Department for Physics and Astronomy

University of Heidelberg, Germany

Diploma Thesis
in Physics

submitted by

Florian Painke
born in Kaiserslautern, Germany

February 2007

Hardware-Software Interface and Data Flow of the ALICE HLT

Development of critical components for the data flow through the preprocessing hard- and software used in the High-Level Trigger of A Large Ion Collision Experiment at CERN

by

Florian Painke

This diploma thesis has been carried out at

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Volker Lindenstruth

Hardware-Software Schnittstelle und Datenfluß des ALICE HLT

Der Datenfluß des *High-Level Trigger* (HLT) ist ein kritischer Faktor für den Erfolg des *A Large Ion Collision Experiment* (ALICE) am CERN. Für diese Diplomarbeit wurden ein Linux Kernel Treiber sowie ein Memory Controller für die DDR SDRAM Chips auf der *Read-Out Receiver Card*, der Hardware-Beschleuniger Karte für den HLT, entwickelt. Die Diplomarbeit gibt einen Überblick über den Datenfluß und tieferen Einblick in die Arbeit an den beiden Komponenten. Ein weiteres Design für eine dritte Komponente, den Datenreformatierer, wird vorgestellt.

Die Komponenten wurden zwar für den HLT entwickelt, sind jedoch generisch und in weiten Teilen konfigurierbar. Insofern sind sie auch für andere Projekte verwendbar und werden zum Teil auch anderweitig bereits eingesetzt. Sie werden hier in der logischen Reihenfolge, die durch den Datenfluß vorgegeben ist, vorgestellt und nicht in der chronologischen Reihenfolge, in der sie entwickelt wurden.

Hardware-Software Interface and Data Flow of the ALICE HLT

The data flow of the *High-Level Trigger* (HLT) is critical to the success of *A Large Ion Collision Experiment* (ALICE) at the CERN. During the work for this thesis, a Linux kernel driver, and a memory controller for the DDR SDRAM chips on the *Read-Out Receiver Card*, the preprocessing hardware for the HLT, have been developed. The thesis gives an overview of the data flow, and an insight to the work on those two components in detail. The design for a third component, the data reformatter, is presented.

While the components were developed with the HLT in mind, they are generic and highly configurable, and can be used, or are already in use, by other projects. They will be presented in this thesis in logical order given by the data flow, rather than in the chronological order in which they have been developed or designed.

Contents

List of Figures

List of Tables

I'm astounded by people who want to 'know' the universe
when it's hard enough to find your way around Chinatown.

Woody Allen

1 Introduction

Though not easily comprehensible to many, the innermost workings of the world, and this means matter itself and the interactions it is subject to, are a constant source of fascination and ever deeper questions for the physicist. Today, we have developed, confirmed and refined, mostly through experiments in the constantly evolving field of high energy physics, a high level of understanding of matter, formulated in the standard model of particle physics.

While it is often difficult to reconcile the vast amount of resources spent on fundamental research, and the lack of immediate, conceivable use, the long term effects on applied science and industry, but also on philosophy and with it society as a whole, cannot be put aside. That said, fundamental research has spawned many by-products, most notably in the field of modern medical diagnostics and treatment, but also in far less serious fields, like the entertainment industry.

1.1 Thesis Overview

The focus of this thesis is on the data flow of the *High-Level Trigger* (HLT) of *A Large Ion Collision Experiment* (ALICE). ALICE is one of five experiments of the *Large Hadron Collider* (LHC) built at the research facility of the *European Organisation for Nuclear Research* (CERN) near Geneva, Switzerland.

The thesis starts with a brief introduction to the basics of high energy physics. In chapter 2, an overview of the LHC and ALICE is given, along with more detailed information about the detectors and the HLT in particular. The rest of the thesis deals with the HLT, only. Chapter 3 gives an insight to the *HLT Read-Out Receiver Card* (H-RORC) and specifically to the **DDR SDRAM controller** developed for this thesis. The memory controller is a core component, vital to the basic functionality of the preprocessing hardware. In chapter 4, a detailed description of the interface between the hardware and software of the HLT is given. The **PCI and Shared-memory Interface** (PSI), also developed for this thesis, allows the software to communicate with the hardware and control it, and thus is also one of the essential components. The last chapter sums up the results and gives some prospects for further development and improvement. A component not yet developed, the **data reformatter**, is presented in principle, and critical aspects are discussed.

The appendix contains documentation for the DDR SDRAM Controller and the PCI and Shared-memory Interface Library.

Generation	Leptons	q/e	m	Quarks	q/e	m
first	ν_e (e Neutrino)	0	≈ 0	d (Down)	$-1/3$	$\approx 5\,\mathrm{MeV}$
	e (Electron)	-1	511 keV	u (Up)	$2/3$	$\approx 2\,\mathrm{MeV}$
second	ν_μ (μ Neutrino)	0	≈ 0	s (Strange)	$-1/3$	$\approx 95\,\mathrm{MeV}$
	μ (Muon)	-1	106 MeV	c (Charm)	$2/3$	$\approx 1.2\,\mathrm{GeV}$
third	ν_τ (τ Neutrino)	0	≈ 0	b (Bottom)	$-1/3$	$\approx 4.2\,\mathrm{GeV}$
	τ (Tau)	-1	1.78 GeV	t (Top)	$2/3$	$\approx 174\,\mathrm{GeV}$

Table 1.1: The twelve fundamental particles [23, p. 33]. So far, physicists believe that these are the basic constituents of all matter known. Every particle has an anti-particle with corresponding physical properties which have not been included in the table. The electrical charge is given in quantities of the charge of a single electron, and the mass of the particle is given in its energy equivalent.

1.2 Physics Overview

The standard model states that matter is constituted from two types of fundamental fermionic particles, *leptons* and *quarks*. Interaction between those particles involves three[1] fundamental forces, *electromagnetic*, *strong* and *weak interaction*, and can be described through corresponding *gauge bosons*. The standard model incorporates both the *theory of electro-weak interaction* and the *theory of quantum chromodynamics*. While the leptons are subject only to electro-weak interaction, quarks also interact through the strong force.

All of the twelve fundamental fermions and their masses are listed in table 1.1. For each of these particles, there exists an anti-particle. These have not been included in the table as they have corresponding properties.

Table 1.2 shows the fundamental forces together with their typical range, relative strength and the corresponding gauge bosons. While the electromagnetic and weak forces and gravity decrease with growing distance, the strong force actually increases with growing distance. This phenomenon is known as *confinement* and the result is that quarks cannot be observed as solitary particles, like leptons.

If we leave out gravity, all forces couple to a charge. The weak force couples to the unipolar *weak charge*, while the *electric charge* associated with the electromagnetic force is bipolar. Coupling of particles subject to strong interactions can be explained by introducing a *colour charge*, hence the name chromodynamics. The colour charge can be one of red, blue, green or the respective anti-colours. Confinement can now be formulated as free particles having to be colour-neutral. This results in two types of composite particles, called *hadrons*. While *mesons* consist of quark-antiquark pairs with

[1]The fourth fundamental force, *gravitation*, is not part of the standard model and has yet to be formulated into a quantised theory. Thus, the standard model is not a complete theory of fundamental interactions.

Force	Range [m]	Strength	Gauge Boson	m
Strong	10^{-15}	> 1	g (Gluon)	0
Electromagnetic	∞	10^{-2}	γ (Photon)	$< 6 \cdot 10^{-17}$ eV
Weak	10^{-17}	10^{-14}	$W^{\pm}$	80 GeV
			Z^0	91 GeV
Gravitation	∞	10^{-41}	unknown	–

Table 1.2: The four fundamental forces [23, p. 31] and [15, ch. 16]. All interactions between particles can be described with the forces above. These forces give form and stability to matter at very different scales. While the strong and weak interaction can be observed only on a very small scale within the nucleus, the electromagnetic force and gravitation have only minor effects on this level. Electromagnetism, however, holds together atoms in molecules and molecules in solid states. Gravitation works on an even larger scale and keeps together the solar system, for example.

opposite colour charge, *baryons* are quark-triplets with each quark having a different colour. In both cases the composite colour is regarded as white, or neutral.

While the standard model seems to be sufficient to explain the phenomena seen in particle physics so far[2], it leaves a lot of questions open. Like, why there are three generations of fundamental particles, and where their properties actually come from. One of the most interesting questions is where the seemingly arbitrary mass of fundamental particles originates. The *Higgs mechanism* tries to answer this question by introducing the *Higgs field*. Interaction with this field results in particles acquiring mass. To confirm the Higgs mechanism, a *Higgs boson* has to be observed in an experiment. While physicists believe to know approximately the energy that is necessary to observe the Higgs boson, no accelerator could reach this energy, as of today. The search for this particle is one of the reasons for building the LHC at the CERN.

There are, however, other interesting things to be observed at the high energy density LHC will reach, especially when it is operating in heavy ion collision mode. With ALICE, for example, physicists hope to gain insight on the physics of the universe at a very early stage, and thus learn more about how the universe was born, actually. Quantum chromodynamics predicts that confinement will be cancelled out at either high density of hadron matter or high temperatures, leaving matter in a phase called *quark-gluon plasma*. While densities high enough might still be reached in neutron stars our days, the quark-gluon plasma at high temperatures is important mostly in the *Big Bang scenario*.

Quark-gluon plasma has been studied in other experiments, e.g. at the *Super Proton Synchrotron* (SPS) at CERN or the *Relativistic Heavy Ion Collider* (RHIC) at Brookhaven National Laboratory. However, since ultra-relativistic heavy ion collisions

[2]Apart from the non-zero mass of neutrinos, which is a bit problematic.

produce far more particles than other experiments in high energy physics, the demands at detector resolution are very high and far more data has to be processed for a single event. Even today, the raw data can neither be processed in real-time, nor can it all be saved for off-line processing at reasonable cost. A dedicated preprocessing system, the ALICE HLT, has to decide in real-time which pieces of data should be relayed for either further processing or storage.

Since the trigger has quite an impact on the quality of results, it is subject to intense studies. The ALICE HLT scans and filters the data coming from the detectors for relevant information. This is done both in hardware and software, specifically designed for this task.

I love deadlines.
I like the whooshing sound
they make as they fly by.

Douglas Adams

2 The Experiment

The history of accelerators, and thus the history of high energy physics, begins somewhere in the twenties of the last century [6]. Particle accelerators, as the name implies, accelerate particles to nearly the speed of light. In high energy physics experiments, either a particle beam is aimed at a target, or two particle beams from opposite directions are focused in one point. Either way, particles collide with very high energy. They are scattered and can form new particles. Some of the newly created particles decay almost instantaneous, and the decay products, new particles again, can be observed with detectors[1]. There are detectors which measure the momentum, the energy or the track of a particle. For different types of particles, different types of detectors have been developed.

The need for ever higher energies has pushed the cost of accelerators and experiments to the point, where only international collaborations can afford to build them. In 1952, a then still provisional council, the *Conseil Européen pour la Recherche Nucléaire*[2] (CERN), was established by eleven European governments [7]. Today, the CERN has 20 member states, and operates six accelerators and one decelerator. About 3000 employees work at the CERN, and 6500 guest scientists from over 80 countries in the world visit the CERN for their research. The University of Heidelberg, Germany, is one of about 500 universities in the world, involved in the research and building of new experiments at the CERN [8].

2.1 The Large Hadron Collider

At the time of writing of this thesis, a new accelerator is built at the CERN site near Geneva, Switzerland. The *Large Hadron Collider* (LHC) will go on-line in 2007 and will be able to accelerate protons up to 7 TeV, and heavy ions up to 575 TeV. It is built in the tunnel which already hosted its predecessor, the *Large Electron Positron collider* (LEP), and will deliver the ultra-relativistic particles to five experiments. Figure 2.1 shows an overview of the LHC and its experiments.

The tunnel is about 4.3 km in diameter and 100 m below the ground level. A cascade of smaller accelerators will deliver pre-accelerated protons or heavy ions to the main storage ring of LHC, which will then accelerate these to the final collision energy.

[1] In elastic scattering, no new particles are formed and only the scattered particles are observed with detectors. Only inelastic scattering may result in new particles being created, if the centre of mass energy is high enough.

[2] In 1954, when the CERN was officially established, the name changed to *Organisation Européenne pour la Recherche Nucléaire*. The acronym was left untouched despite the change, however.

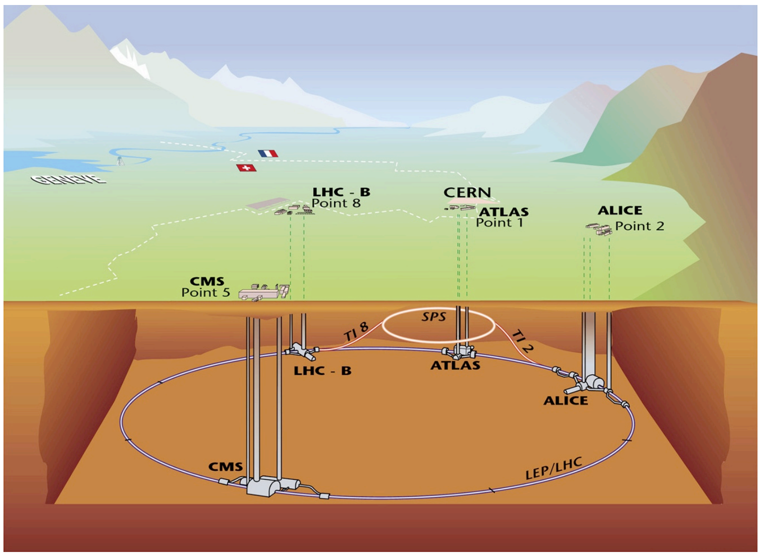
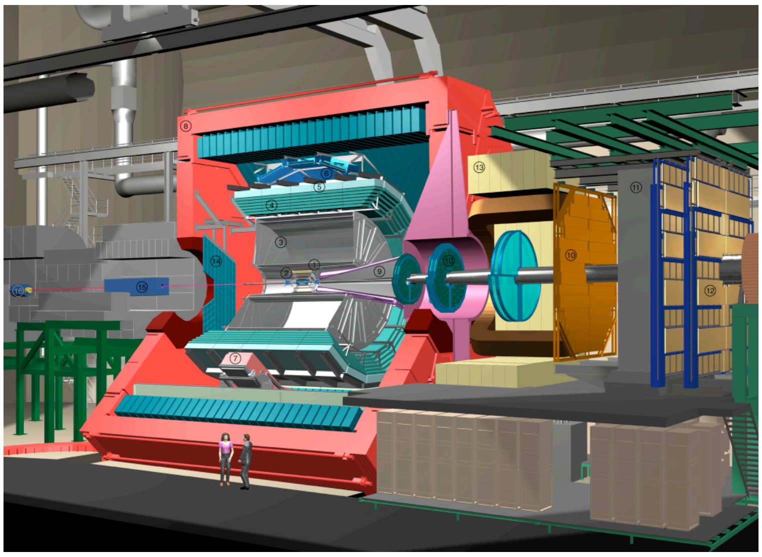


Figure 2.1: Overview of the LHC and its experiments [8]. The picture shows the main accelerator ring, the smaller pre-accelerator SPS, and the location of the four major experiments and their control centres above ground.

Two particle beams, each travelling in the opposite direction, will be kept on their circular track[3] by over 1000 super-conducting magnets, each 13 m long, generating a magnetic field of up to 8 T. The magnets will be cooled down to a temperature of about 1.8 K, below the temperature of outer space. The LHC will be by far the largest super-conducting installation in the world for the time being.

Four of the experiments will analyse proton-proton collisions. With **ATLAS** and **CMS**, physicists hope, among other things, to answer the question whether the *Higgs boson* exists, or not. This particle is critical to the explanation of why particles in general have a mass. But ATLAS will also explore physics beyond the standard model. It is yet unclear, whether quarks are built of even smaller particles, for example. An extension to the standard model model states that every particle has a *supersymmetric* counterpart. If this is true, ATLAS might be able to observe such supersymmetric particles. CMS will mainly study *muons*, but is also designed as a multi-purpose experiment. Attached to CMS is **TOTEM**, where physicists will try to measure the total cross section of proton-proton collisions. Finally, **LHCb** is built to examine *mesons* which contain the second most heavy quark, the *bottom quark*, and violation of *CP-symmetry* when these mesons decay.

