starts by initializing a fixed number of boids and setting up a timer callback every 33 milliseconds, which is corresponds to 30 frames per second. This timer callback is responsible for updating the simulation and the visualiser by sending the SDP packets with the newly updated positions.

The timer callback starts by calculating the centre position and the average velocity of the flock. It then iterates through the boids array and updates each boid, which after being updated

the new positions are immediately copied to the outgoing SDP packet. After an packet is full, it is sent and it starts filling a new one until every boid's position has been transmitted.

The boids update process is an simple implementation of the model presented in the previous section. The boid has access to the complete array and the flock's center and average velocity. Using its current position and velocity, it computes two accelerations by comparing the flock's averages to its own status. Then it produces an aggregate acceleration from the collision avoidance algorithm, where it tries to avoid every boid that is closer than a certain constant value. With all these behaviours it produces an weight average where the collision avoidance is the most important followed by velocity matching and finally the flock centring. Additionally the velocity is limited to realistic values. The last step checks if the boid is confined to the defined world by verifying each coordinate separately, when they are not, a new fixed correcting acceleration is added on each simulation step until they are back to the allowed area. The current algorithm has $O(N^2)$ complexity since it will iterate though each boid to update it and during the collision avoidance verification it will iterate through the full set again, the original paper about this model proposes some possible improvements but since it not the main goal of this implementation, these were not pursued.

The second iteration added the propagation of the boids positions to the other cores in order to from one big flock. In practical terms, this meant adding a new callback for the reception of SDP packets which immediately calculates new collision avoidance acceleration if needed and the creation of a higher level average velocity and center for the larger flock while also propagating the boids' positions and the stats of the local flock to the other cores. There was a third iteration that tuned the weighted average so that each core remained close together while forming the bigger flock.

4.2.2 Computer Visualiser

The other component is the computer program which was named the Boids Visualiser. It was written in Java and it used a game development framework named libGDX[Zec13], which is open source, licensed under the Apache 2 License, cross platform and it has very simple APIs for drawing. The motivation to use a game development framework comes from the fact that the visualiser needs to represent a three dimensional world filled with birds. Additionally, it allowed to run the visualiser in other non-standard platform like Android and iOS.

The initial step was to port of the xboids to Java and libGDX. The limitations that required some handling while porting the software to the SpiNNaker did not apply to this version since it would be running on a workstation. The 2D API form libGDX proved to be simple enough to simulate the drawing code used on the original version for X11, the standard windowing system on Linux systems.

The next step was to remove the simulation code from the Java version and add an UDP socket listener that would take the received UDP datagrams, parse the information and update accordingly the boids present in that packet. This listener was placed on another thread running asynchronously to prevent frame drops although a mutex was placed in order to avoid race conditions.

A screen shot of the this visualiser is depicted in Figure 4.2. A runnable Java Archive (JAR) with the computer version is available for download from the project's Bitbucket page at https://bitbucket.org/rui_araujo/boidsvisualizer/downloads/boids.jar.

4.3 Evaluation

There were two versions developed for this simulation, one relying on the SpiNNaker Computing Module and another simpler only using the computer capabilities, which allows a direct comparison between the performance of both solutions with the increase of the flock's population. The methodology used for this evaluation was the following:

- select a number of birds for the simulation;
- run the simulation on the SpiNNaker Computing Module, and record the frames per second (FPS) in the visualisation:
- run the simulation on the computer and record again the frames per second.

The results of this benchmark are compiled in Table 4.1. The central processing unit (CPU) of the computer used for running this simulation was an AMD Phenom II X4 945 running at 3 GHz.

The O(N^2) complexity of the algorithm is clearly shown in the results. The increase of the number of birds leads to an ever increasing reduction of the frame rate. The version that uses the SpiNNaker Computing Module also suffers from a reduction in the frame rate from the shear number of objects it has to represent. The graphics code was not optimized to handle a large number of birds but since it is the same code for both versions, this effect can be despised.

Number of birds in the simulation	FPS with the SpiNNaker Computing Module	FPS without the SpiNNaker Computing Module
2176	60	60
4352	59	30
6528	43	15
8704	32	10

Table 4.1: Frames per second for the simulation with and without the SpiNNaker Computing Module.

4.4 Summary

The implementation of this model was a prime example of the general purposes capabilities of the SpiNNaker chip. Although this is not a typical neural application, it was a parallelizable algorithm that benefited from the fast interconnect capabilities and high speed core communication. It also allowed to showcase the true computing capabilities since the computer Java version with the included simulation could not run in real time, with noticeable frames drops, when the flock grew larger than 3000 birds.

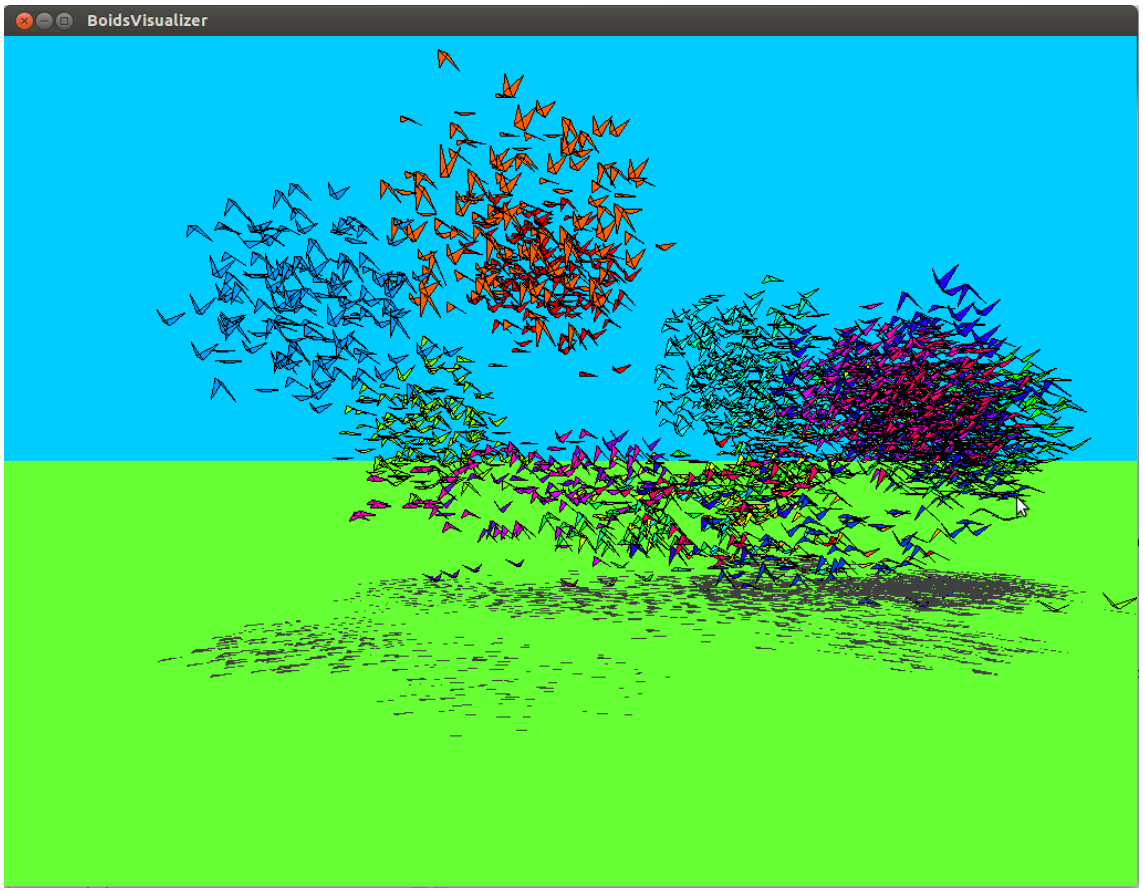


Figure 4.2: A frame of the Boids Visualiser with 2176 birds.