[3]To be exact, the track is not circular, but rather consists of short linear sections intermitted by the magnets, which deviate the particle beam by a certain angle, each.



1	Inner Tracking System	9	Absorber
2	Forward Multiplicity Det.	10	Tracking Chambers
3	Time Projection Chamber	11	Muon Filter
4	Transition Radiation Det.	12	Trigger Chambers
5	Time Of Flight det.	13	Dipole Magnet
6	High-Mom. Part. Ident.	14	Photo Multiplicity Det.
7	Photon Spectrometer	15	Compensator Magnet
8	L3 Magnet	16	Centauro & Strange Obj. Res.

Figure 2.2: Overview of ALICE with its sub-detectors [8].

2.2 A Large Ion Collision Experiment

The fifth experiment, **ALICE**, is the only experiment which will analyse heavy ion collisions. It will study a phase of matter, called *quark-gluon plasma*, where *confinement* is cancelled out and quarks can be viewed as quasi-free particles. While other experiments claim to have observed QGP already, none of them had the energy density nor luminosity[4] at hand that LHC will provide. That said, ALICE of course will not be idle when the LHC is running in proton-proton mode, but rather this will be used to calibrate the subsystems and gather reference data. To study the dependency of the observations on energy density, LHC will vary beam particles from proton-ion collisions, lighter ion collisions to heavy ion collisions. The main mode for ALICE will be Pb-nucleus collisions.

2.2.1 The Detectors

In figure 2.2 the layout of ALICE with its detectors is shown. The assembly of ALICE is about 25 m long, 15 m high, and weighs something around 10 000 t.

The point of collision is surrounded by the **Inner Tracking System** (ITS, 1), which allows tracking of particles with a very high resolution of up to 12 µm. It consists of six

[4]Particle density in the accelerator beam. The length of the volume unit is expressed in time units, rather than length units.

layers of silicon detectors, cylindrically arranged around the beam at around 3–50 cm distance. When a particle leaves the ITS, it enters the **Time Projection Chamber** (TPC, 3), a chamber filled with Ne and CO_2 in an electric field. When particles collide with gas atoms, the latter are ionised and drift in the electric field. Free electrons produced in the ionisation process drift much faster, however. They can be observed through electrodes in the chamber and allow further reconstruction of the particle track. At a distance of around 2.5 m from the beam, the TPC is followed by the **Transition Radiation Detector** (TRD, 4), which serves mainly as trigger detector for the TPC. Six layers of drift chambers filled with Xe and CO_2, interleaved with a layer of radiator material each, detect transition radiation, which is emitted when ultra-relativistic electrons pass through the barrier layer. When the particle has travelled about 3.7 m away from the beam, the **Time Of Flight detector** (TOF, 5) measures the travelling time with a resolution of up to 150 ps. This allows calculation of the mass of the particle. For particles with very high energy, the **High-Momentum Particle Identification** (HMPID, 6) determines the mass through detection of Cerenkov radiation, emitted when the particle passes through a dielectric medium. The temperature of the collision is measured in the **Photon Spectrometer** (PHOS, 7), which is built of lead-tungsten crystals, and observes photons emitted in the collision.

Charged particles are deviated in the homogeneous field of the **L3 Magnet** (8). Through measurement of the deviation of the particle track, its momentum can be calculated.

In addition to the detectors cylindrically arranged around the centre of collision, there are a few smaller detectors for triggering purposes and event characterisation located at forward angles, as well as an array of scintillators atop the L3 Magnet which serves as a trigger for cosmic rays [1, p. ix–xiii].

2.2.2 The Trigger System

One of the main challenges when analysing heavy ion collisions is that, while the detectors work on the same principles, and the collision rate is rather moderate compared to other high energy physics experiments, due to the multiplicity of events from a single collision, the demand for detector resolution is much higher. This in turn leads to much higher data volume to be processed. In fact, the raw data volume will by far exceed the limit of what can be processed in real-time or be stored for later processing at reasonable cost, today.

Three to four[5] stages of low level triggers control the data acquisition of events. The trigger system has been carefully designed to both handle the normal mode of operation of ALICE, heavy ion collision, as well as the normal mode of operation of the rest of the LHC experiments, which is proton-proton collisions. The data is fed into the *High-Level Trigger* (HLT) system, which takes a threefold approach to reduce the data volume by more than one order of magnitude. Events which are of no interest

[5] If we take into account the pre-trigger.

are filtered out to reduce the total number of events. To reduce the size of an event, sub-events are selected within every event, and advanced loss-less data compression is applied. This of course means that trigger decisions require both local and global pattern recognition and event reconstruction.

The focus of this thesis will be the HLT, which consists of a cluster of off-the-shelf personal computers, and custom hard- and software developed at the Kirchhoff-Institute for Physics of the University of Heidelberg and other research groups around the world. While using low-cost hardware seems not to go well at first with the demand for high availability and robustness on detector hardware, a lot of work has been put to provide those demands while keeping the main advantages of this approach. For example, in case of defect, hardware can be easily replaced without the need for special distributors. The Data Transport Framework [18] developed by Timm Steinbeck and On-line Monitoring [14] in development by Camilo Lara provide the means to assure high availability without loss of data even in case of failure of single cluster nodes, and allow for fast intervening.

2.2.3 Data Flow

Readout of detector data is triggered by the *Level 2 Trigger*, and the data is sent to the *Data Acquisition* (DAQ). *Event Fragments* are assembled to sub-events in the *Local Data Concentrators* (LDC) and sent over the *Event Building Network* for further processing by the *Global Data Collectors* (GDC). Finally, event data is written to long-term storage. The LDCs will also send the raw data to the HLT. The HLT, in turn, processes and sends data back to the LDCs. Figure 2.3 shows the HLT embedded in the ALICE data flow.

To transfer data, *Detector Data Link* (DDL) modules are used. A *Source Interface Unit* (SIU) sends data over an optical fibre to a *Destination Interface Unit* (DIU). A single *HLT Read-Out Receiver Card* (H-RORC) can host both SIUs and DIUs, and provides an interface to computers via the PCI local bus. Chapters 3 and 4 deal with the H-RORC in detail, which also serves as a co-processor for the HLT tasks, and the interface for software to access the H-RORC.

The logical functions performed by the HLT can be seen in figure 2.4. On-line event reconstruction is required for the HLT trigger decision and advanced data compression. This is done in several stages, making use of locality and parallelism in the data. The *Front End Processors* (FEP) receive event data from up to four DDLs and run the *Cluster Finder*. *On-line Tracking* is done by the *Sector Processors* and, finally the *Event Processors* will do *Global Track Merging and Fitting*.

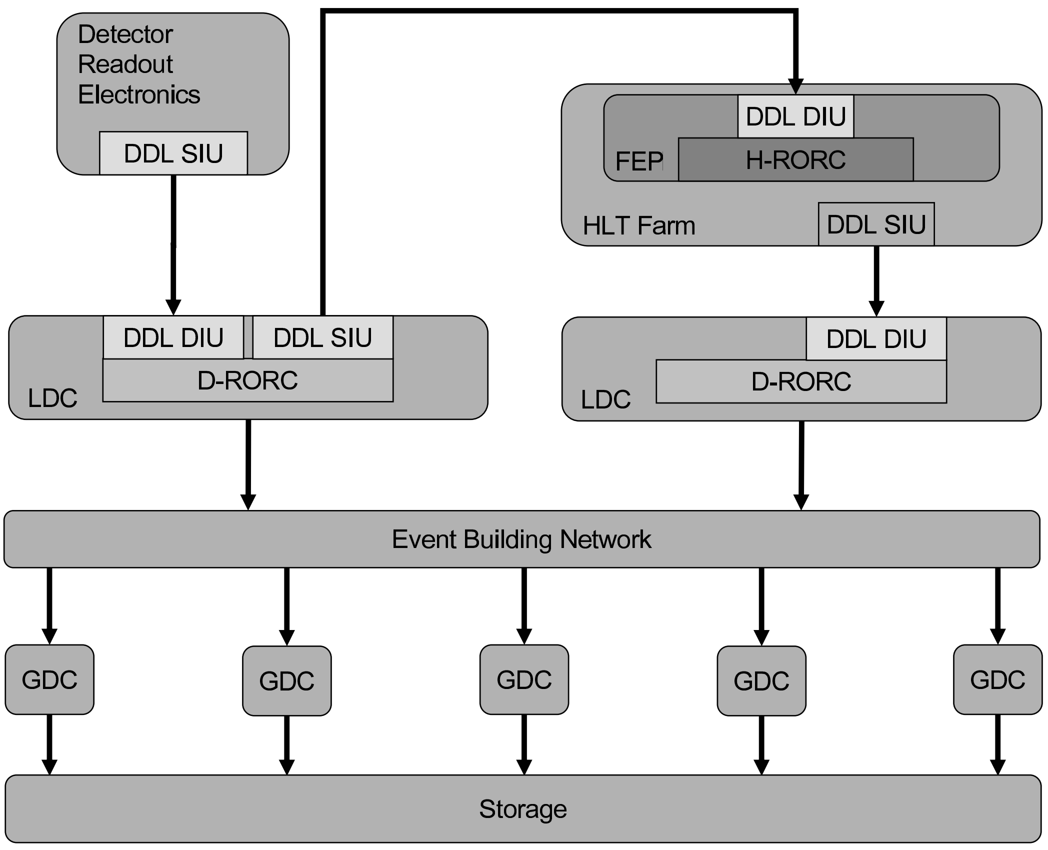


Figure 2.3: The HLT embedded in the ALICE data flow [1, p. 176]. The *Data Acquisition* send data from the detector read-out electronics to the *High Level Trigger* (HLT), which processes and sends data back to the DAQ. Both receiving and sending data is handled by *HLT Read-Out Receiver Cards* (H-RORC), equipped with *Detector Data Link* (DDL) modules. The *Destination Interface Unit* serves as data sink, and the *Source Interface Unit* as data source.

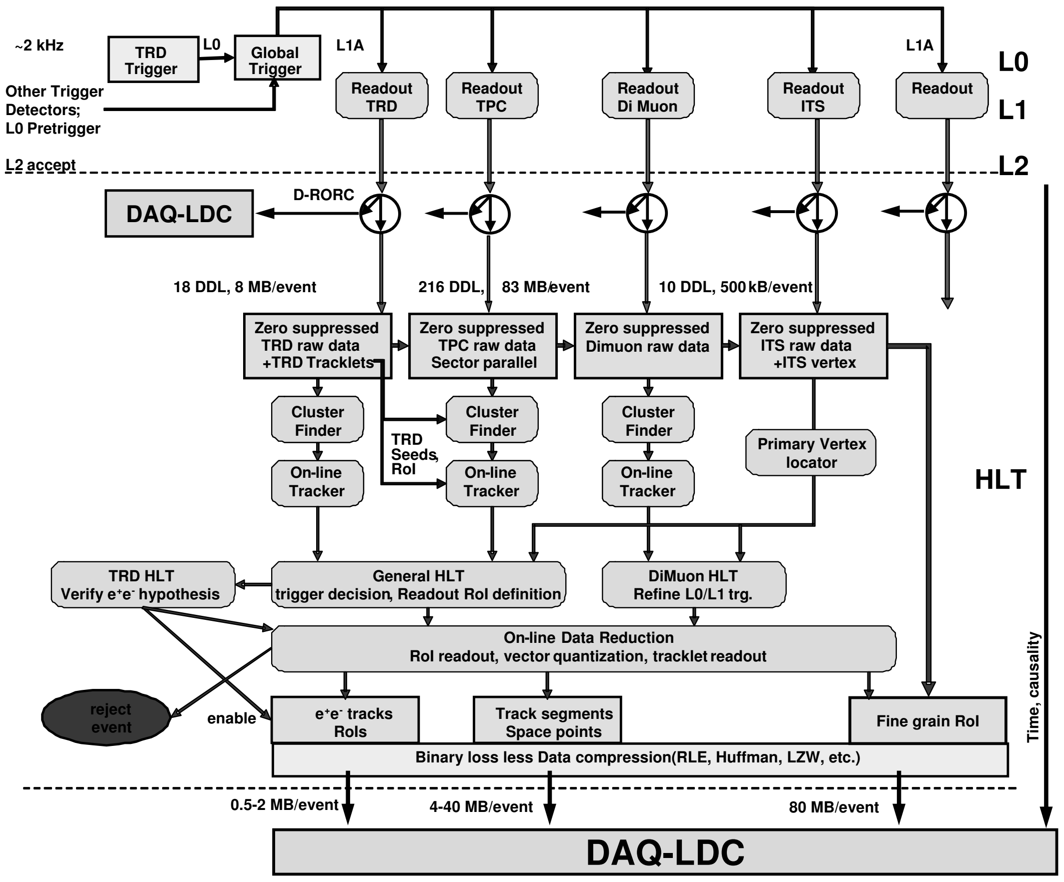

~2 kHz
TRD Trigger
L0
Global Trigger
L1A
Other Trigger Detectors; L0 Pretrigger
Readout TRD
Readout TPC
Readout Di Muon
Readout ITS
Readout
L0
L1
L2
L2 accept
D-RORC
DAQ-LDC
18 DDL, 8 MB/event
216 DDL, 83 MB/event
10 DDL, 500 kB/event
Zero suppressed TRD raw data +TRD Tracklets
Zero suppressed TPC raw data Sector parallel
Zero suppressed Dimuon raw data
Zero suppressed ITS raw data +ITS vertex
Cluster Finder
On-line Tracker
TRD Seeds, RoI
Primary Vertex locator
HLT
TRD HLT Verify e+e- hypothesis
General HLT trigger decision, Readout RoI definition
DiMuon HLT Refine L0/L1 trg.
On-line Data Reduction RoI readout, vector quantization, tracklet readout
reject event
enable
e+e- tracks RoIs
Track segments Space points
Fine grain RoI
Binary loss less Data compression(RLE, Huffman, LZW, etc.)
Time, causality
0.5-2 MB/event
4-40 MB/event
80 MB/event
DAQ-LDC

Figure 2.4: Overview of the logical functions of the HLT [1, p. 254]. Cluster finding and on-line tracking is done on a local scale in part using the *HLT Read-Out Receiver Card* (H-RORC) co-processor boards. Local tracking data is then successively combined to global tracking data used for the trigger decisions and data compression algorithms.

Any sufficiently advanced technology
is indistinguishable from magic.

Arthur C. Clarke

3 The Read-Out Receiver Card

In section 2.2.3, the *HLT Read-Out Receiver Cards* (H-RORC) were already introduced briefly as interfaces between the *Detector Data Link* (DDL) and the computers processing detector data. The H-RORC hosts up to two DDLs, each receiving data from a single data source inside the detector or transmitting data back to *Data Acquisition* (DAQ). For the *Time Projection Chamber* (TPC), the data source connected to a *Destination Interface Unit* (DIU) hosted on an H-RORC is one of six *patches* of one of 18 *sectors* at each side of the TPC. Each patch consists of a number of read-out chips, called *pads*, varying with the patch geometry. For each pad with hits in a single event, up to 1024 read-out values are packed together with a specific time granularity. These data packets for a single patch make up one event fragment.

While one of the duties of the H-RORC is to receive data from the detectors and provide it to the *Front End Processors* (FEP) for cluster finding and on-line tracking, this is not its key feature. Its main purpose is to serve as a co-processor to preprocess the event fragments, and do cluster finding and, possibly, partial tracking in hardware. Of course, since the H-RORC at most receives event fragments from two patches, this task has to be done on a local scale and the tracking data must be combined on a global scale by the software running on the cluster nodes. Fortunately, due to the natural locality of the data, this can easily be done.

The predecessor of the H-RORC, the CIA RORC, was initially designed by Deyan Atanasov for his PhD thesis [4], based on an Altera *Field Programmable Gate Array* (FPGA). The current incarnation is a complete redesign by Torsten Alt and Holger Höbbel [3], based on a Xilinx Virtex™-4 FPGA device. Figure 3.1 shows the H-RORC equipped with two DIUs and optical fibres plugged in.