Chapter 5

Conclusions and Future Work

This dissertation is finalized by drawing a conclusion, summarizing the contributions and difficulties, and discussing directions for future work.

5.1 Summary

A deep analysis of the SpiNNaker system and specially of the SpiNNaker chip was given in Chapter 2. This chip is equipped with a complete tool chain and advanced parallel capabilities that can foster interesting new applications in the fields of small robotics and others, where this style of computational power is welcome.

In Chapter 3, it was present the developed system which included a custom PCB board with a microcontroller and a single SpiNNaker chip together with an inter-chip connector to allow communication with extra SpiNNaker chips without any changes, firmware for the microcontroller that emulated the networking capabilities of a SpiNNaker chip with an Ethernet connection and a computer application that was necessary to trap the UDP datagrams that are normally used with standard SpiNNaker machines. Additionally, an extension board with a single SpiNNaker chip was proposed in order to allow granular increases in computing power.

An example study case was developed which implemented the simulation of the movement of a flock of birds. The implemented model and the architecture used were presented in Chapter 4. This example allowed to demonstrate the computing capabilities that a single SpiNNaker chip may bring and its general purpose applicability.

Observing the results it may be concluded that the Spinnaker platform in the form of a module like the one that was developed during this work is an interesting candidate for the implementation of a controller brain in a wide range of applications of highly parallelizable processes with very low energy consumption.

5.2 Difficulties

The main difficulties came from the implementation of the SDP over P2P protocol where it was necessary to set up several configuration after the second stage boot. In order to understand which packets were needed, a thorough analysis of the SC&MP source code was required, specially the sections where the kernel processed the received SpiNNaker packets. While debugging, a large amount of the process was spent analysing the source code of the this kernel in order to understand what was happening inside of the SpiNNaker chip and how it was reacting to the transmitted packets.

5.3 Future Work

Although the presented solution fully complies with the proposed objectives, there a few improvements that could be added with future work.

5.3.1 PCB Layout

The developed layout was a proof of concept and a development board for the software that was required to build for a working solution. It can be improved by using the two sides of the PCB for mounting of components and selecting smaller packages for the microcontroller and other components. Additionally more connectors can be added to the SpiNNaker chip to extend the inter-chip connectivity possibilities.

As for the SpiNNaker Extension Board, an extra chip could be place in the other side of the board meaning that each extension board would add two extra SpiNNaker chips without any increases in size. Another possibility would be the move of the power circuits to the other side of the PCB and allow significant reductions in the size of the board.

5.3.2 Full Workstation Independence

The current software is capable of booting the SpiNNaker chip without using a workstation by saving the boot image to a local EEPROM memory. This is not enough to have a complete fully independent mobile system as it is missing the binaries for the application processors.

Since the SpiNNaker chip is capable of running different binaries on each of its cores and the SpiNNaker system is extremely flexible and extensible, the solution is non trivial. A possible solution is a new hardware and software design that can handle the recording of a previous communication session with the Host system and replicate it automatically on later power-ons. Basically the SDP packets used by *ybug* would be recorded in a non-volatile memory which would be retransmitted on following initialisations. In order to prevent using new components, the internal flash banks could be used to store these recording though the life time of these bank can severely reduced if there is constant deployment of new code. Otherwise, a new storage component could be selected with higher levels of durability in terms of write cycles.

Appendix A

Developed Hardware

The developed boards are available in a Git repository at https://bitbucket.org/rui_araujo/extensiblespinnakerpcb.

A.1 Test Board

Figure A.2 depicts the PCB layout for the test board while Figure A.1 depicts the schematic. Figure A.3 displays how the test board was used together with the SpiNNaker machine 102 to develop the software.

A.2 Final Board

The final board layout is displayed in Figure A.5 while Figure A.4 depicts the schematic.

A.3 SpiNNaker Extension Board

The SpiNNaker Extension board schematic is depicted in Figure A.6 and the layout in Figure A.7.

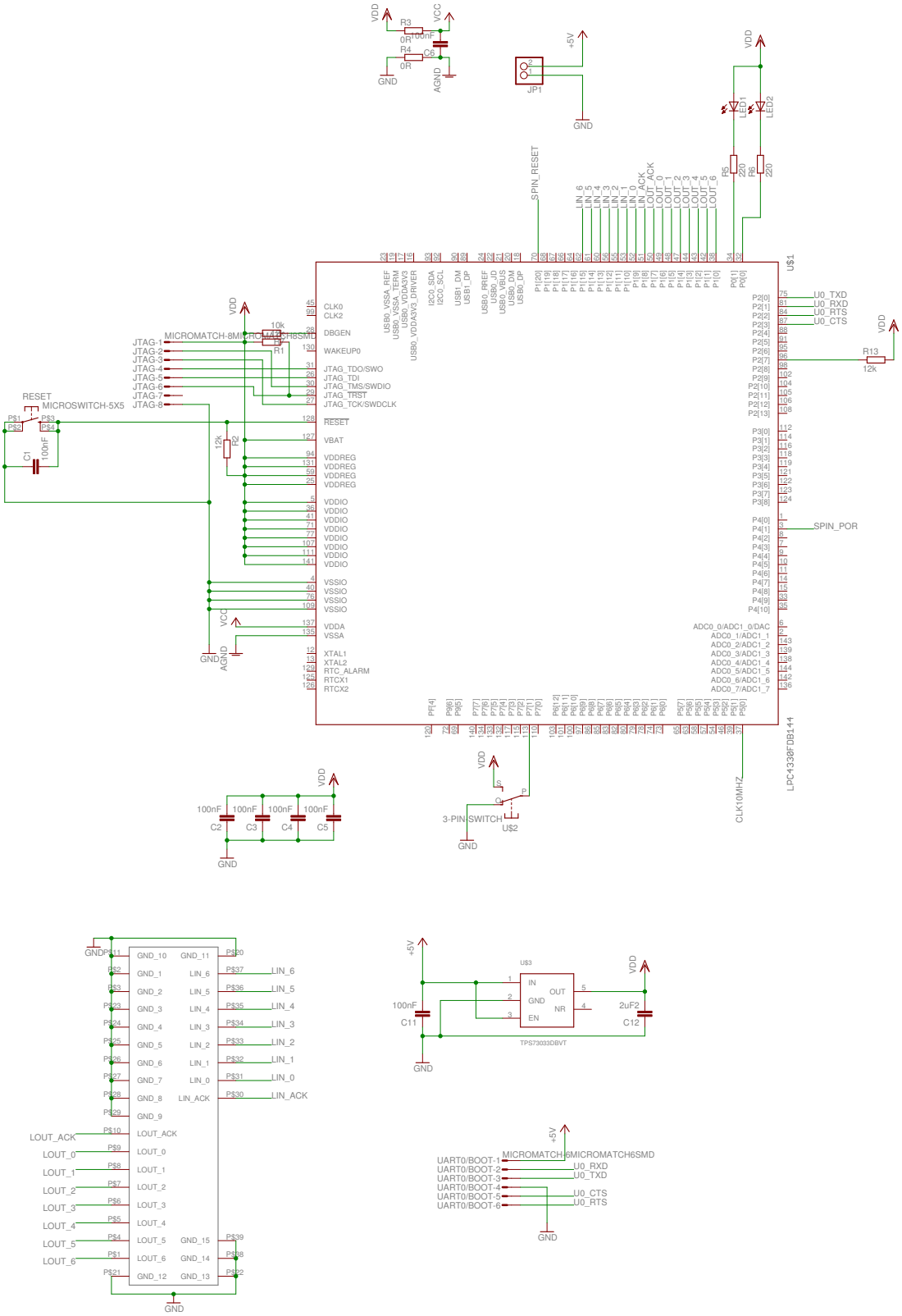


Figure A.1: Schematic of the initial testing board.