Figure 3.2 shows a block diagram of the H-RORC. The board features two half length *Common Mezzanine Connectors* for the DDL interfaces and a 64 bit PCI connector, both attached to the user I/Os of the FPGA. Several serial links allow communication with a wide range of external devices, including two fast *Low Voltage Differential Signalling* (LVDS) ports, allowing to build a communication chain between the H-RORCs. The firmware for the FPGA can be either stored in the platform PROM or Flash based memory. The Flash can be used to store up to four different firmware versions and a *Complex Programmable Logic Device* (CPLD) handles loading them into the FPGA on demand. This process, called *On-line Configuration*, has been implemented by Jörg Peschek for his diploma thesis [17]. Four DDR SDRAMs are attached to the FPGA and can be used independently of each other. The development of the necessary controller logic was part of the work done for this thesis.

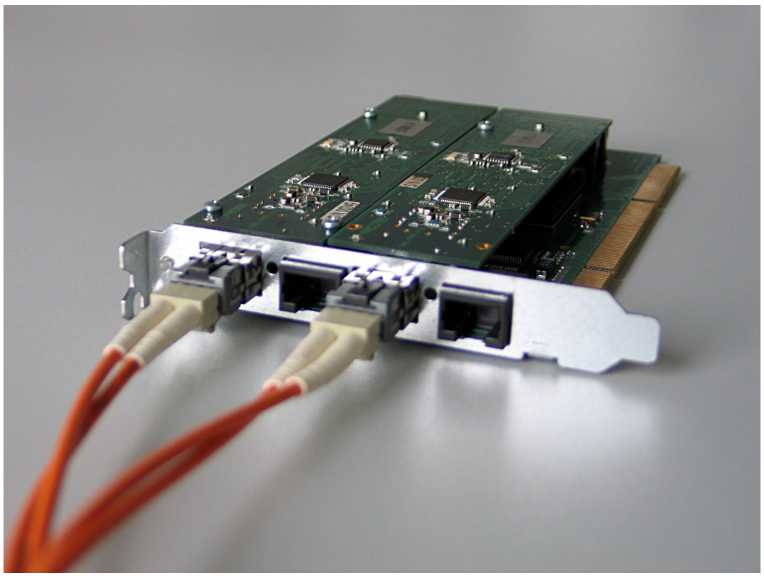


Figure 3.1: The H-RORC with DIUs and optical fibres.

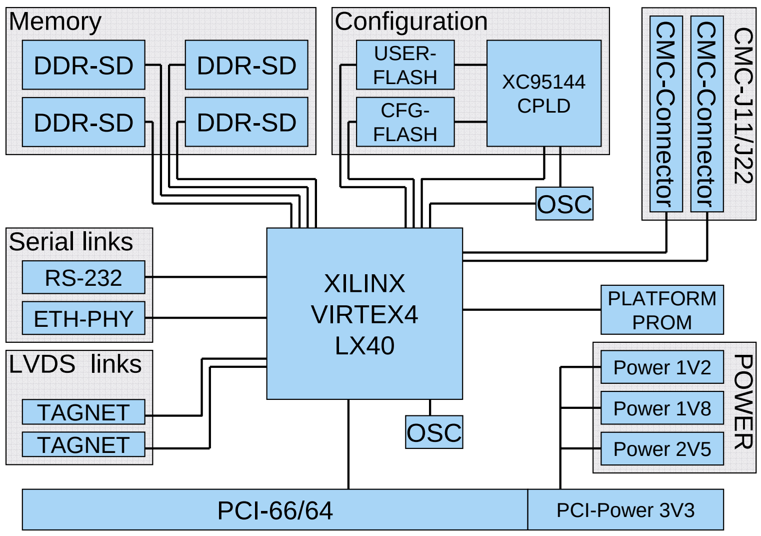


Figure 3.2: Block Diagram of the H-RORC [2]. The main component of the H-RORC is the Xilinx Virtex$^{\text{TM}}$-4 FPGA device, which communicates with the host computer through a PCI interface. Several on-board components support the FPGA and provide means for communication with other devices and daisy-chaining of H-RORCs.

3.1 Field Programmable Gate Arrays

The heart, or the brain, if you prefer, of the H-RORC is a Xilinx Virtex™-4 LX40 *Field Programmable Gate Array.*

FPGA devices are used in the industry for prototyping chip design, but gain on importance as reprogrammable co-processors, to speed up calculations that can be massively parallelised. While most FPGA devices cannot be driven in the high frequency range current processors use, they can be reprogrammed on-line, and be used for special tasks to support the main processor. This has led to several approaches to a self-reconfiguring processor that can optimise itself, and thus overcome the limitations of common processors which are tailored toward specific usage. In the H-RORC, an FPGA device is used as stand-alone processor, not for prototyping purposes, but rather because the natural parallelism in the data to be preprocessed allows to benefit from the FPGAs advantages.

The roots of FPGA devices lie within CPLD technology. Basically these are semiconductor devices with programmable logic, input and output circuitry. While programming is not possible on single gate granularity, but rather logic blocks pack together programmable look-up tables, flip-flops, multiplexers and the likes as building blocks, the details are mostly hidden from the programmer, as the synthesis software will automatically spread the design over the logic blocks. When designing at the limits of the device specification, however, the programmer has to concern himself with exactly these details and adopt a rather low-level approach to hardware programming.

The *Configurable Logic Blocks* (CLB) of a Xilinx Virtex™-4 FPGA are attached to a switch matrix and contain four *slices*, each slice providing two four-to-one *Look-Up Tables* (LUT), two storage elements which can be configured either as latches or flip-flops, two multiplexers and basic arithmetic blocks. The switch matrices interconnect the CLBs. In addition to the CLBs, there are several ready-to-use supplementary block resources incorporated to the FPGA, like *Digital Clock Managers* (DCM), *Block RAMs* (BRAM), *Double Data Rate Input/Output Registers* (IDDR, ODDR) and *Input Delays* (IDELAY), that can be instantiated in a custom design. For detailed information on the Xilinx Virtex™-4 FPGA device, see [20].

3.2 Hardware Description Language

There are several ways to *program* a hardware device like an FPGA. Most vendors provide software that features a graphical user interface for easy assembling of logic in a point-and-click metaphor, more like designing schematics. The designer can place logic gates and complex logic blocks, interconnect these, and create hierarchies of more and more complex logic functions. While this approach is convenient for smaller designs, it proves to be tedious on a larger scale. The most notable disadvantage, however, is that such a design is almost impossible to port when switching to a different FPGA device.

The solution is to use *Hardware Description Languages* (HDL). Most of these languages have their roots in simulation of digital circuitry rather than synthesis of real hardware, but as of lately have been extended in that direction. So, basically, the hardware programmer uses only a subset of such a language, which can be actually synthesised. Again, most of the higher level constructs which may be synthesiseable, should be avoided when designing near the limits of a hardware device or when trying to remain portable among different synthesis packages.

While most HDLs bear a certain similarity to computer programming languages, such similarities are quite problematic. The programming paradigms for hardware design are very different from those common in software development, and the time and effort for the development cycle is often underestimated. Let alone debugging hardware is somewhat more difficult than debugging software, comparable at most to the complexity of debugging operating system level software.

For the H-RORC firmware, VHDL is used. The V in VHDL is an abbreviation for *Very High Speed Integrated Circuit* (VHSIC). VHDL allows hierarchical design by structuring separable logic functions into *entities*. Entities and parameters can be grouped together in *packages*. The definition of an entity allows intermixed *concurrent statements* and *sequential statements*, as well as instantiation of sub-entities. Statements operate on *signals* or groups of signals. Concurrent statements are evaluated continuously and in parallel, but dependent on the levels of logic, different concurrent paths deliver their results from a change in one of the signals involved at different times. Sequential statements can be *sensitive* to the state of certain signals and define a sequence of logic evaluations with a certain result path. A block of sequential statements is evaluated every time one of the signals it is sensitive to changes its state.

Both concurrent and sequential statements can be used to describe asynchronous logic, which is subject to critical timing requirements. In most digital designs, however, asynchronous or combinatorial logic is at some point synchronised to a *clock signal*, to assure stable signals at defined, equally spaced points in time. *Flip-flops* are used to store the result of some combinatorial path at the *rising edge* of the clock signal. Synchronous logic components can only be described in sequential statements.

An entity has an *interface*, which defines the input and output signals for communication with other parts of a design, and one or more *architectural implementations*, which define its behaviour. Before using an entity somewhere in a design, the interface has to be declared in the corresponding scope. This is done in a *component declaration*.

For a deeper understanding of the hardware components developed or designed for this thesis, a level of knowledge of VHDL, and hardware design in general, is necessary which cannot be given here. There is a plethora of literature on VHDL available both in print and on-line, so matters will be left with the brief overview given. This may suffice for the reader to understand what this is all about.

3.3 The DDR SDRAM Controller

Connected to the FPGA on the H-RORC are four 32 MB DDR SDRAM chips, independent from each other. While a design using one or more of those devices could possibly drive all necessary control signals, there is quite some overhead in managing

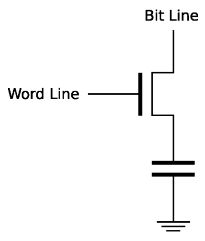


Figure 3.3: A single DRAM cell. The word line controls access to the capacitor which stores the bit value. The bit value can be written or read by means of charging or discharging the capacitor using the bit line.

SDRAM type memory, which justifies the use of a dedicated controller component.

There are several DDR SDRAM controller designs available which claim to be ready-to-use. The catch is, however, either these are tailored towards a specific FPGA family, which happens not to be the Xilinx Virtex™-4, or they are not so ready-to-use as you might wish for. Since four independent controller instances are needed, the design needs to be very lightweight, so that the overall footprint for accessing the memory chips is minimal. After evaluation of the remaining choices it became clear, that for stripping down and customising of an existing design, it must be fully understood, and the time and effort to do so is comparable to the development of such a controller from scratch.

3.3.1 Memory Technology

DDR SDRAM is short for *Double Data Rate Synchronous Dynamic Random Access Memory.* DRAM technology stores single bits in capacitors, a charged capacitor representing a logic one (see figure 3.3). The immediate consequence from using capacitors as storage elements is that reading the contents of the memory is destructive. To see whether a given bit is set or not, the capacitor has to be discharged and the current is measured. This implies a write-after-read strategy.

The memory is organised in rows and columns of bits, grouped together in *words*[1]. When reading from or writing to the DRAM chip, first the row containing the word, or words, addressed has to be *activated.* The contents of the row are read into an internal row buffer. Read and write operations are then performed on columns within the row buffer, not the memory. After operations on the row buffer are done, the row must be *precharged*, that is, the row buffer is written back to the memory. This means, while accessing row contents in the buffer, the corresponding contents do not exist in memory, only in the buffer. Rows are further grouped together in *banks.* Every memory bank can have at most one active row at a time.

[1]A word is the smallest element addressable and corresponds to the width of the data port of the chip.

The second consequence from using capacitors is that the memory suffers from self-discharge effects due to leak currents and fluctuations. So the charge in the capacitors has to be refreshed at certain intervals. This can be either done for the entire memory, or one row at a time.

Due to the analog nature of DRAM, the interface is asynchronous and subject to strict timing requirements. SDRAM adds a synchronous interface to DRAM, to simplify handling of memory access from digital designs, which are almost unanimously synchronous. Still, the DRAM timing requirements are somewhat transparent as wait-states in the synchronous interface definition.

DDR SDRAM inherits most of the properties from SDRAM. The double data rate feature only applies to transmission of data words, not control signals. While in standard single data rate designs, signals are recorded to the rising edge of the clock signal only, double data rate designs use both the rising and the falling edge of the clock signal[2]. This yields in double the rate of data transmitted, hence the name. However, it is important to note that, since control signals are single rate and the timing requirements are mostly the same, overall performance is not really doubled. In fact, when using the Xilinx Virtex™-4 DDR input/output registers, there's an overhead of at least one additional clock cycle compared to *Single Data Rate* (SDR). Since the design itself may use SDR only, a single-rate-double-width translation is done.

For a good introduction to memory technologies, see [13, ch. 7] and [5, ch. 8]. A detailed description of the DDR SDRAM chips mounted on the H-RORC board is given in [10]. There you can also find a more detailed block diagram and simplified state machine for the control sequences. Only those parts relevant to the controller design will be given later on.

3.3.2 Implementation

The demands at the memory controller for the DDR SDRAMs of the H-RORC are to provide linear addressing, opaque handling of refresh, and light-weight high-performance design. The DDR SDRAM controller developed for this thesis fulfils all three requirements and is a generic, highly configurable design, which can be used with other designs based on FPGAs of the Xilinx Virtex™-4 device family. It consists of three sub-components. The **state machine** generates control signals for the DDR SDRAM chip and the other sub-components of the controller. The **address path** provides mapping of a linear address to the address scheme of the chip, which is divided into bank, row and column address. Activation and precharging of rows is handled automatically. The third sub-component, the **data path**, does the single-rate-double-width translation and generates the write strobe signal, which is used by the DDR SDRAM to record the data signals. Figure 3.5 shows a block diagram of the controller.

The memory controller uses only a burst length value of two, thus requiring one command for each double-word read or written. While this may seem to contradict the

[2]To be exact, for DDR SDRAM, rather a dedicated strobe signal is used than the clock signal itself.

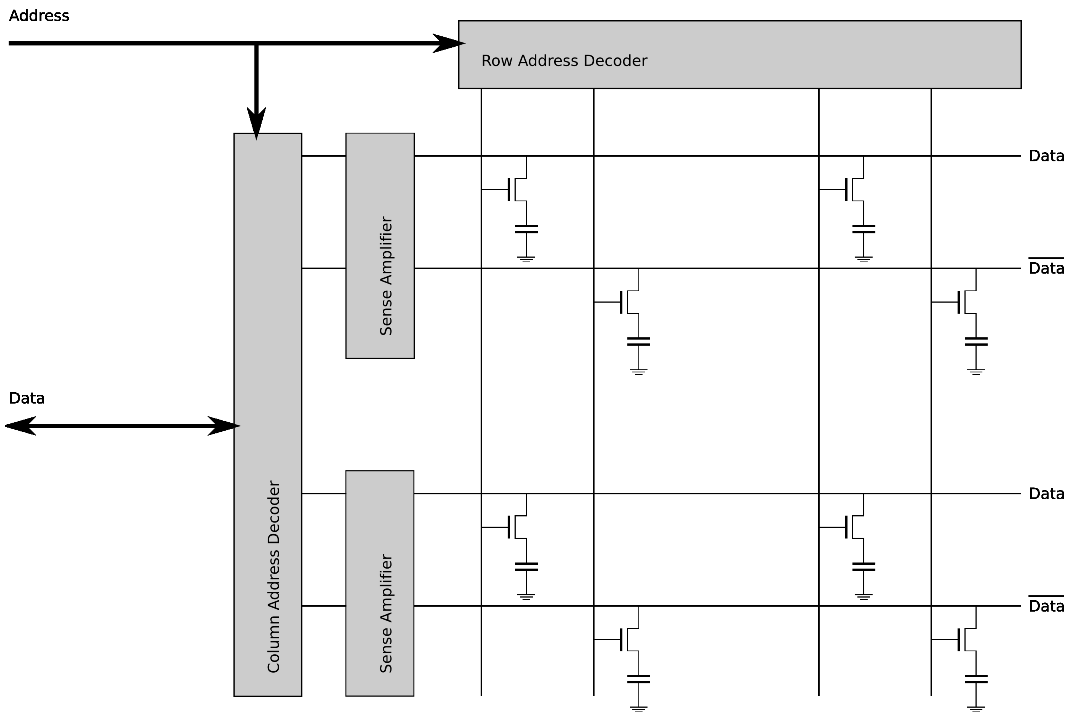


Figure 3.4: Simplified DRAM bank block diagram. The row address decoder will activate a number of DRAM cells by assigning their word lines. All cells within a row are then read by discharging the corresponding capacitors. A single bit is actually represented by *two* DRAM cells, of which only one will be charged, depending on the bit value. Since the capacitors are very small, a sense amplifier will measure the current during discharging. All bit values will be read into the row buffer and the column address decoder is used to access single words within the buffer. After the operation is done, the bit values have to be written back to the DRAM cells by charging the corresponding capacitors.

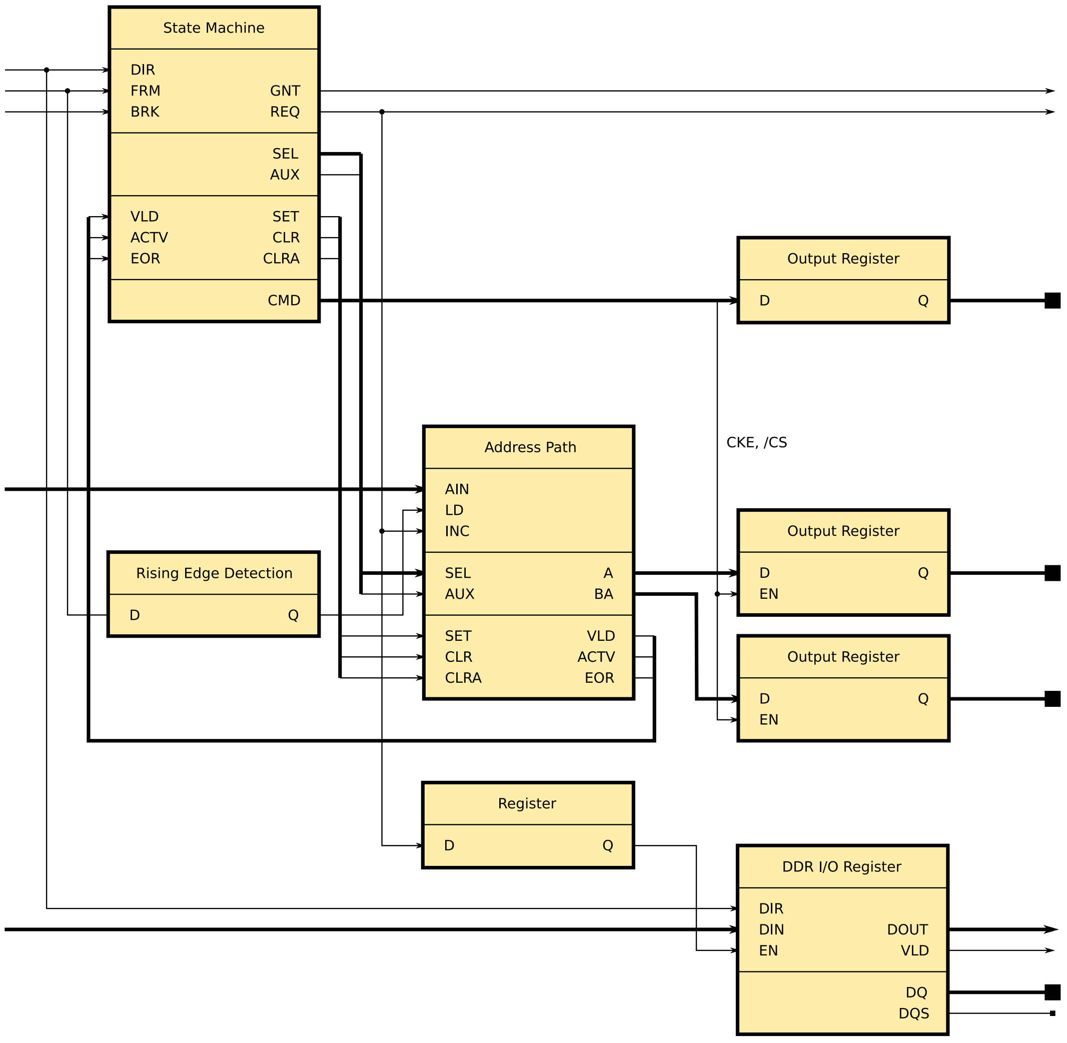


Figure 3.5: Memory controller block diagram. The state machine reacts to user control inputs and generates user handshake and DRAM control outputs, and control signals for the other sub-components. The address path handles activation and precharging of rows and generates DRAM address outputs. The data path captures both data from the user design and the DRAM through a double DDR input/output register.

need for high data throughput, it makes address translation a whole lot easier, and the controller takes advantage of the back-to-back burst feature[3] available with SDRAM. Effectively, this decision has no impact on the memory performance after all.

In appendix A, a detailed description how to use the DDR SDRAM controller is given.

The State Machine is the central and most complex sub-component of the controller. It implements frame based memory access using three user control input signals and three user handshake output signals. Opaque handling of the SDRAM management overhead results in 20 states for the main state machine and another 17 states for the initialisation sequence. Figure 3.6 shows the state machine for the controller, the initialisation sequence state machine is shown in figure 3.7.

The initialisation sequence is critical to successful operation of DDR SDRAM type memory. The sequence resets the DDR SDRAM internal state machine and *Delay Locked Loop* (DLL), and sets operational parameters like burst length and CAS latency. It is not necessarily identical for different chip families, but there is a rather slow sequence which works with almost all chips. Since initialisation is done only once at start up, there is no need in optimising for a single chip and loosing compatibility with others.

The main state machine is active after the initialisation sequence is done and enters the `IDLE` state. When the refresh timeout counter reaches zero, the refresh sequence is triggered. This happens every 7.8 µs. The counter wraps around and is not reset after the refresh is done, to warrant for the periodic refresh cycle not to exceed the maximum allowed average timing. When the user design asserts the `FRM` signal, the state machine enters a data transfer frame with the `ENTER_FRAME` state. The state machine communicates directly with the address path and handles row activation and deactivation as necessary in the `NEXT_ROW` and following states. Depending on the `DIR` signal state, either read or write transaction is done using back-to-back burst in `DO_READ` and `DO_WRITE` respectively. The controller will signal a data request to the user design through the `REQ` signal and the user design can pause requesting using the `BRK` signal. At the end of a row, the address path will assert the `EOR` signal and the next row will be activated. Handling of refresh is done if necessary, and only in the paused states `READ_WAIT` and `WRITE_WAIT`. The transfer frame can be exited only in the paused states, and the user design should never deassert the `FRM` signal without asserting `BRK` first.

The Address Path consists of a two-stage[4] counter and a register file, to store the active row for each memory bank. The stage one counter value is compared to the

[3]Using back-to-back bursts, it is possible to access the contents of a row without wait states between single transactions.

[4]This fashion of pipelining a design is often used for high-speed logic. When combinatorial logic gets more complex, many levels of logic are needed and the path delay my become too large.

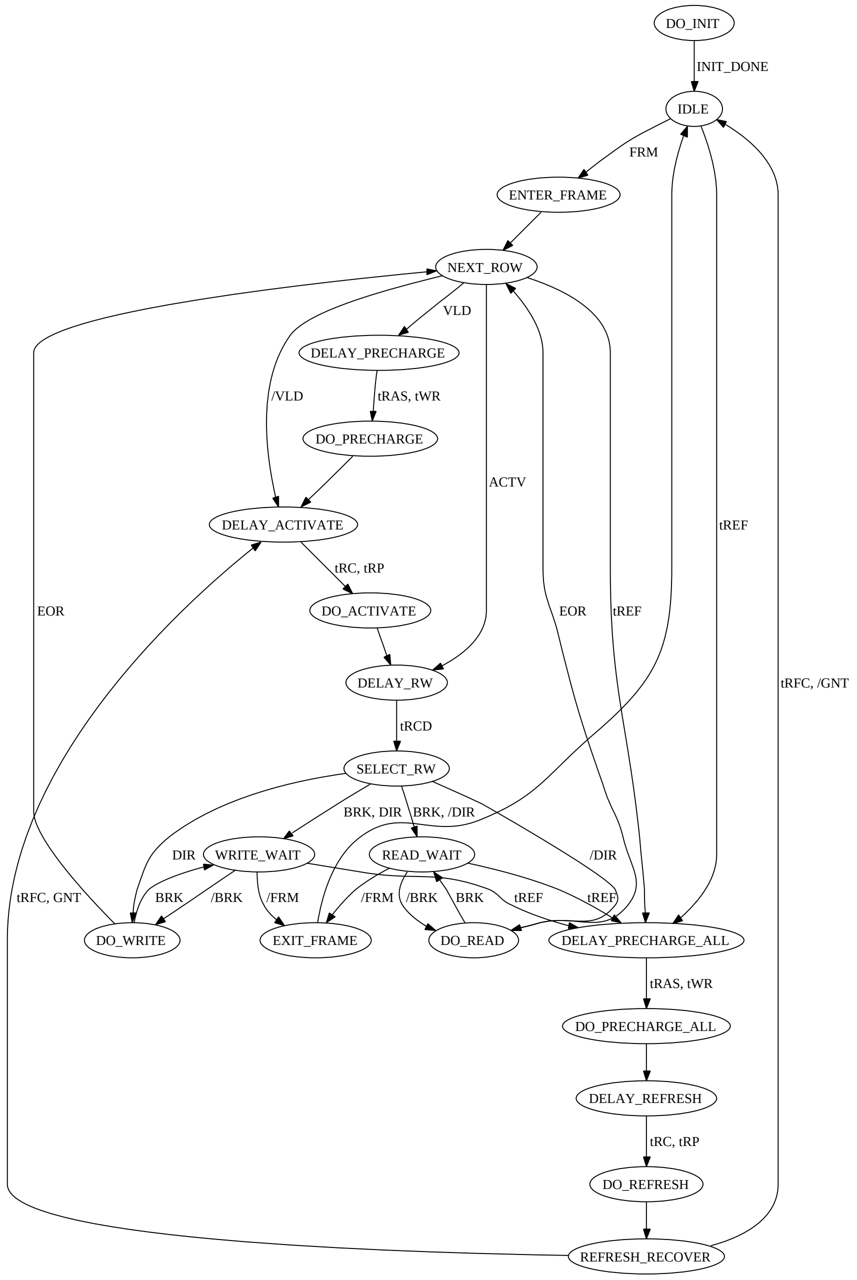


Figure 3.6: Memory controller state machine. The state machine powers up in the `DO_INIT` state. After initialisation, it enters the `IDLE` state, waiting for `FRM` or scheduling the periodic refresh. The refresh sequence can be seen in the lower part, beginning with `DELAY_PRECHARGE_ALL`. The data transfer frame is handled in the part to the left, between `ENTER_FRAME` and `EXIT_FRAME`.

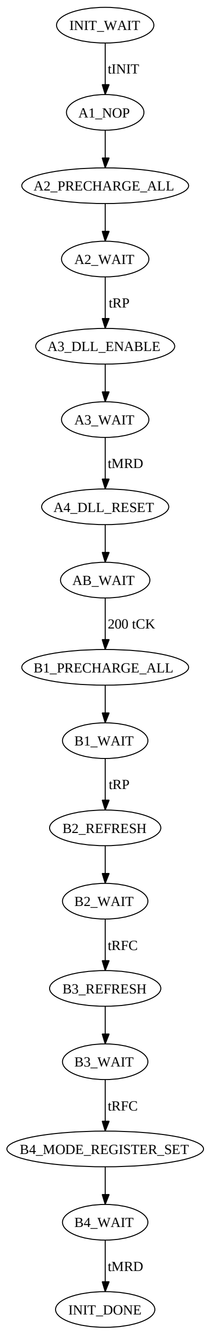


Figure 3.7: SDRAM initialisation sequence. This sequence initialises the SDRAM DLL and sets operational parameters. It is critical to the successful operation of the SDRAM device.

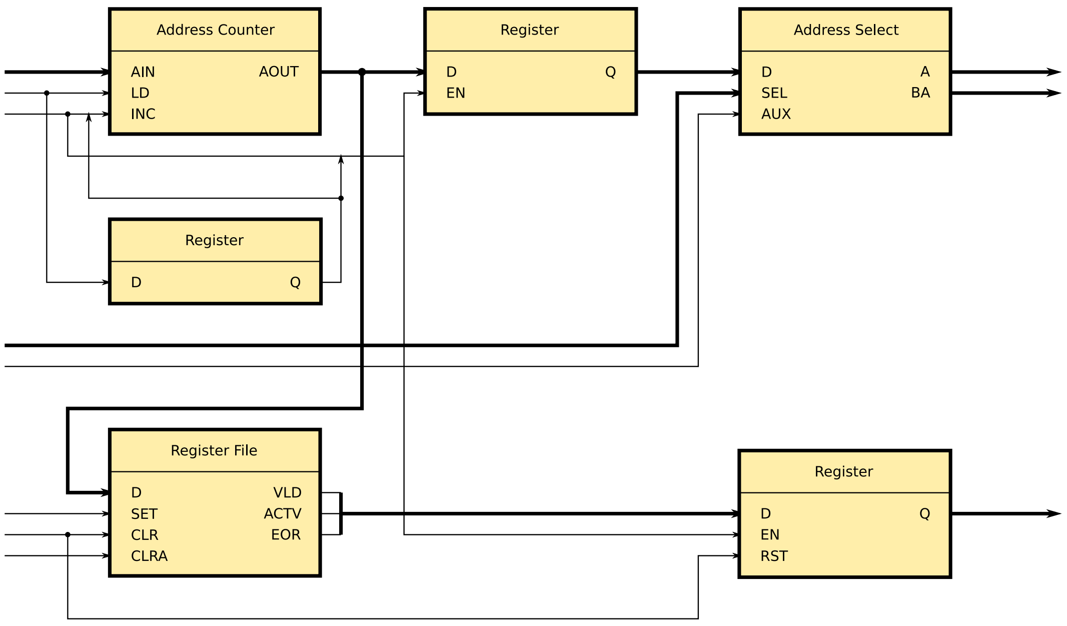


Figure 3.8: Address path block diagram. A two stage counter is used in combination with the active row register file to trigger activation and precharging of rows. The register file stores the active row for each memory bank and compares it to the stage one counter value. From this, control signals for the state machine are generated. `VLD` is high if there is an active row for the current bank. `ACTV` is high if the current row is already active and `EOR` is high when the counter value reaches the end of a row.

register file values and control output signals are generated accordingly to allow the state machine to decide when to precharge or activate rows or end a burst transfer. `VLD` signals a valid register file entry, i.e. the bank has an active row. `ACTV` is asserted when the stage one counter value is equal to the active row register for the bank addressed. Finally, `EOR` is asserted when the counter reaches the end of a row. The register file is controlled through three signals. With `SET`, the stage one counter value is stored in the active row register for the corresponding bank. `CLR` causes the active row register to be invalidated. When the `CLRA` flag is set, `CLR` invalidates the active row registers for all memory banks. Figure 3.8 shows a block diagram of the address path.

The `SEL` port is used to select how the address lines of the SDRAM are driven. There are four possible modes: row address, column address, mode register and extended mode register. The `AUX` flag is dependent on the selected mode. When mode register is selected, `AUX` corresponds to the DLL reset bit, for extended mode register, it corresponds to the DDL disable bit and in column address mode it can be used to flag auto precharge after memory access. In row address mode it is ignored. Mode selection is done by the state machine according to the command submitted.

FPGA clock to output	tOCKQ	0.49 ns
FPGA output to pad	tIOOP	1.98 ns
circuit path	tCP	< 0.25 ns
SDRAM clock to output	tAC	± 0.7 ns
FPGA pad to input	tIOPI	1.51 ns
round-trip time	tRT	4.5 ± 0.7 ns

Table 3.1: Data path delays. The values for the FPGA are taken from [19], and the values for the DDR SDRAM from [10].

The Data Path connects the external bidirectional double-rate-single-width `DQ` data bus to the internal unidirectional single-rate-double-width data input and output ports. Depending on the state of the `DIR` signal, either the input or the output path is active, and the `DQ` and `DQS` ports are tri-stated or driven accordingly.

When `DIR` is low, the input path is active and data is input from the `DQ` data bus. The `VLD` signal simply follows the `EN` signal with a delay of several clock cycles. The `EN` to `VLD` delay can be configured to accommodate different CAS latency values and additional delays due to circuit latency.