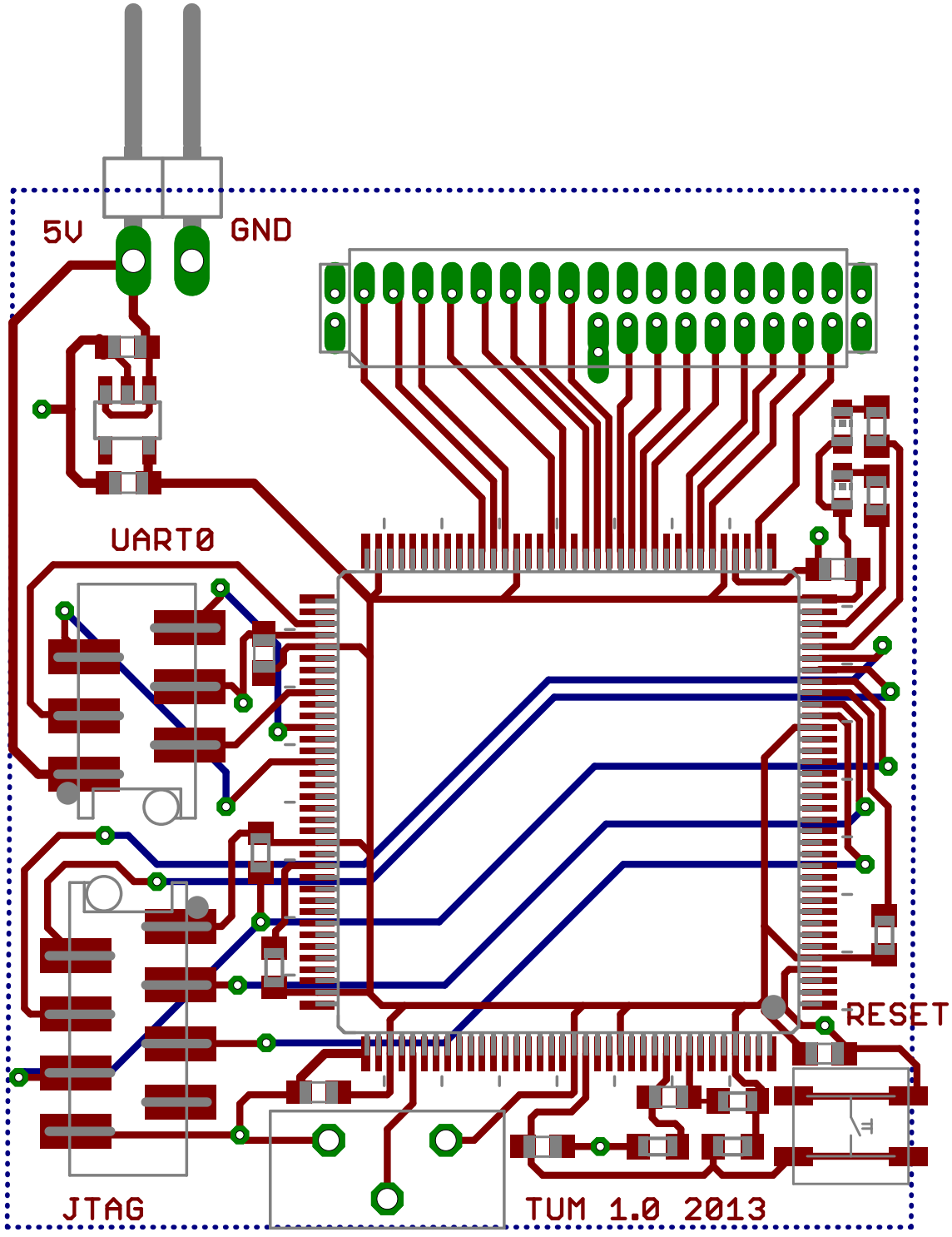


Figure A.2: Developed PCB layout for the initial testing board.

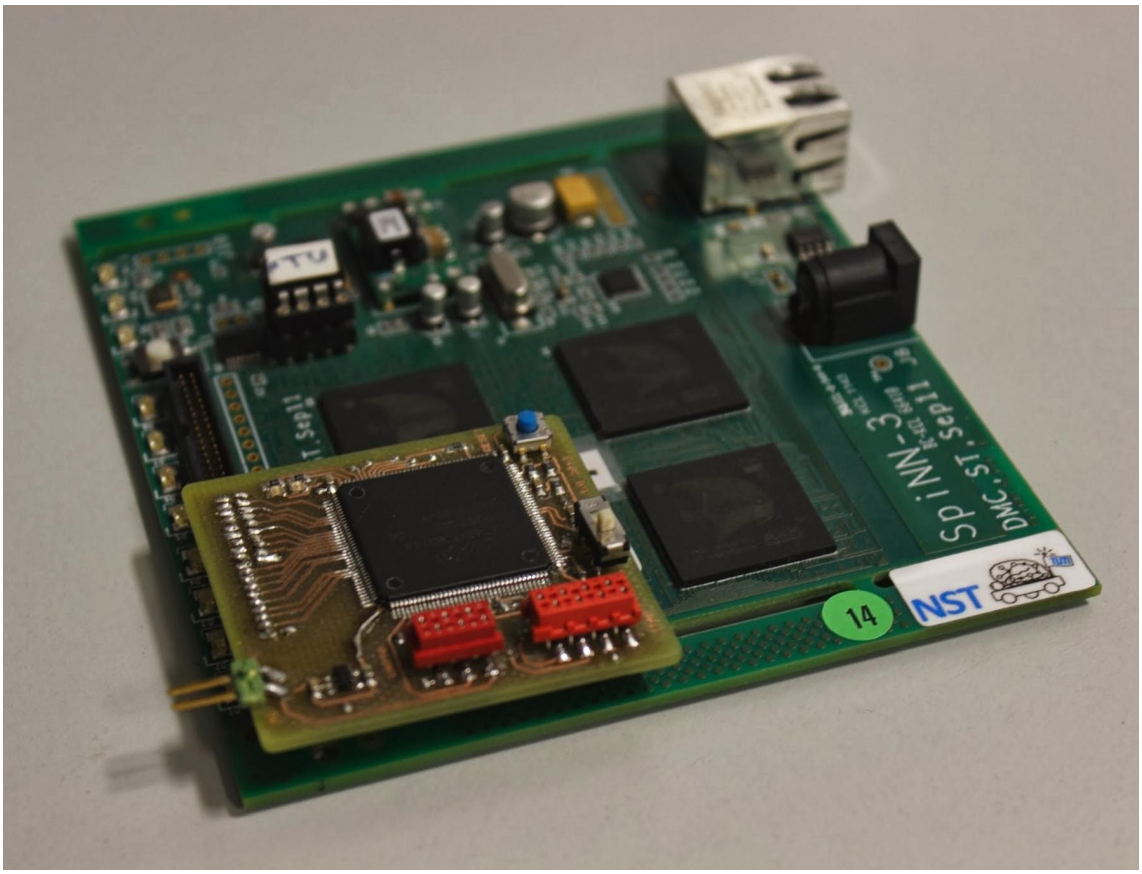


Figure A.3: SpiNNaker 102 machine with the test board.