It is important to understand the different delays relevant to the data path. When the state machine issues a read command, it is shifted 180° to centre the signals to the rising edge of the clock signal, then delayed another clock cycle in the output registers. It then travels through the output buffers and down the circuit path to the DDR SDRAM chip, where the state machine of the chip registers the command and answers the request with a certain delay called CAS latency plus an additional clock-to-output delay, similar to the delay caused by the output buffers of the FPGA. It's back the circuit path to the FPGA, through the input buffers to a DDR input register. See table 3.1, which lists the delays as seen on the H-RORC. The most important point is, that the data signal most probably will not be aligned to the internal clock of the FPGA design when it finally reaches the DDR input register.

Enter the input delay cells of the Xilinx Virtex™-4 FPGA, which offer an adjustable additional delay for an input signal, and can be used to align the signal to the internal clock of the design. The granularity, called *tap*, of the input delay cell is 74 ps[5], and up to 64 taps can be added to a signal, allowing up to 4.7 ns delay. The input delay cells are basically 64 buffers with a 64-to-1 multiplexer and a bit of additional counter logic. The memory controller design uses a fixed value which has been calculated from the delays above. Of course, this value can be easily adjusted for other hardware.

With a command-to-output delay of 1.5 clock cycles, a CAS latency of 2.5 clock cycles, and an additional delay of 4.5 ± 0.7 ns, the first word of the data should be safely centred around the rising edge of the internal clock used in the memory controller design, which runs at 167 MHz, or a 6 ns period. So there should be no additional taps

[5]Tap 0 is 39 ps.

necessary. In reality, however, the delay proved to be smaller than expected, so a value of 15 was used for the tap count.

When `DIR` is high, the output path is active and the `DQ` data bus and `DQS` signals are driven according to the `EN` signal.

3.3.3 Simulation

Since the DDR SDRAM controller is a rather complex component, using simulation to verify functional operation and fulfilling of timing requirements is indispensable. For simulation, Mentor Graphics ModelSim® SE 6.2 was used. Figure 3.9 shows the results of simulating start up and write access to the memory. While the initial 200 µs are skipped, you can clearly see the initialisation cycle, transfer handshake and two consecutive blocks of back-to-back burst transactions.

Figure 3.10 shows the start of the first write burst. While this is not easy to understand at first, the `REQ` signal is asserted when the write command is issued, valid data has to be supplied one clock cycle later, so the first double-word written to the memory is 0xAFFED00F. Also remember, that the output control signals are shifted 180° to align them centred to the rising edge of the clock signal. The same is true for the `DQ` data output relative to the `DQS` strobe signal.

While all sub-components and write access were simulated and verified thoroughly before incorporating the design to the FPGA, read access is not as easy to simulate, as it requires a working memory model. Such models exist, but simulating with these is not without problems. Since the exact timing is not necessarily known, as already described previously, some testing has to be done in the FPGA, *in vivo*. So verification of read transactions was postponed to that stage.

3.3.4 Integration and On-Chip Tests

Integration of the DDR SDRAM controller on the H-RORC is done with Xilinx ISE Project Navigator using a *User Constraints File* to connect top-level output ports to FPGA pins and set timing and area constraints. After synthesis, placing and routing the design, a first comparison with the DDR SDRAM controller provided by Xilinx can be given. Both designs can be used with clock frequencies up to 200 MHz and more. The controller developed for this thesis uses 201 slices on the FPGA, while the Xilinx controller uses 944 slices [22]. To be fair, the Xilinx controller allows automatic self-configuration for the read timing—a feature which is not needed for the H-RORC and other designs where timing can be calculated or measured and doesn't change during operation.

To verify the memory controller design, but also to test the DDR SDRAM chips and circuit paths on the H-RORC board, several testbench designs were used. Data mismatch in the testbench designs triggers a sticky error flag which is output to a LED on the H-RORC board.

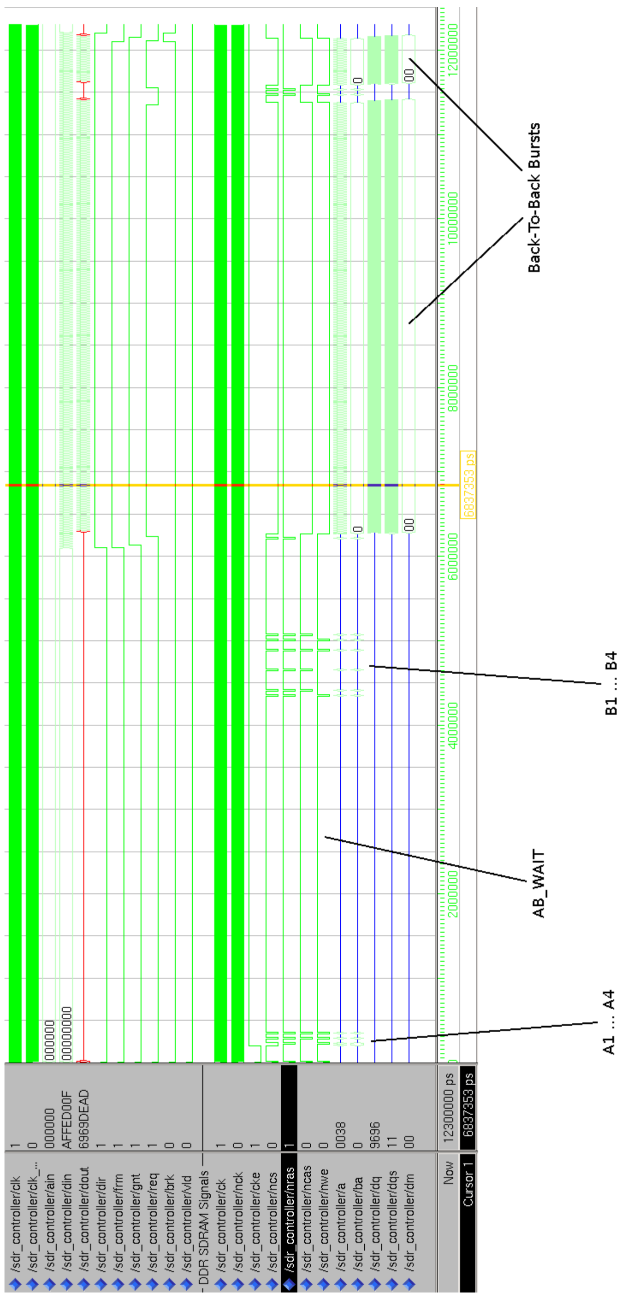


Figure 3.9: Overview of a simulated write access to the memory. The first part shows the initialisation sequence with corresponding values in the command lines NCS, NRAS, NCAS and NWE. In the second part, two consecutive back-to-back burst transfers can be seen. The pause in between is due to row activation at the end of a row.

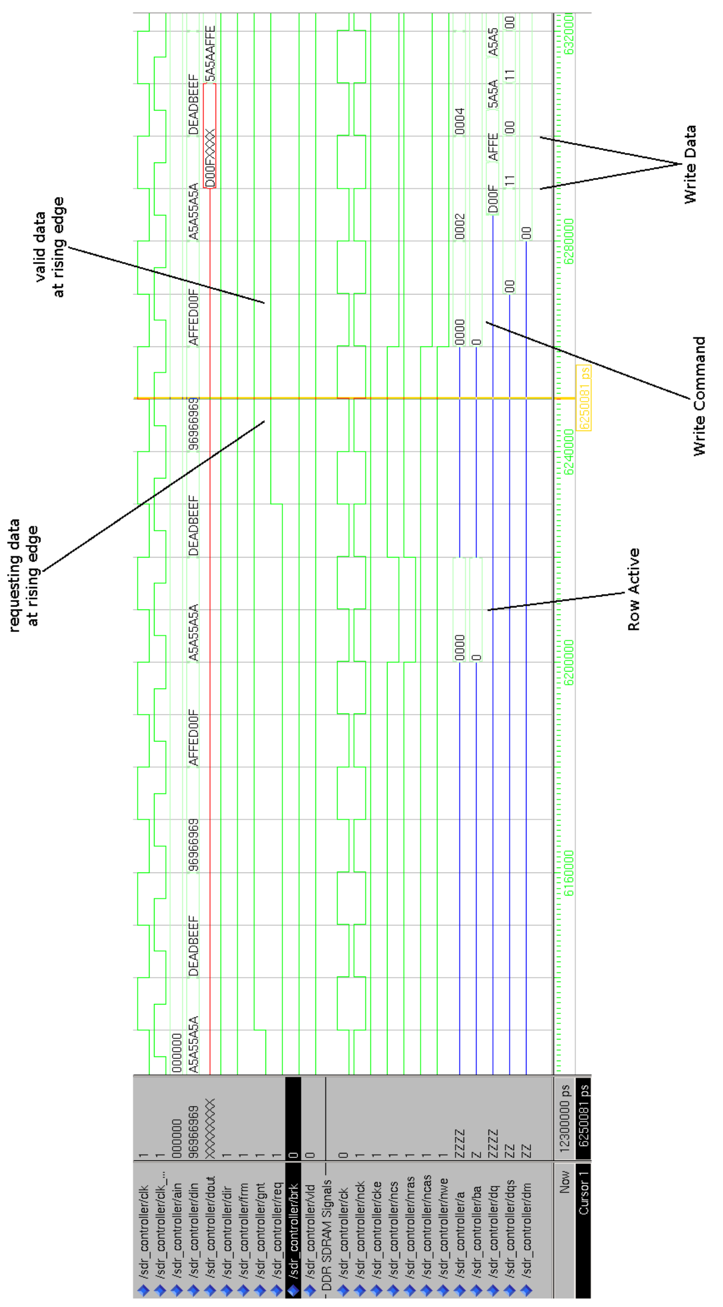
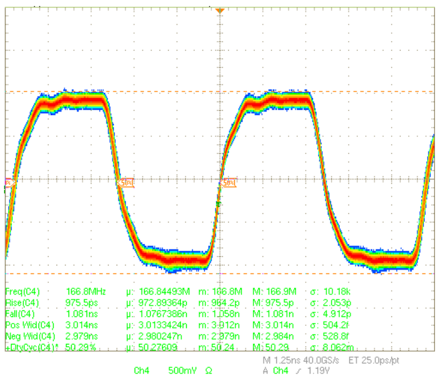
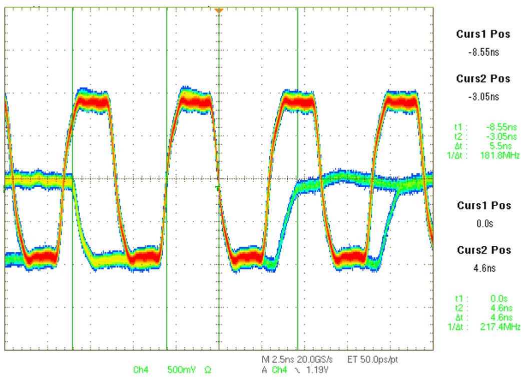


Figure 3.10: Start of the first block of a back-to-back burst transaction. The cursor is positioned at the rising edge of the clock cycle after `REQ` goes high. Valid data has to be provided one clock cycle later. All signals are shifted by 180° to centre them around the rising edge of the clock.



```
Freq(C4)       166.8MHz   μ: 166.84493M   m: 166.8M   M: 166.9M   σ: 10.18k
Rise(C4)       975.5ps    μ: 972.89364p   m: 964.2p   M: 975.5p   σ: 2.053p
Fall(C4)       1.081ns    μ: 1.0767386n   m: 1.058n   M: 1.081n   σ: 4.912p
Pos Wid(C4)    3.014ns    μ: 3.0133424n   m: 3.012n   M: 3.014n   σ: 504.2f
Neg Wid(C4)    2.979ns    μ: 2.980247n    m: 2.979n   M: 2.984n   σ: 528.8f
+DtyCyc(C4)*   50.29%     μ: 50.27609     m: 50.24    M: 50.29    σ: 8.062m
```

(a) Clock signal



```
Curs1 Pos
-8.55ns
Curs2 Pos
-3.05ns
t1 :    -8.55ns
t2 :    -3.05ns
Δt :    5.5ns
1/Δt :  181.8MHz

Curs1 Pos
0.0s
Curs2 Pos
4.6ns
t1 :    0.0s
t2 :    4.6ns
Δt :    4.6ns
1/Δt :  217.4MHz
```

(b) DQS signal

Figure 3.11: Level stability and jitter of the controller signals. Averaging the Clock and `DQS` signals over several seconds makes jitter and level stability visible. The signal quality of the controller is more than adequate.

The Address Line Verification Testbench writes a unique value to each memory location. This is done simply by writing the address counter value. Since the data port is wider than the address port, an additional counter value is written to the most significant bits of the word, which is incremented each complete run of the testbench. After the complete address space is written, the contents of the memory are read back and compared to the counter values. Then the run counter is incremented and the testbench starts over again.

The Data Line Verification Testbench writes different test patterns to the memory, to test for crosstalk effects on the `DQ` data bus lines. As with the address line verification, the whole address space is written before reading the values back.

The Memory Stability Verification Testbench is derived from the address line verification testbench, but introduces a successively increasing delay between writing to the memory and reading back. This tests the memory for symptoms of amnesia, resulting from refresh problems or defects in the chips.

Memory stability has been successfully verified to a delay of a few days between write and read access.

The Signal Quality can be measured using a high-end digital oscilloscope. Figure 3.11 shows the signal level stability and jitter for the clock and the `DQS` signal during write access. For these measurements, signals have been recorded for several seconds with a Tektronix TDS 7254.

The level stability is about 100 mV and the clock jitter is less than 125 ps. The signal width is 3.0140 ± 0.0005 ns high to 2.9790 ± 0.0005 ns low. This is all well within the chip specification given in [10]. In the right picture, the `DQS` signal is shown. The strobe preamble can be seen in the left part of the picture, the postamble in the right part. These result from the controller taking and releasing the bidirectional `DQS` signal. Since the DDR SDRAM uses the SSTL_2 signal specification, signals which are not driven take the reference voltage between a logic low and high. The preamble has a duration of 5.5 ± 0.1 ns and the postamble 4.6 ± 0.1 ns. However, the postamble has to be measured without the rise time of the signal, since no reference level is given for the timing specification in the documentation. The controller stops driving the bus after less than 3.5 ns, thus fulfilling the specifications.

The Memory Performance can be determined using any of the testbench designs mentioned above simply by monitoring the `DIR` signal, which represents the access mode. With every period of `DIR`, 32 MB of data is written and read back for a single DDR SDRAM chip on the H-RORC board. The data rate for linear access using full-row burst access using either the address line or data line verification testbench is $608 \pm 1\,{}^{MB}\!/\mathrm{s}$, which is 95.6 % of the raw memory data rate, or 99.9 % of the highest possible memory data rate. The *raw* memory data rate is given only by the clock frequency and is the data rate that can be achieved in a single burst, without any overhead. The *highest possible* memory data rate takes into account row activation and precharge which cost a total of nine clock cycles and is due each block access, and periodic refresh which costs twelve clock cycles and is due every 1300 clock cycles at 167 MHz.

Figure 3.12 shows the relative performance of the controller versus the highest possible performance for random access, dependent on the block size. Note that even with the largest block size possible, the performance still is worse than for linear access. This is because of the overhead for the controller state machine each time a new frame is requested. The plot for the memory controller takes into account the number of clock cycles from the state machine.

The Temperature Pattern for the FPGA under full load has been one of the major concerns since the early development stages. As is illustrated by the photo in figure 3.1 at the start of this chapter, the design of the H-RORC is very compact. There is little space left for air flow along the surface of the heat sink mounted on the FPGA when the board is fully equipped with two DDL interfaces. With too little air flow, not only the FPGA but also the DDR SDRAM chips will run very hot and could possibly be damaged. To ensure healthy operation of the board, it was tested using a design operating at maximum load of all four DDR SDRAM chips in one of the cluster nodes *to be*, lid closed. Figure 3.13 shows temperature build-up and cooling down after shutting the host computer down. The graph demonstrates the impact the DDL interfaces have on the temperature level. While the temperature in the configuration with two DDL interfaces settles around 47 °C, a naked board will operate at around 38 °C.