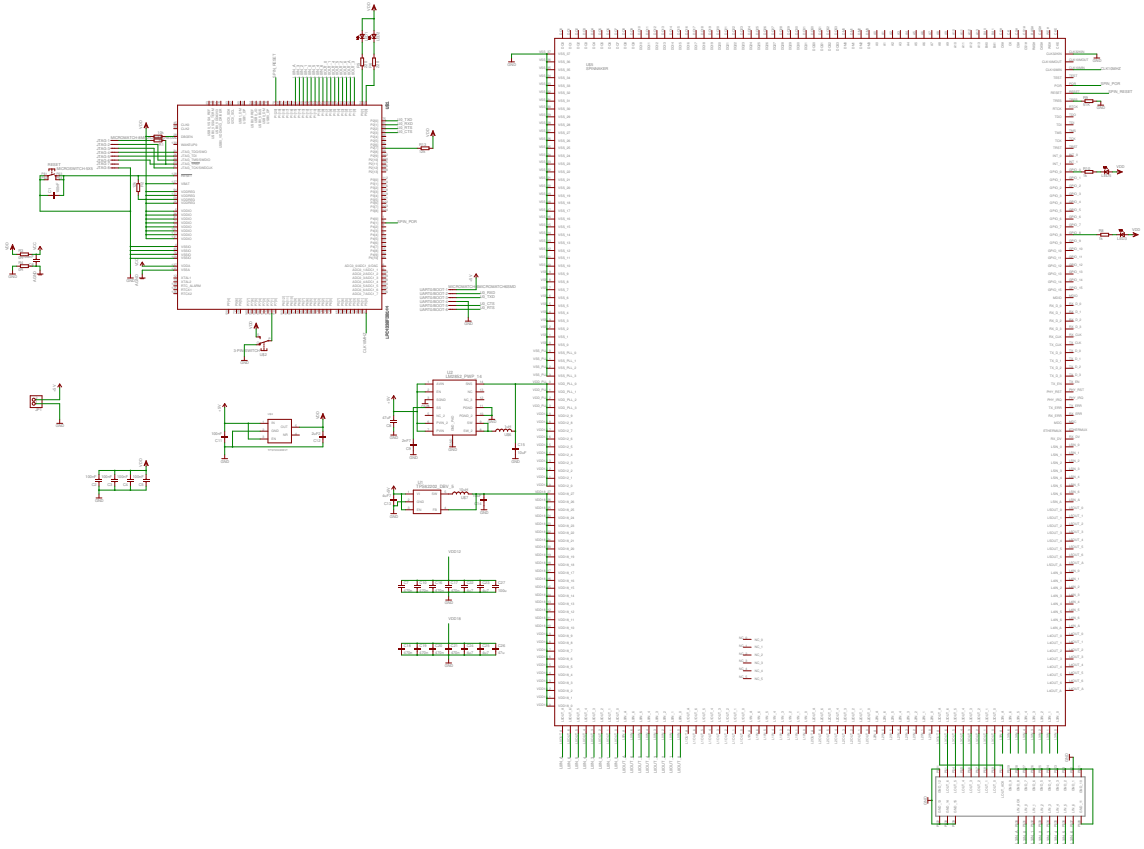


Figure A.4: Schematic of the final design.

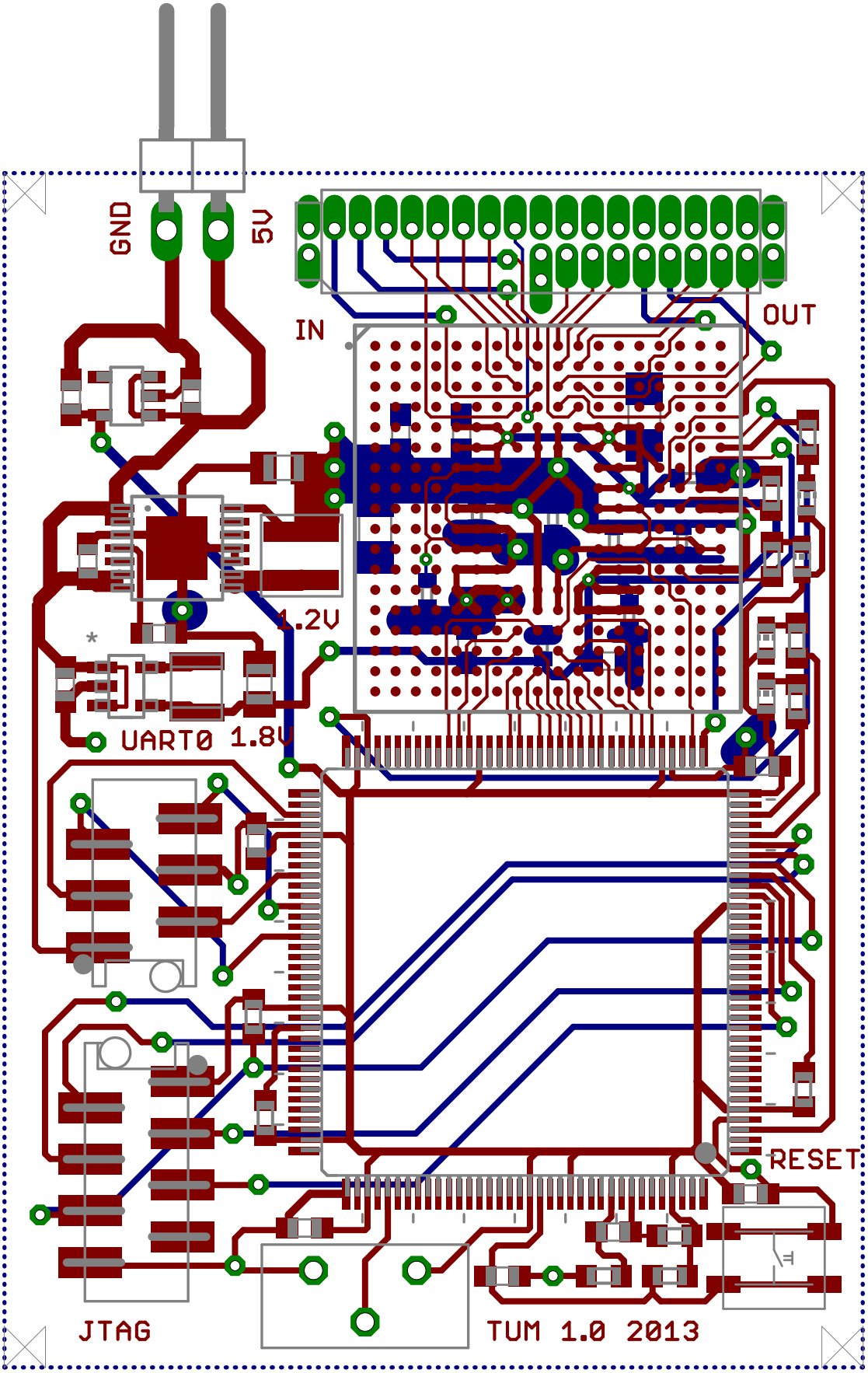


Figure A.5: Final PCB layout with the SpiNNaker chip and power supplies.

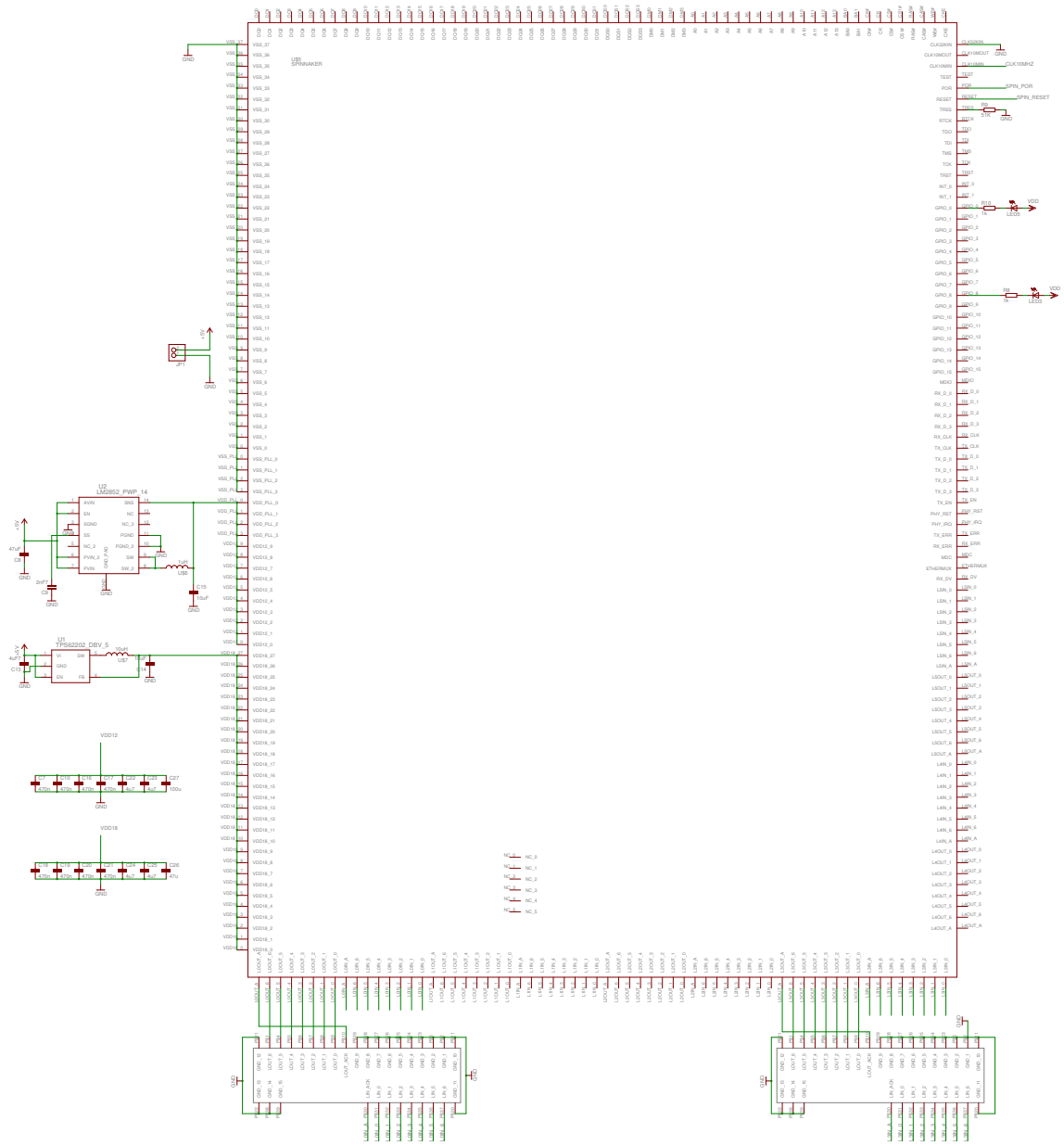


Figure A.6: Schematic of the SpiNNaker extension board.

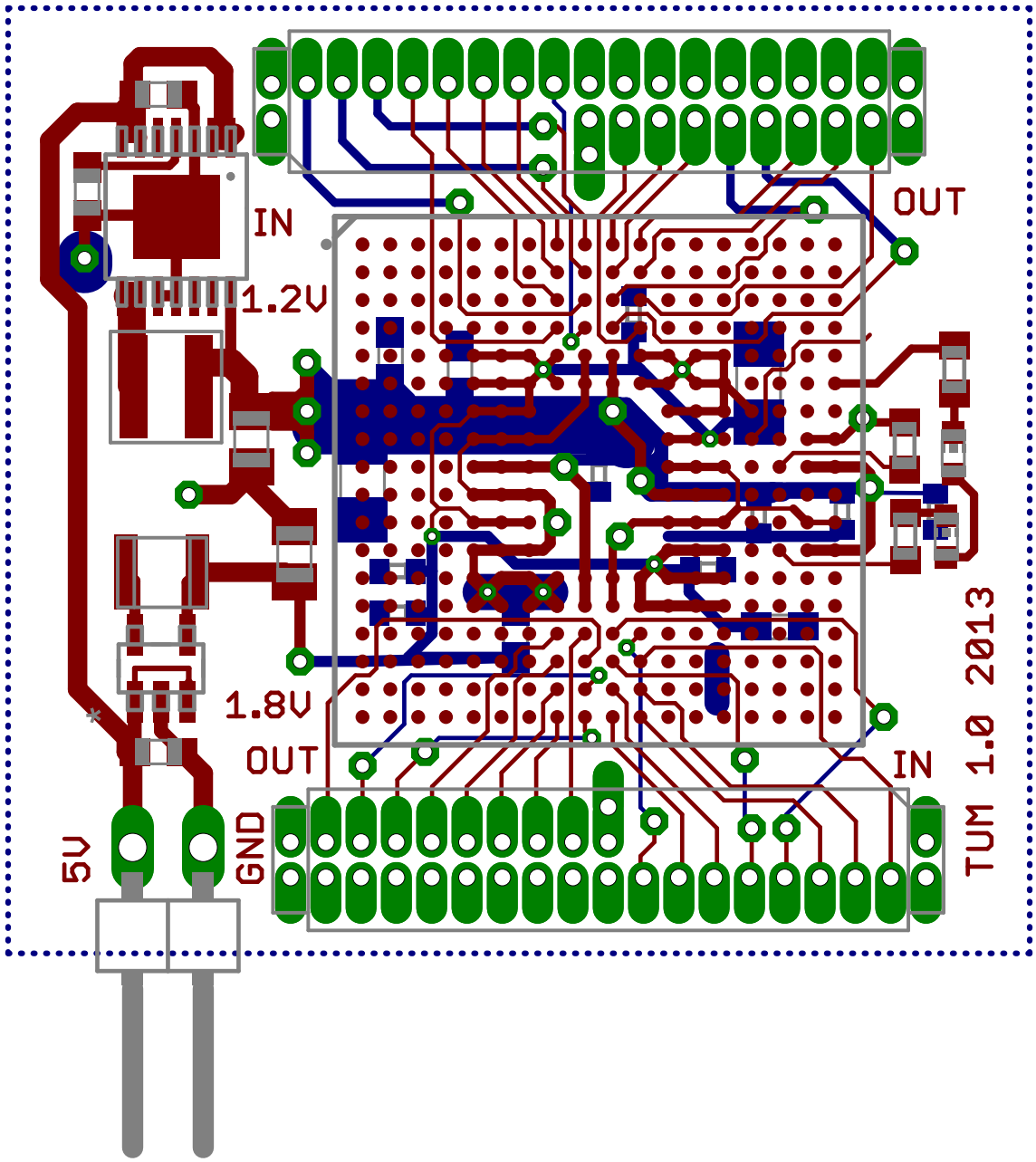


Figure A.7: SpiNNaker chip extension board.