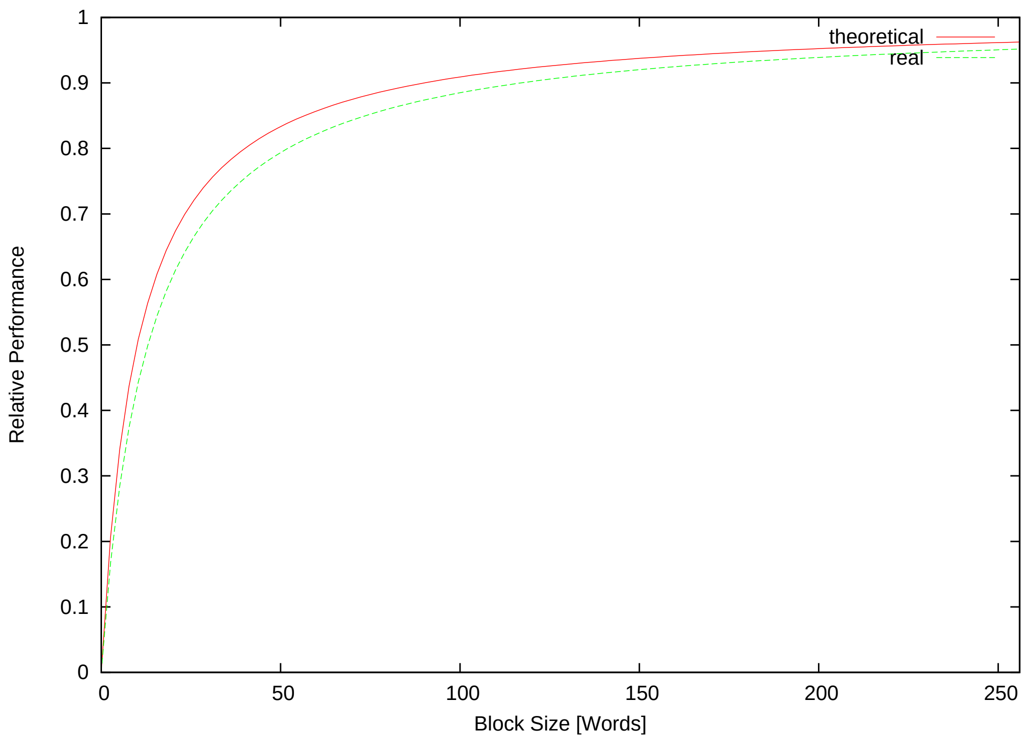


Figure 3.12: Relative performance of the memory controller. The red curve shows the maximum relative memory performance theoretically possible for purely random access to the memory depending on the size of contiguous blocks. The green curve shows the real performance of the controller as calculated from the state machine.

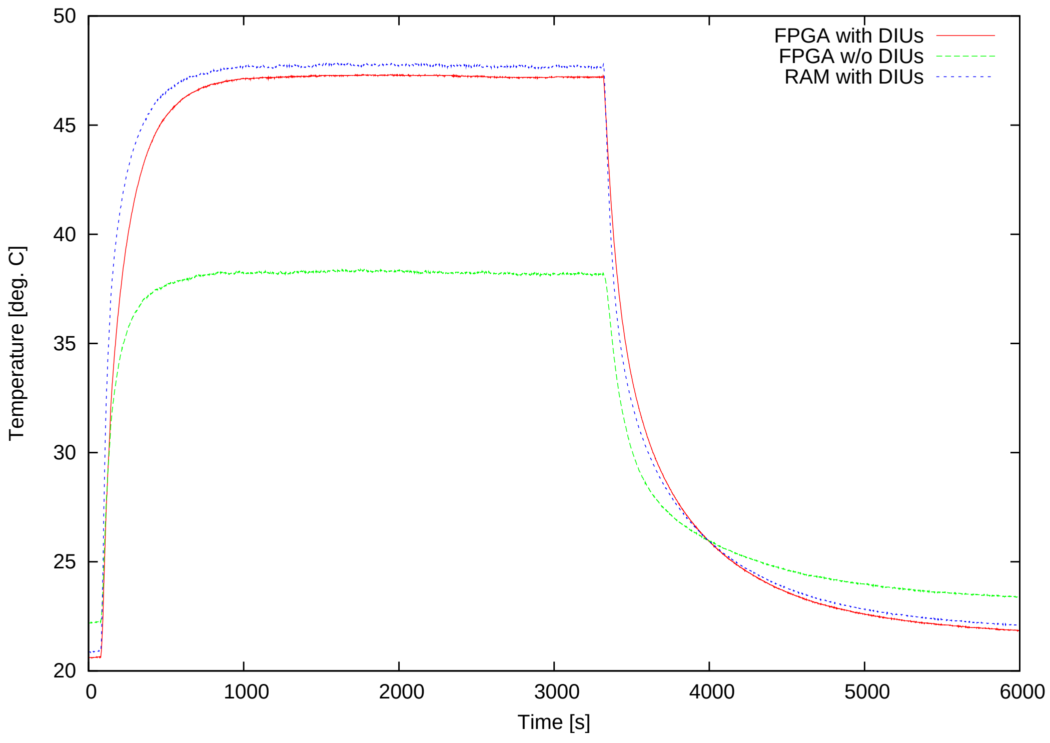


Figure 3.13: Temperature pattern of the H-RORC. The temperature is measured using PTC resistors located between the fins of the heat sink mounted on the FPGA. As can be seen, the temperature level on a board equipped with two DDL interfaces is significantly higher than on a naked board. Note that the base levels suggest a systematic error of about 1 °C for the graphs. The steeper slope of the temperature graph for the DDR SDRAM chips results from the lack of a heat sink on those devices.

Home computers are being called upon
to perform many new functions,
including the consumption of homework
formerly eaten by the dog.

Doug Larson

4 The Hardware-Software Interface

Whether the *HLT Read-Out Receiver Card* (H-RORC) works in pass-through mode or preprocesses event fragments to find clusters or applies Hough transformation, at some point the data has to be transferred to a computer system for further processing and finally combining event fragments to events. To allow maximum performance, data is injected via *Direct Memory Access* (DMA) into the main memory of the host computer. The memory must be reserved for this purpose, of course, and the computer must be able to communicate with the H-RORC. On the hardware level, the *Peripheral Component Interconnect* (PCI) bus is used. On the system level, the *PCI and Shared-memory Interface* (PSI) allows reserving of memory for DMA and communication with the H-RORC. Figure 4.1 shows an overview of the hardware-software interface.

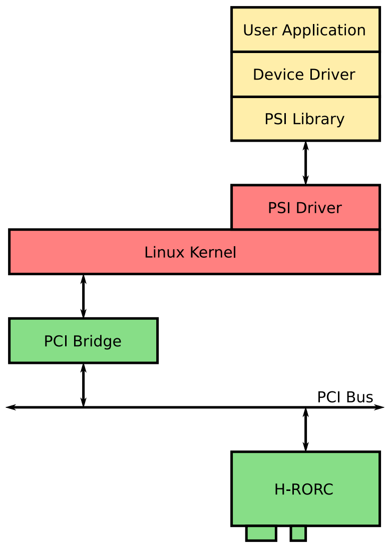


Figure 4.1: Overview of the hardware-software interface. The PSI driver allows user mode applications to access the PCI bus using functions provided by the Linux kernel. It works as an abstraction level between different kernel versions and the hardware dependent part of user applications.

4.1 The Peripheral Component Interconnect Bus

The PCI bus was developed in the early nineties of the last century and replaced the *Industry Standard Architecture* (ISA) and *VESA Local Bus* (VLB). For over ten years, it has been the standard bus system for personal computers, allowing the CPU to communicate with peripheral devices, from on-board components integrated in the mainboard chipset to extension cards located in slots on the mainboard. It is now superseded and slowly being replaced by PCI Express.

PCI uses a parallel bus architecture with either 32 or 64 bit width at 33 or 67 MHz. Address and data share the same physical lines and are interleaved. Every PCI device can use up to four interrupt lines which are mapped to corresponding interrupt service requests by the PCI bridge and system software. The H-RORC uses the 3.3 V, 64 bit PCI at 67 MHz, thus allowing a raw transfer data rate of 533 MB/s. Several buses can be interconnected and hierarchically structured with bridge devices. Up to 256 buses are allowed with a maximum of 32 devices for each bus. A single device can be divided into 8 different sub-devices called *functions*. For each function a device supports, a separate 256 byte size *configuration space* must be provided. The configuration space mediates vital information about the device function, like the general functionality provided, identification and status information. It is also used to configure the *Base Address Regions* (BAR) and interrupt service requests for a device function. Each device function can have up to six BARs, which can either provide memory or I/O functionality. A device might for example provide internal configuration and status registers in an I/O region or frame buffer functionality through a memory region. The system software determines how many BARs any device function requires and the requested size. It will then map these regions and store the addresses in the corresponding configuration space registers.

The most important point to understand here is that the configuration space and especially the BARs play a central role in communicating with a device. A device driver typically looks for a certain combination of vendor and device identification and starts communicating with the device by reading from and writing to BARs, either by using direct or memory mapped input/output.

For full details on the PCI specification see [16] or later versions thereof.

4.2 The PCI and Shared-memory Interface

The PCI and Shared-memory Interface was initially developed by Timm M. Steinbeck during the work for his PhD thesis [18]. It is a Linux kernel driver and accompanying library, allowing user space applications direct access to the PCI bus and computer memory. The concept allows device driver functionality to be implemented—and debugged—in user mode.

The second generation of the PSI tools started as a crude port of the kernel driver to the 2.6 series of the Linux kernel and ended up as a complete rewrite of the driver. The internal structure of the driver was changed to do resource management and allow additional features and transition of concepts which have changed with the kernel architecture.

This section will give an overview of the Linux kernel driver. For a description of the library, see chapter B in the appendix.

4.2.1 The Linux Kernel Architecture

The Linux kernel is the core, and the namesake, of the Linux operating system family. It is a monolithic kernel, with device drivers and kernel extensions running in kernel mode. It provides support for preemptive multitasking, both in user mode and kernel mode, virtual memory, shared libraries, demand loading, shared copy-on-write executables, memory management, multithreading, symmetric multiprocessing, and comes with a powerful set of network protocol stacks. Despite the monolithic approach, support for loadable modules allows very flexible extension of hardware support and kernel functionality at runtime.

A loadable module for the Linux kernel has to implement at least two functions, an `__init` function which is called upon loading the module, and an `__exit` function which is called when the module is removed. For a device driver, the `__init` function will typically register a *device* with the kernel, so that the *device interface* can be accessed through a *device node* later. The device interface consists of a number of additional functions which are called when the device node is opened or closed, accessed or when control functions are requested. The device node requires the *major* and *minor device number* for the device interface. The major number roughly distinguishes between different classes of devices, and the minor number is used to address a single device instance. The major number can be allocated dynamically when registering a device with the kernel or it can be statically given either implicitly by registering a device of a certain class or explicitly by convention.

Another way of allowing user mode applications to communicate with the driver is to export virtual files in one of the `/proc` or `/sys` subdirectories. The former is being obsoleted by the latter which is available since version 2.6 of the kernel. This feature is not used by the PSI driver however.

For a detailed introduction to writing device drivers for the Linux kernel see [9], and [12] for information on porting from 2.4 to 2.6 versions of the kernel. Another very valuable resource on the Internet is the Linux kernel cross-reference [11].

4.2.2 The Linux Kernel Driver

In a traditional sense, a device driver is a kind of black box, providing a well-defined programming interface to a specific hardware device or class of devices. On the other hand, a kernel extension provides additional functionality that is not tied to such specific hardware. While drivers typically export device interfaces that can be accessed through device nodes in the `/dev` tree by user mode programs, kernel extensions add new API routines to the kernel and provide system calls.

The PSI driver is a crossover between the two concepts. It is not tied to specific hardware other than the PCI bus itself, and extends the kernel to allow user mode drivers for PCI devices. Still it exports its API as a device interface that can be accessed through `/dev/psi`. For kernel version 2.4 with support for the dynamic device filesystem, *devfs*, the major number will be allocated dynamically and the device node will be created automatically when the module is loaded. Otherwise, the static major number 100 is used and the device node has to be created manually. For kernel version 2.6, a miscellaneous device is registered and the device node is created automatically by the kernel hotplugging mechanism when the module is loaded[1].

The concept of user mode drivers is not new to Linux. It has been used by the X server for a very long time. As of the current version of the kernel, however, there is still no consistent support for user mode drivers for PCI devices. The benefits of such a concept are evident in an environment where a lot of custom hardware is developed by different teams. By introducing an additional layer of abstraction between the constantly evolving and changing kernel API and user mode programs, only the abstraction layer has to be actively maintained, allowing the teams to update their software to new kernel versions without much hassle. The applications targeting custom hardware can tightly integrate a driver for the custom hardware by means of a library or directly. Debugging a user mode driver is a lot easier than debugging kernel mode code.

The PSI driver supports only the input/output control, or *ioctl*, device interface for `/dev/psi`. Ioctls are considered deprecated, still the intention was to remain as backward compatible as possible, and the ioctl device interface already proved to be a fairly straight-forward approach to the problem in the first generation of the driver.

The second generation driver developed for this thesis has many advantages over the first generation. The different parts of the driver have been completely separated. Module management, device management, device interface and driver backend are implemented as separate files with well-defined, tight interfaces. This allows to change each part of the driver without breaking code in other parts. For example, without too much effort additional interfaces may be supported in future versions of the driver. The driver keeps track of each and every request. This has completely eradicated strange behaviour often observed with the first generation driver. Block requests are handled more efficiently. This results in a speed-up for most projects which have been using the first generation driver. Many features have been added, including support for 64 bit architectures, support for machines using input/output virtualisation, hardware interrupts and basic support for reconfiguring PCI devices after a firmware update without reboot. The driver source files are located in the `src/module` subdirectory.

Figure 4.2 shows an overview of the PSI tools architecture. The arrows show logical connections. For example, the PSI library accesses the driver only *by means of* the

[1]Only the latest versions of the PSI driver use the misc device interface, since the manual triggering of the hotplugging mechanism stopped working as of kernel version 2.6.17 without any conceivable reason, i.e. `misc_register()` handles it the same way it was implemented before [*sic*].

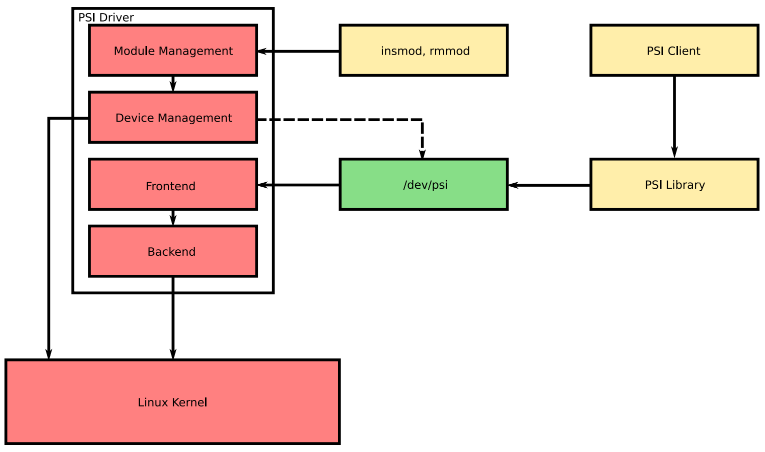


Figure 4.2: Overview of the PSI tools architecture. The module and device management introduce the PSI driver to the Linux kernel, creating the device node `/dev/psi`. The PSI library communicates with the driver frontend through the device node. The driver backend handles the requests from the frontend via the kernel API.

device node. It calls system level functions passing a handle to the device node. In turn, the kernel calls the device interface functions. Much the same is true for the creation of the device node. This is either triggered by the device management of the driver or by explicitly calling `mknod`. The device node is created by the kernel. However, the logical connections serve to show the working principles, showing a complete call graph would only be confusing without making matters any clearer.