Appendix B

SDP over P2P Packets

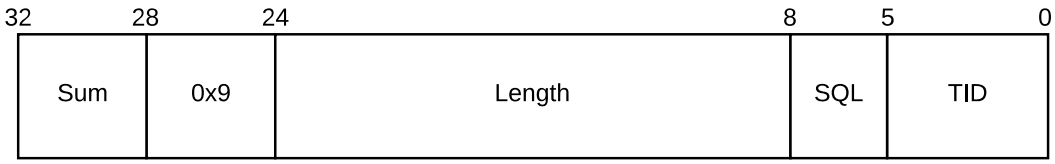
B.1 Payload of the different packets used

The Figure B.1 depicts the payloads of the P2P packets used in the several steps of the protocol. The meaning of the abbreviations used in the figure is explained here.

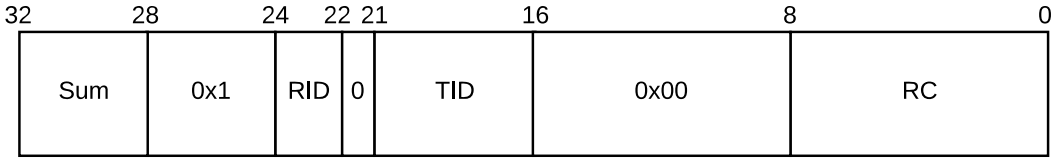
- Sum – a 4 bit checksum used in every non-data packet.
- Length – the length of the SDP packet.
- SQL – the number of P2P data packets sent in each burst, currently fixed at 16.
- TID – an identifier on the sender side of the transmission.
- RID – an identifier on the receptor side of the transmission.
- RC – a response code used by the receptor to report state and possibly errors.
- Seq Num – a number that identifies the data packet inside of the sixteen packet sequence.
- Data – 24 bits of actual SDP packet data.
- ACK Mask – a 16 bit number that the receptor send to the sender to inform it of the missing data packets with each bit set to 1 being a missing packet of the sequence.

Every non-data packet has a constant number in the bits 24:28 which is an identifier of the packet type. If it does not have this identifier then it is considered a data packet, the test is done through the bit 25 which is always zero in the data packets while in the others is always one.

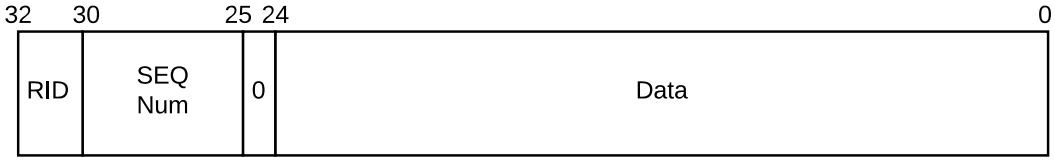
Open Request



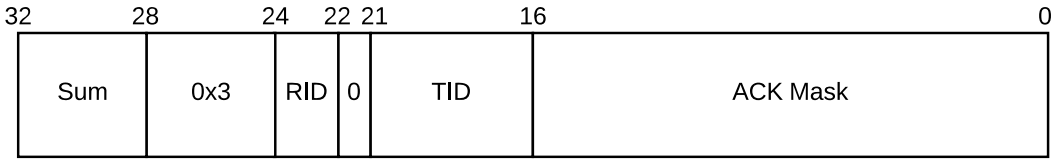
Open ACK



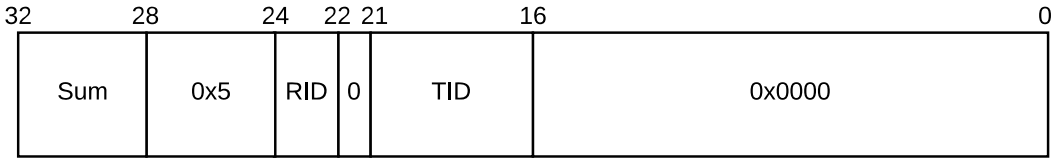
Data Packet



Data ACK



Close Request



Close ACK

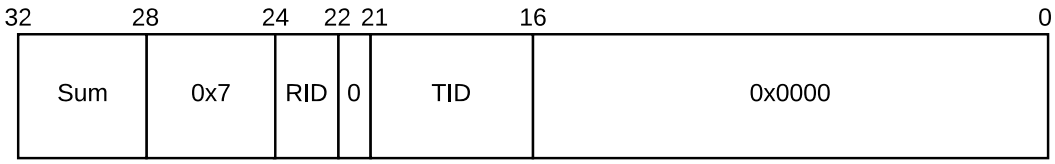


Figure B.1: The different payload packets.

Appendix C

Source code

The developed source code is available in multiple Git repositories [Tor05]. The final total amount of code in terms of lines count is around 7000 lines of code.

C.1 Microcontroller Firmware

The microcontroller firmware is located at https://bitbucket.org/rui_araujo/extensiblespinnakerfirmware. It is divided in four LPCXpresso projects [NXP13b], two for each core where one is a peripheral driver project supplied by NXP which was optimized in certain hot spots, like the GPIO routines, to avoid unnecessary overhead when running in non-debug mode and the other is the main firmware project. The line count for the two firmware projects is 4200 lines.

C.2 Workstation application

The workstation application is simpler when compared with the microcontroller's firmware. It is available as a single Eclipse Java project at https://bitbucket.org/rui_araujo/extensiblespinnakerwrapperapp. The line count for this project is 1377 lines.

C.3 Boids Simulation

The case study was divided in two repositories, one with the SpiNNaker implementation available at https://bitbucket.org/rui_araujo/spinnboids and another with the computer components and initial proof-of-concept, accessible at https://bitbucket.org/rui_araujo/boidsvisualizer. The computer components include the basic C version that used as a base for the SpiNNaker implementation. The line count for these projects is 1476 lines.

Appendix D

SARK Source code and API

The SARK source code is available at https://solem.cs.man.ac.uk/trac/browser/SpiNNaker_svn/spin1_api/tags/spin1_api-20121110. The following section displays the header file with all the relevant API functions whose behaviour was explained in section 2.2.4.

D.1 SARK API

```
/****a* spinn_api.h/spinn_api_header
*
* SUMMARY
*  SpiNNaker API main header file
*
* AUTHOR
*  Luis Plana - lap@cs.man.ac.uk
*
* DETAILS
*  Created on   : 03 May 2011
*  Version      : $Revision$
*  Last modified on : $Date$
*  Last modified by : $Author$
*  $Id$
*  $HeadURL$
*
* COPYRIGHT
*  Copyright (c) The University of Manchester, 2011. All rights reserved.
*  SpiNNaker Project
*  Advanced Processor Technologies Group
*  School of Computer Science
*
*******/
#ifndef __SPINN_API_H__
```

```
#define __SPINN_API_H__

#include "spinnaker.h"
#include "spinn_sdp.h"

// -------------------------------------------------------------
// Useful SpiNNaker parameters
// -------------------------------------------------------------
/* shared memory */
/* system RAM address and size */
#define SPINN_SYSRAM_BASE SYSRAM_BASE
#define SPINN_SYSRAM_SIZE SYSRAM_SIZE
/* SDRAM address and size */
#define SPINN_SDRAM_BASE SDRAM_BASE
#define SPINN_SDRAM_SIZE SDRAM_SIZE

// -------------------------------------------------------------
// general parameters and definitions
// -------------------------------------------------------------
/* boolean constants */
#define TRUE         (0 == 0)
#define FALSE        (0 != 0)
/* function results */
#define SUCCESS      (uint) 1
#define FAILURE      (uint) 0
/* Null pointer value */
#define NULL         0

// -------------------------------------------------------------
// event definitions
// -------------------------------------------------------------
// event-related parameters
#define NUM_EVENTS     5
#define MC_PACKET_RECEIVED 0
#define DMA_TRANSFER_DONE 1
#define TIMER_TICK     2
#define SDP_PACKET_RX  3 // !! ST
#define USER_EVENT     4
// -------------------------------------------------------------

// -------------------------------------------------------------
// DMA transfer parameters
// -------------------------------------------------------------
// DMA transfer direction (from core point of view)
#define DMA_READ     0
```

```
#define DMA_WRITE    1

// -------------------------------------------------------------
// packet parameters
// -------------------------------------------------------------
// payload presence
#define NO_PAYLOAD   0
#define WITH_PAYLOAD 1

// -------------------------------------------------------------
// type definitions
// -------------------------------------------------------------
// !! ST typedef unsigned char uchar;
// !! ST typedef unsigned int uint;
// !! ST typedef unsigned short ushort;

typedef void (*callback_t) (uint, uint); // callbacks
// -------------------------------------------------------------

// -------------------------------------------------------------
// simulation control functions
// -------------------------------------------------------------
uint spin1_start(void);
void spin1_stop(void);
void spin1_kill(uint error);
void spin1_set_timer_tick(uint time);
void spin1_set_core_map(uint chips, uint * core_map);
uint spin1_get_simulation_time(void);
void spin1_delay_us (uint n);
// -------------------------------------------------------------

// -------------------------------------------------------------
// callback and task functions
// -------------------------------------------------------------
void spin1_callback_on(uint event_id, callback_t cback, int priority);
void spin1_callback_off(uint event_id);
uint spin1_schedule_callback(callback_t cback, uint arg0, uint arg1, uint
    priority);
uint spin1_trigger_user_event(uint arg0, uint arg1);
// -------------------------------------------------------------

// -------------------------------------------------------------
// data transfer functions
// -------------------------------------------------------------
```

```
uint spin1_dma_transfer(uint tag, void *system_address, void
    *tcm_address, uint direction, uint length);
void spin1_memcpy(void *dst, void const *src, uint len);
// -------------------------------------------------------------

// -------------------------------------------------------------
// communications functions
// -------------------------------------------------------------
uint spin1_send_mc_packet(uint key, uint data, uint load);
void spin1_flush_rx_packet_queue(void);
void spin1_flush_tx_packet_queue(void);
// -------------------------------------------------------------

// -------------------------------------------------------------
// SDP related functions
// -------------------------------------------------------------
void spin1_msg_free (sdp_msg_t *msg);
sdp_msg_t* spin1_msg_get (void);
uint spin1_send_sdp_msg (sdp_msg_t *msg, uint timeout);
// -------------------------------------------------------------

// -------------------------------------------------------------
// interrupt control functions
// -------------------------------------------------------------
uint spin1_irq_disable(void);
uint spin1_fiq_disable(void);
uint spin1_int_disable(void);
void spin1_mode_restore(uint sr);
// -------------------------------------------------------------

// -------------------------------------------------------------
// system resources access funtions
// -------------------------------------------------------------
uint spin1_get_id(void);
uint spin1_get_core_id(void);
uint spin1_get_chip_id(void);
void spin1_led_control (uint p);
uint spin1_set_mc_table_entry(uint entry, uint key, uint mask, uint
    route);
void* spin1_malloc(uint bytes);
// -------------------------------------------------------------

#endif /* __SPINN_API_H__ */
```

References

[ARM13] ARM. ARM968 Processor, 2013. URL: http://www.arm.com/products/processors/classic/arm9/arm968.php [last accessed 2013-12-18].

[Bed03] Mark A. Bedau. Artificial life: organization, adaptation and complexity from the bottom up. *Trends in Cognitive Sciences*, 7(11):505 – 512, 2003.

[BF02] J. Bainbridge and S. Furber. Chain: a delay-insensitive chip area interconnect. *Micro, IEEE*, 22(5):16–23, 2002.

[Bro09] Sean Brotherson. Understanding Brain Development in Young Children, April 2009. URL: http://www.ag.ndsu.edu/pubs/yf/famsci/fs609.pdf [last accessed 2013-08-31].

[CSBR10] Hsin Chen, S. Saïghi, L. Buhry, and S. Renaud. Real-time simulation of biologically realistic stochastic neurons in vlsi. *Neural Networks, IEEE Transactions on*, 21(9):1511–1517, Sept 2010.

[CSF+12] Swadesh Choudhary, Steven Sloan, Sam Fok, Alexander Neckar, Eric Trautmann, Peiran Gao, Terry Stewart, Chris Eliasmith, and Kwabena Boahen. Silicon neurons that compute. In AlessandroE.P. Villa, Włodzisław Duch, Péter Érdi, Francesco Masulli, and Günther Palm, editors, *Artificial Neural Networks and Machine Learning – ICANN 2012*, volume 7552 of *Lecture Notes in Computer Science*, pages 121–128. Springer Berlin Heidelberg, 2012.

[CV11] Tennessee Carmel-Veilleux. AutoBGA, February 2011. 1.2 Release. URL: https://code.google.com/p/autobga/ [last accessed 2013-09-20].

[DBE+09] Andrew P Davison, Daniel Brüderle, Jochen M Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2(11), 2009.

[DD11] P. Deitel and H. Deitel. *Java How to Program (early objects)*. Pearson Education, 2011.

[DLBG+13] Christian Denk, Francisco Llobet-Blandino, Francesco Galluppi, LuisA. Plana, Steve Furber, and Jörg Conradt. Real-time interface board for closed-loop robotic tasks on the spinnaker neural computing system. In Valeri Mladenov, Petia Koprinkova-Hristova, Günther Palm, Alessandro E.P. Villa, Bruno Appollini, and Nikola Kasabov, editors, *Artificial Neural Networks and Machine Learning – ICANN 2013*, volume 8131 of *Lecture Notes in Computer Science*, pages 467–474. Springer Berlin Heidelberg, 2013.

[DT01] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689, 2001.

[ES03] Terry Elliott and Nigel R Shadbolt. Developmental robotics: manifesto and application. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1811):2187–2206, 2003.

[FB05] S. Furber and J. Bainbridge. Future trends in soc interconnect. In *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, pages 183–186, 2005.