The Driver Backend works with a concept called *regions*. A region can be a contiguous chunk of the main memory of the computer, the configuration space or a BAR of a PCI device. Memory regions are identified by a unique name or the corresponding physical address, configuration space or base address regions are identified using PCI bus addresses. Once a region has been created, it has to be *sized*. This is done automatically for regions that have a fixed size, like the configuration space and base address regions. Memory regions, however, have to be explicitly sized before they can be used. Sizing a named memory region is synonymous with allocating a contiguous memory block. To allow allocation of very large contiguous blocks, the PSI driver can use the bigphysarea[2] kernel patch. Physical memory regions have to be sized, too, even though this is more of a symbolic act, since access to physical addresses is not subject to any memory management whatsoever. Use of physical memory regions is strongly discouraged and is only kept for backward compatibility. Note that a region

[2]The bigphysarea patch allows to pre-allocate a large memory contingent that can be used later by kernel drivers as DMA buffer without the need for scatter-gather lists. Since the bigphysarea patch has no active maintainer, updates pop up randomly over the Internet.

can be sized only once. If different processes share data by accessing the same region for example, only the first process actually *creates* the region. If the region has to be sized explicitly, it has to be done at this instance.

The backend is implemented in the files `region.h` and `region.c`. There are several possibilities to access a region once it has been created and successfully sized. The backend supports single byte, word or double word access or block access with corresponding granularity. Alternatively, memory regions and memory BARs of a PCI device can be mapped and accessed just like a regularly allocated memory buffer.

The region handling includes two-fold house-keeping. A global list holds all regions created and keeps track of DMA mappings or assignment of interrupt handlers. Each entry to the region list in turn has a list of each and every process that uses the region. Access to a region is granted only to processes which have obtained a valid handle. For each process, mappings are maintained separately. While this imposes some overhead to region access, this goes unnoticed when using block or memory mapped access, and it allows the backend to release regions which have not been closed, for example because the process using a region hangs or has died unexpectedly. More important even, this may be used in a policy to minimise the security risk imposed by the driver itself, since it allows access to almost any resource of the computer.

The Driver Frontend implements only the ioctls that form the device interface, as of now. These are implemented in the files `include/psi_ioctl.h`, `ioctl.h` and `ioctl.c`. Parameters passed to the backend are thoroughly checked for consistency. In addition to the region handling, some functionality has been implemented as dedicated ioctls. This contradicts the claim of complete separation of the parts of the driver and will be left for future maintenance.

These additions include utility ioctls to save and restore the configuration space of a device to or from a user space memory buffer, remove and re-insert devices from or to the kernel device list, respectively, and active waiting with sub-microsecond granularity, depending on the hardware platform. While removal and re-insertion of devices provide support for basic hotplugging, they only work if the PCI device uses the same physical layout for the BARs. Changes to the BARs involves reconfiguration of the PCI bridge(s) of the computer, which is non-trivial and may not work on all, or even any, configurations without hotplugging support by the computer firmware.

Module and Device Management is implemented in the files `module.h`, `module.c`, `device.h` and `device.c`. Module insertion and removal is handled here, together with registering and unregistering the device interface and creation and removal of `/dev/psi`.

The most exciting phrase to hear in science
the one that heralds new discoveries
is not "Eureka!" but "That's funny . . . "

Isaac Asimov

5 Summary and Future Prospects

In the previous two chapters, two essential components for the data flow in the *High Level Trigger* (HLT) of *A Large Ion Collision Experiment* (ALICE) developed for this thesis were presented. Both components have been implemented and tested and are already in use by the time of this writing. The **DDR SDRAM controller**, for example is not only used by the HLT group at the *Kirchhoff-Institute for Physics* (KIP), but also at the *Istituto Nazionale di Fisica Nucleare* (INFN) in Cagliari, Italy. The second generation of the **PCI and Shared-memory Interface** (PSI) has been quickly adopted by practically every group developing custom hardware at the KIP, but is also used by other groups at the Universities of Karlsruhe and Frankfurt.

5.1 The Data Reformatter

While the memory controller presented in chapter 3 so far is mainly used to verify the DDR SDRAM chips on the *HLT Read-Out Receiver Card* (H-RORC) board, this is of course not its sole, nor main purpose. The memory has been included in the board design to allow preprocessing tasks which require temporary storage beyond the limited capabilities of the *Field Programmable Gate Array* (FPGA) used.

Since the H-RORC deals with event fragments, it is essential for any preprocessing that the single data points have defined locations in time and space. The *Read-out Control Unit* (RCU) collects the data from the detector *Front-End Electronics* (FEE) and delivers it through the *Detector Data Links* (DDL) in a rather inconvenient way for further processing. Also, the data is formatted in a way which makes it necessary to hold a complete event in memory. Thus, the necessity arises to sort and reformat the data in the H-RORC, otherwise there will be no preprocessing in hardware, wasting the potential of such an approach—and wasting the time and effort put into the development of the H-RORC in the first place. At the beginning of chapter 3, event fragments were already described shortly. To recap the layout, figure 5.1 gives an overview of how the TPC is divided into sectors, patches and pads. Note that the pads are organised in rows, with less space between two pads of the same row than between two rows. This means that any algorithm searching for groups of peaks in the data will most likely want to work on a pads within a row rather than different rows.

An event fragment combines the data packets from the pads of one patch. If zero suppression is used, only those pads will be included which have signal data values above the electronics baseline. Sorting the data has to be done on the pad data packet level. The pad data packets are embedded in an event data frame. The event data

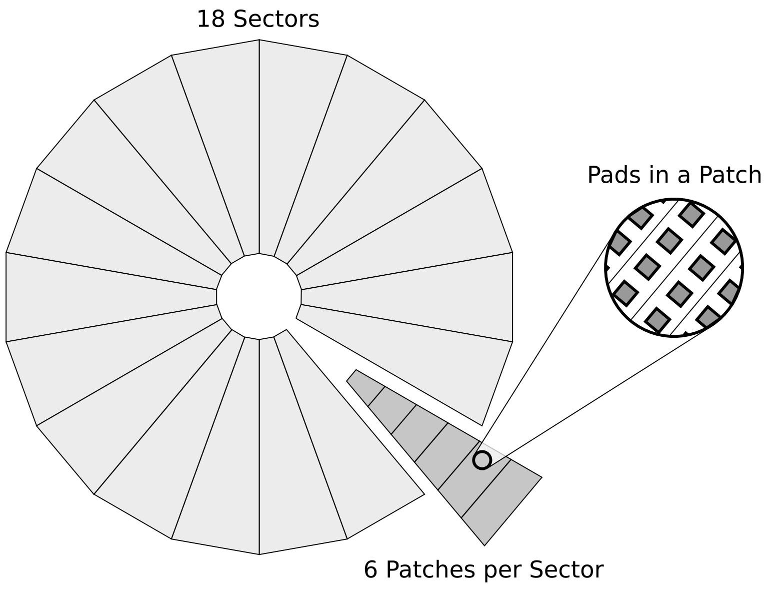


Figure 5.1: TPC layout. Both ends of the *Time Projection Chamber* (TPC) contain read-out electronics organised into sectors, patches and pads as illustrated above. The H-RORC receives the data from one patch per DIU.

frame consists of the *Common Data Format header*, which is eight 32 bit words in size, a variable number of pad data packets and the *RCU Data Block*, which is up to three 32 bit words in size.

It is important to pay attention to the different word sizes used in the data. While the event data frame uses 32 bit words, the pad data packets use 40 bit words which will be padded to be aligned to a 32 bit boundary, so they fit into the event data frame. Inside the pad data, 10 bit words are used and padded to be aligned to a 40 bit boundary.

The RCU Data Block contains the total number of 40 bit words used in the pad data packets. The last[1] 40 bit word of a pad data packet contains the number of 10 bit words used in the pad data and the hardware address of the pad. The latter is needed to look-up the location of the pad, the former to calculate the position of the previous pad data packet in the event data frame. Figure 5.2 shows the layout of a sample event data frame and pad data packet.

This layout has serious impact on the way event fragments have to be processed. Since the event data frame has to be read from back to front, the whole event must be stored in memory. The size of an event data frame varies, but in most cases it will be too large to fit within internal memory of the FPGA. While working on one event fragment, further event fragments may be pending. To assure event fragment

[1]Not including words appended for padding.

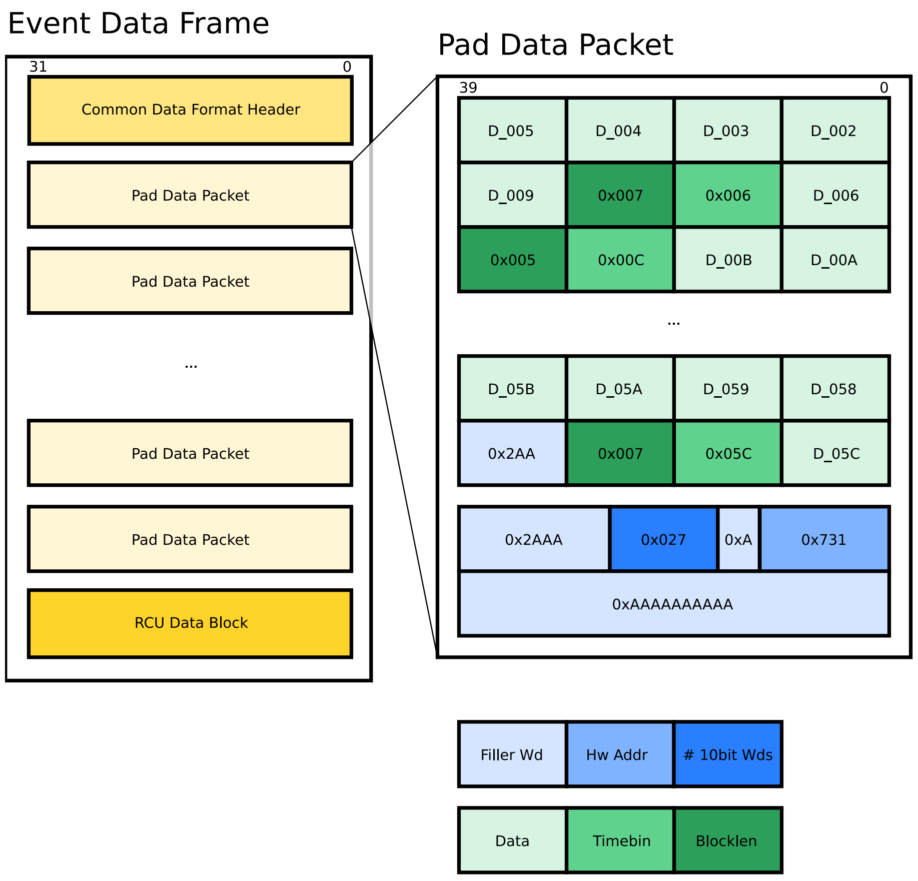


Figure 5.2: Format of TPC event data frame. The figure above shows how an event data frame contains header information, pad data packets and footer information. A sample pad data packet is shown to illustrate the data layout within a packet.

processing without loss of a single event, a flexible double-buffering scheme has to be implemented which allows a varying number of events to be written in one buffer, while the other buffer is still being processed. It may seem tempting to use a simpler approach and store only a single event data frame per buffer. However, if a rather large event fragment is being sorted while a rather short event fragment is coming in, this will lead to the next event fragment pending to be discarded. Since the H-RORC hosts two DIUs, all four DDR SDRAM chips will be used permanently.

For convenience, the back linked structure of the data can be turned upside-down almost literally very easily. With very little effort, the address path of the DDR SDRAM controller can be changed to allow burst access to the memory in both directions. Simply by adding a control signal that enables decrementing the address counter instead of incrementing, and adjusts the comparator generating the `EOR` signal, the event data frame can be written to the RAM in forward mode and read in reverse mode. This will make the structure of the data look like being forward linked.

5.1.1 Discussion of Feasibility

In the case given, sorting is a three phase process. In phase one, the data to be sorted is written to a buffer. Phase two is scanning the buffer, mapping the hardware address of each pad data packet to an address slot and recording its memory addresses to that slot. Finally, the recorded addresses are used to read the pad data packets and deliver the data to the next preprocessing stage. The sorting process is pipelined using double-buffering as described earlier, so phase one always is done in parallel with phase two and three.

It is immediately clear, that phase two and three have to be accomplished, in average, at least with the same rate as receiving the data in phase one. If one buffer receives data faster than the other buffer is sorted, sooner or later a buffer overflow will result in events not being sorted or, even worse, being discarded. It is also clear, that the memory data rate defines the upper limit for sorting. This is, however, a very critical point in the discussion of feasibility for sorting the data in hardware. Since there is a certain overhead for accessing data stored in memory, the data rate strongly depends on the access pattern. Linear access making full use of bursts results in a memory efficiency of about 95 %[2]. Random access performance is a lot worse. In fact, for single word access it is less than 7.1 %[3].

Phase one and two can be implemented using linear access to the SDRAM[4]. Phase three requires random access to the SDRAM, thus memory performance is dependent on the size of the pad data packets.

From the discussion above it should be understood that the average memory data rate for phase two and three together has to be at least twice the incoming data rate in

[2]This is due to row precharging and activation and refresh cycles.

[3]State switching for a single word frame requires 14 clock cycles, not including the overhead due to refresh cycles.

[4]The address path has to be extended to allow access in both directions, however, as described earlier.

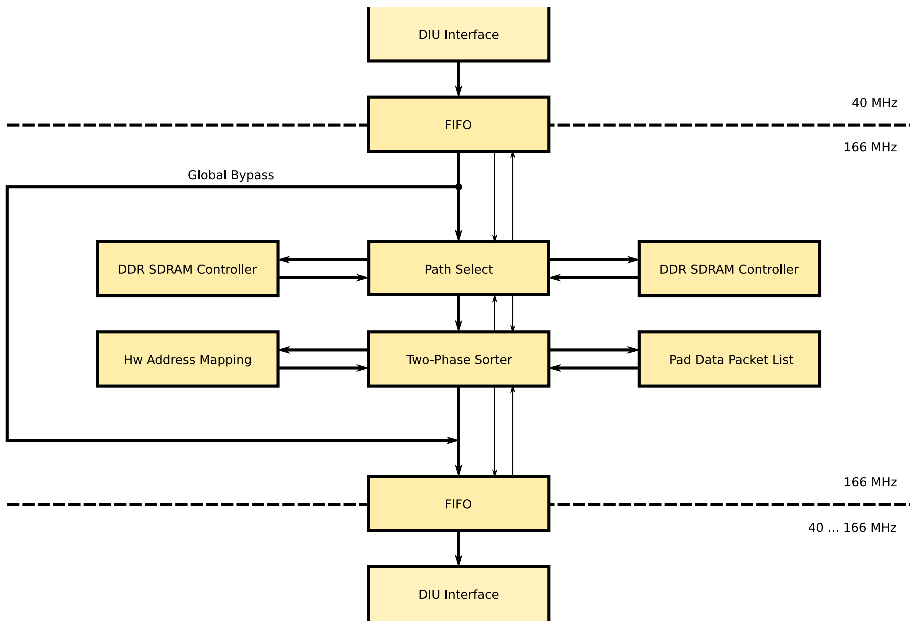


Figure 5.3: Overview of the data reformatter. The data reformatter can be implemented in four sub-components and will instantiate two DDR SDRAM controllers. The path selection handles double-buffering of incoming event data and distribution to the two phase sorter. The latter uses hardware address mapping and the pad data packet list to reorganize packets in the event data frame. An input and output FIFO will be used to handle crossing of clock domains and buffer event data. The FIFOs will be connected to DIU style interfaces, so the component can be inserted before any component attached to a DIU.

phase one. In the current implementation of the H-RORC firmware, the incoming data rate ρ_i is 40 MHz, and the raw memory data rate ρ_m is 166 MHz. So the actual lower limit for the memory performance $\eta = \rho_i/\rho_m$ is 24 %. The overall memory rate has to be shared by phase two and three. Since phase two can be implemented using linear access, the local memory performance will be around 95 %, which leaves around 32 % for phase three. This corresponds to an average pad data packet size of at least 6.1 32 bit words. These values should be regarded only as rough estimates. Still, real-world data has to be analysed and compared to these limits.