[FB09] S. Furber and A. Brown. Biologically-inspired massively-parallel architectures - computing beyond a million processors. In *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, pages 3–12, 2009.

[Fur13] Steve Furber. SpiNNaker - a Spiking Neural Network Architecture, April 2013. URL: https://www.informatics.manchester.ac.uk/SiteCollectionDocuments/Human%20Behaviour%20Network/StephenFurberSpiNNaker.pdf [last accessed 2013-10-10].

[GBC+05] A. Gara, M.A. Blumrich, D. Chen, G.L.-T. Chiu, P. Coteus, M.E. Giampapa, R.A. Haring, P. Heidelberger, D. Hoenicke, G.V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, March 2005.

[Gro11a] Advanced Processor Technologies Research Group. Software Specification and Design, December 2011. URL: https://solem.cs.man.ac.uk/trac/browser/SpiNNaker_svn/spinnSoft_design_doc/tags/v0.0/SpiNNsoft_designV00.pdf [last accessed 2013-12-15].

[Gro11b] Advanced Processor Technologies Research Group. SpiNNaker datasheet, January 2011. URL: https://wiki.lsr.ei.tum.de/lib/exe/fetch.php?media=nst/programming/spinn2datashtv202.pdf [last accessed 2013-12-20].

[Gro13] Advanced Processor Technologies Research Group. SpiNNaker Project - Boards and Machines, 2013. URL: http://apt.cs.man.ac.uk/projects/SpiNNaker/hardware/ [last accessed 2013-12-15].

[HOF+12] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[IBM12] IBM. *A2 Processor User's Manual*. 2012. URL: https://wiki.alcf.anl.gov/parts/images/c/cf/A2.pdf [last accessed 2013-09-30].

[ID00] Giacomo Indiveri and Rodney Douglas. Neuromorphic vision sensors. *Science*, 288(5469):1189–1190, 2000.

[Inc09] Analog Devices Inc. Decoupling Techniques, 2009. URL: http://www.analog.com/static/imported-files/tutorials/MT-101.pdf [last accessed 2013-10-03].

[Ins06] Texas Instruments. TPS62202, May 2006. URL: http://www.ti.com/product/tps62202 [last accessed 2013-11-11].

[Ins11] Texas Instruments. TPS73033, Feb 2011. URL: http://www.ti.com/product/tps73033 [last accessed 2013-11-12].

[Ins13] Texas Instruments. LM2852, Apr 2013. URL: http://www.ti.com/product/lm2852 [last accessed 2013-11-10].

[ISO99] ISO. ISO C Standard 1999. Technical report, International Organization for Standardization, 1999. ISO/IEC 9899:1999 draft. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf [last accessed 2013-11-18].

[Jar06] Trent Jarvi. RXTX, Feb 2006. Release 2.1-7. URL: http://rxtx.qbang.org [last accessed 2013-11-23].

[Ngu10] Thai Nguyen. Total number of synapses in the adult human neocortex. *Journal of Mathematical Modeling: One + Two*, 3(1), 2010.

[NLMA+09] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A. Plana, and Steve Furber. Understanding the interconnection network of spinnaker. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 286–295, New York, NY, USA, 2009. ACM.

[NXP12] NXP. LPC43XX Dual Core Examples, Sep 2012. URL: http://www.lpcware.com/content/project/lpc43xx-dual-core-examples [last accessed 2013-10-16].

[NXP13a] NXP. LPC4300, 2013. URL: http://www.nxp.com/products/microcontrollers/cortex_m4/series/LPC4300.html [last accessed 2013-10-16].

[NXP13b] NXP. LPCXpresso 6, Oct 2013. URL: http://www.lpcware.com/lpcxpresso/home [last accessed 2013-10-23].

[NXP13c] NXP. Placing data at an address, Sep 2013. URL: http://www.lpcware.com/content/faq/lpcxpresso/placing-data-address [last accessed 2013-12-07].

[Par02] Conrad Parker. XBoids, Feb 2002. GPL Version 2 Licensed. URL: http://www.vergenet.net/~conrad/boids/download [last accessed 2014-01-02].

[PBF+08] L.A. Plana, J. Bainbridge, S. Furber, S. Salisbury, Yebin Shi, and Jian Wu. An on-chip and inter-chip communications network for the spinnaker massively-parallel neural net simulator. In *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, pages 215–216, 2008.

[PGJ+12] T. Pfeil, A. Grübl, S. Jeltsch, E. Müller, P. Müller, M. A. Petrovici, M. Schmuker, D. Brüderle, J. Schemmel, and K. Meier. Six networks on a universal neuromorphic computing substrate. *ArXiv e-prints*, Oct 2012.

[PS06] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (cam) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712–727, 2006.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[Roh98] Douglas Rohde. Lens, June 1998. URL: http://tedlab.mit.edu/~dr/Lens/ [last accessed 2013-10-05].

[SMS07] SMSC. MAC-to-MAC MII Interface Connections, March 2007. URL: https://www2.smsc.com/mkt/web_lancheck.nsf/2f473f9215f487db852571b50046a891/6647cebf43d94b72852572b300443d47/$FILE/Schematic%20Design%20Guideline,%20MAC-to-MAC%20MII%20Interface.pdf [last accessed 2013-09-10].

[Tem11a] Steve Temple. AppNote 2 - Programming SpiNNaker with ARM and GNU tools, November 2011. URL: http://solem.cs.man.ac.uk/documentation/spinn-app-2.pdf [last accessed 2013-11-02].

[Tem11b] Steve Temple. AppNote 3 - The APLX File Format, November 2011. URL: http://solem.cs.man.ac.uk/documentation/spinn-app-4.pdf [last accessed 2013-11-02].

[Tem11c] Steve Temple. AppNote 4 - SpiNNaker Datagram Protocol (SDP) Specification, November 2011. URL: http://solem.cs.man.ac.uk/documentation/spinn-app-4.pdf [last accessed 2013-11-05].

[Tem11d] Steve Temple. AppNote 5 - Spinnaker Command Protocol (SCP) Specification, November 2011. URL: http://solem.cs.man.ac.uk/documentation/spinn-app-5.pdf [last accessed 2013-11-12].

[Tem12] Steve Temple. AppNote 7 - SpiNNaker Links, April 2012. URL: http://solem.cs.man.ac.uk/documentation/spinn-app-7.pdf [last accessed 2013-11-13].

[Tor05] Linus Torvalds. git Source Code Management, April 2005. URL: http://git-scm.com/ [last accessed 2013-11-20].

[Ver88] Tom Verhoeff. Delay-insensitive codes — an overview. *Distributed Computing*, 3(1):1–8, 1988.

[Vor12] Ivan Voras. Fixed Point Math Library for C, Oct 2012. URL: http://sourceforge.net/projects/fixedptc/ [last accessed 2013-12-20].

[YIK87] Seiji Yamaguchi, Eisuke Ichinohe, and Johji Katsura. Static random access memory, December 8 1987. US Patent 4,712,194.

[Zec13] Mario Zechner. libGDX, Nov 2013. Release 0.9.9. URL: http://libgdx.badlogicgames.com/index.html [last accessed 2014-01-06].

[Zum08] Hank Zumbahlen, editor. *Linear Circuit Design Handbook*. Newnes, 2008. URL: http://www.analog.com/library/analogdialogue/archives/43-09/linear_circuit_design_handbook.html.