5.1.2 Suggested Implementation

In this section so far, the design for the data reformatter has already been outlined. To sum up the suggested features, figure 5.3 shows an overview of the component.

The path select sub-component handles the double-buffering. It communicates with the two-phase data sorter sub-component to determine when to switch buffers. The two-phase data sorter in turn communicates with the hardware address mapping and pad data packet list sub-components, both of which are mostly large look-up tables. The hardware address map is statically initialised for the patch the H-RORC is working on, and the pad data packet list is filled dynamically in the first sorting phase.

The data reformatter knows two bypass modes, a global bypass and an error bypass. When global bypass is active, the data reformatter is idle and event data frames are simply passed through two FIFO stages. The error bypass mode is entered, when the first sorting phase encounters inconsistencies it cannot resolve while scanning the data. Such inconsistencies may arise from the data format specification for the pad data packets, since there may be cases when a filler word cannot be distinguished from a valid trailer word containing the hardware address and number of 10 bit words used, and the first 40 bit word containing data looks like a trailer word.

This might happen, if the hardware address is 0xAAA and the number of 10 bit words used in the pad data packet is 682 (0x2AA). In that case, one filler word is appended after the trailer word, and the data inside the pad data packet will be padded with two 10 bit words. So, only the value of data bits 12 to 15 of the 40 bit word preceding the trailer word in question decides whether these last 40 bits of data look like a valid trailer word. Even then, a consistency check can be applied, since the number of 10 bit words determines the number of filler words appended after the trailer word, which would have to be two.

While this can happen theoretically, it has to be observed whether it is of any statistical relevance. If it turns out that this problem must somehow be circumvented, strategies to do so have to be evaluated. A first approach to such a strategy might be to pre-select one of the two possibilities arising from the problem and continue based on the assumption that the pre-selection is correct. If the next pad data packed appears to be correct, the process will simply continue. However, if the next pad data packet cannot be decoded correctly, a fall-back to the second possibility is done. Note that this still may result in data being interpreted as pad data packet erroneously, the likelihood of such an event will be much smaller, however.

A The DDR SDRAM Controller

This chapter of the appendix is taken from the documentation written for the DDR SDRAM controller introduced in chapter 3.

A.1 The Core

The core of the DDR SDRAM controller is implemented mainly in three files which are located in the `vhdl/core` directory. The user application will most likely instantiate the toplevel component defined in `sdr_controller.vhdl`. This file instantiates the components from the two other files, which contain most of the functional code.

The core needs two clock inputs, the main clock `CLK` driving most of the logic and `CLK_270`, a clock that is shifted by 270° with respect to the main clock, which is used to centre data in the data path. All signals the user application will use are synchronous to the main clock. This is also true for the reset signal `RST`.

The following control signals are defined as interface to the core: the frame base address `AIN`, the transfer direction `DIR`, the frame request signal `FRM`, the valid frame signal `GNT`, the data request signal `REQ`, the frame pause signal `BRK` and, for read transfers, the data valid signal `VLD`.

Data is fed to the core through `DIN` and read from `DOUT`. The rest of the signals declared in the entity are the output signals for the DDR SDRAM.

A.1.1 Data Frames

The data transfer model is frame based. Once the user application asserts `FRM`, it signals the request to transfer a single data frame. With the assertion of `FRM`, the transfer direction `DIR` and the frame base address `AIN` have to be valid.

The user application must hold the frame base address `AIN` until the interface signals a valid frame by asserting `GNT`. The transfer direction `DIR` must not change during a frame. Note that although not shown in the following examples, it is perfectly valid to start a frame paused by asserting `BRK` along with `FRM`.

To terminate a frame, the user application has to pause the transfer by asserting `BRK`, and deassert `FRM`. The controller will stop requesting data from the user application, for the DDR SDRAM, and deassert `GNT` some time later to signal successful termination of the frame. The user application must not request a second frame while `GNT` is still asserted.

A.1.2 Writing Data

To write data to the DDR SDRAM, take `DIR` high, drive a valid frame base address to `AIN` and assert `FRM`. As long as the user application is not requesting for a pause by asserting `BRK`, the core can request data at any time by asserting `REQ`. Valid data input to `DIN` must follow `REQ` the next clock cycle.

```
entity SDR_CONTROLLER is
 port (
   CLK, CLK_270, RST: in std_logic;
   -- user interface
   AIN: in std_logic_vector( ADDRESS_WIDTH-1 downto 0 );
   DIN: in std_logic_vector( DATA_WIDTH-1 downto 0 );
   DOUT: out std_logic_vector( DATA_WIDTH-1 downto 0 );
   DIR, FRM, BRK: in std_logic;
   GNT, REQ, VLD: out std_logic;
   -- DDR SDRAM interface
   CK, NCK, CKE, NCS, NRAS, NCAS, NWE: out std_logic;
   A: out std_logic_vector( ADDRESS_PORT_WIDTH-1 downto 0 );
   BA: out std_logic_vector( BANK_PORT_WIDTH-1 downto 0 );
   DQ: inout std_logic_vector( DATA_PORT_WIDTH-1 downto 0 );
   DQS: inout std_logic_vector( STROBE_PORT_WIDTH-1 downto 0 );
   DM: out std_logic_vector( MASK_PORT_WIDTH-1 downto 0 )
 );
end SDR_CONTROLLER;
```

Figure A.1: DDR SDRAM controller core interface

If the user application cannot provide data, it must request for a pause by asserting `BRK`. The core will react by deasserting `REQ` the next clock cycle. Once the user application is ready to provide data again, it has to deassert `BRK` and the controller will begin to request data one, or more, clock cycles later.

A.1.3 Reading Data

While writing data is straight forward, reading data is slightly more complicated.

To read data from the DDR SDRAM, pull `DIR` low, drive a valid frame base address to `AIN` and assert `FRM`. The core will signal a data request to the DDR SDRAM by asserting `REQ` and the arrival of valid data by asserting `VLD`. When the user application requests for a pause by asserting `BRK`, the controller will stop requesting data from the DDR SDRAM the next clock cycle. The user application will still have to handle the arrival of pending data, however.

So the user application has to keep track of the capacity to store or process data by counting requests to the DDR SDRAM, rather than actually received data. Of course, if data is only important up to some unknown point, the rest of the data can be safely ignored. Note that for read frames there may be pending data even after the frame has been terminated.

A.2 Configuring the Core

Configuration of the core is done in `sdr_configuration.vhdl` in the `vhdl/config` directory. Anything that cannot be configured here, involves changes in the VHDL code of the core. Please note that even some changes in this file involve changes to the VHDL code. For example, the core only supports a burst length of two, so always set the corresponding mode register bits accordingly. Also note that values defined relative to other values should never be changed. The parameters most likely to be changed when using the core with any other design than the H-RORC are listed below.

`DATA_PORT_WIDTH` the width of the `DQ` data port of the memory chip. The width of the controller data port will be twice this value and is defined as `DATA_WIDTH`.

`ADDRESS_PORT_WIDTH` the width of the `A` address port of the memory chip. The width of the controller address port is not derived from this value, but rather the following three and is defined as `ADDRESS_WIDTH`.

`COLUMN_ADDRESS_WIDTH` the number of bits of the address port of the memory chip used for the column address.

`ROW_ADDRESS_WIDTH` the number of bits of the address port of the memory chip used for the row address.

`BANK_PORT_WIDTH` the width of the `BA` bank address port of the memory chip.

`STROBE_PORT_WIDTH` the width of the `DQS` strobe port of the memory chip.

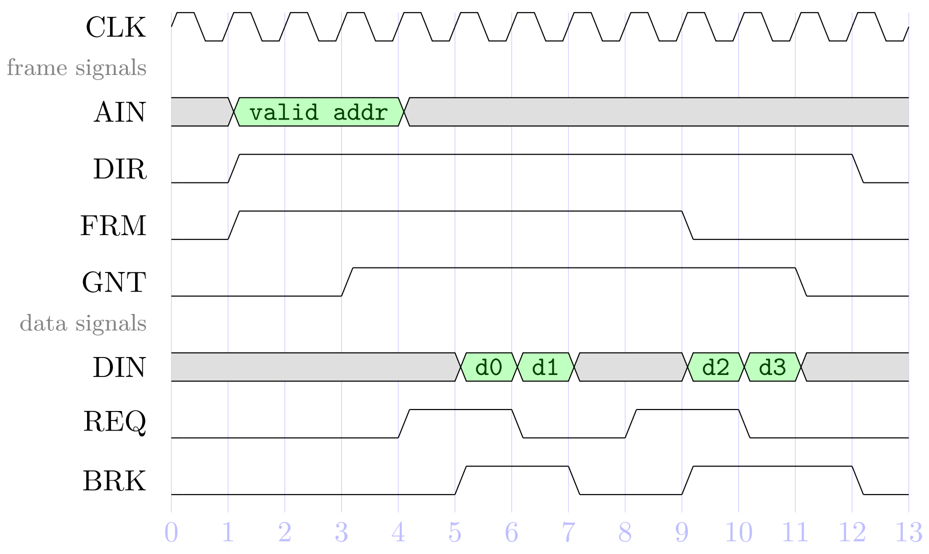


Figure A.2: Writing data using the core interface

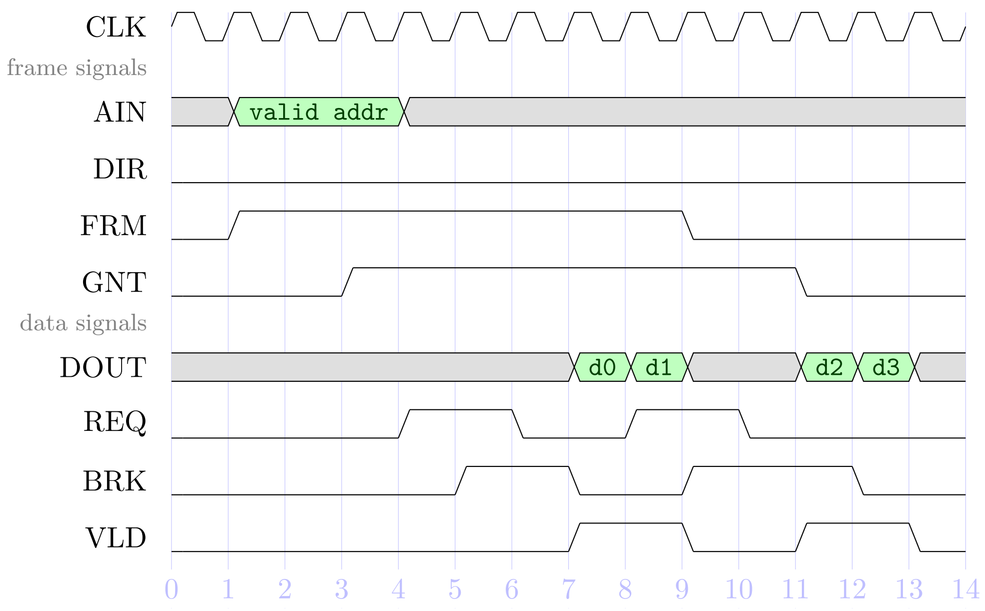


Figure A.3: Reading data using the core interface

MASK_PORT_WIDTH the width of the DQM data mask port of the memory chip.

T_* timing parameters according to the data sheet of the memory chip used, given in number of clock cycles. When in doubt, see [10]. T_DLL is the time for DLL reset, T_REF is the periodic refresh interval and T_INIT is the time to wait before starting with the initialisation sequence. The corresponding *_WIDTH values define the width of the respective counters.

CAS_LATENCY mode register bits for the CAS latency.

***_CNT** with these values the read timing can be adjusted.
The value for IDELAY_TAP_CNT defines the number of taps for signal delay. One tap is 74 ps for the Xilinx Virtex™-4. The other values are given in number of clock cycles. Half clock cycle CAS latency has to be taken into account with IDDR_LWRE_FLAG or adjusted with IDELAY_TAP_COUNT accordingly.

IDDR_LWRE_FLAG defines whether the low word is received to the rising edge (true) of the clock or the falling edge (false).

***_START, *_END** these values can be adjusted to support a different mapping of logical linear address to column, row and bank address.

A.3 Single Clock FIFO Interface

Most of the overhead associated with handling DDR SDRAM is absorbed by the core presented. The only vestige that cannot be hidden is that the user application can't just stream data to the RAM, but rather has to tell the core to get ready for data transfer. The core will then signal the user application when data should be provided or picked up. The user application, in turn, has to tell the core whether it is ready to provide or pick up data one clock cycle ahead.

While this can be alleviated by providing alternate interfaces that can be attached to the core, it cannot be completely eliminated. One such interface which is included with the controller is the single clock FIFO interface, which buffers the data. The single clock FIFO interface is implemented in the file `sdrif_sc_fifo.vhdl` in the `vhdl/if` subdirectory. It provides simple FIFO operation on the DDR SDRAM with a few additional control signals.

Read and write operations originate from the same clock domain, while the DDR SDRAM controller operates in its own clock domain. The term single clock applies only to the signals for the user application.

The interface defines the following control signals: the frame base address AIN, the frame length LEN, the transfer direction signal DIR, the frame request signal FRM and the valid frame signal GNT.

```
entity SDRIF_SC_FIFO is
 port (
   -- user interface
   RW_CLK, RW_RST: in std_logic;
   AIN, LEN: in std_logic_vector( ADDRESS_WIDTH-1 downto 0 );
   DIR, FRM: in std_logic;
   GNT: out std_logic;
   -- FIFO write interface
   DIN: in std_logic_vector( DATA_WIDTH-1 downto 0 );
   WR_EN: in std_logic;
   FULL, ALMOST_FULL: out std_logic;
   -- FIFO read interface
   DOUT: out std_logic_vector( DATA_WIDTH-1 downto 0 );
   RD_EN: in std_logic;
   EMPTY, ALMOST_EMPTY: out std_logic;
   -- DDR SDRAM interface
   SDR_CLK, SDR_CLK_270, SDR_RST: in std_logic;
   CK, NCK, CKE, NCS, NRAS, NCAS, NWE: out std_logic;
   A: out std_logic_vector( ADDRESS_PORT_WIDTH-1 downto 0 );
   BA: out std_logic_vector( BANK_PORT_WIDTH-1 downto 0 );
   DQ: inout std_logic_vector( DATA_PORT_WIDTH-1 downto 0 );
   DQS: inout std_logic_vector( STROBE_PORT_WIDTH-1 downto 0 );
   DM: out std_logic_vector( MASK_PORT_WIDTH-1 downto 0 )
 );
end SDRIF_SC_FIFO;
```

Figure A.4: Single clock FIFO interface

A.3.1 Data Frames

As with the core, the data transfer model is frame based. Once the user application asserts `FRM` it signals the request to transfer a single data frame. With the assertion of `FRM`, the transfer direction `DIR`, the frame base address `AIN` and, for read transfers, the frame length `LEN` have to be valid.

Again, the user application must keep the address signals `AIN` and `LEN` valid until the interface signals a valid frame by asserting `GNT`. The transfer direction `DIR` must not change during a frame.

How to terminate a frame is described for write and read transfers separately as it is handled slightly different in each case. However, after terminating a frame the user application of course has to wait until the interface signals the end of the frame by deasserting `GNT`, before requesting the next frame.

A.3.2 Writing Data

To write data to the DDR SDRAM, take `DIR` high, drive a valid frame base address to `AIN` and assert `FRM`. The user application can write data to the DDR SDRAM through the FIFO write interface without waiting for `GNT` to be asserted.

Here, `WR_EN`, `FULL` and, optionally, `ALMOST_FULL` control writing data to the FIFO. While `FULL` is deasserted, data can be written by asserting `WR_EN`. `ALMOST_FULL` is asserted one clock cycle ahead with respect to `FULL`. This makes control from a state machine easier.

To terminate the write frame, the user application simply deasserts `FRM` after writing the last word to be transferred to the FIFO and waits for `GNT` to be deasserted. Note that, although the user application does not have to wait for `GNT` to be asserted to write data to the FIFO, `AIN` must be held until that point, and that you must not deassert `FRM` earlier, either. Also note that, of course, the figure showing write access is only an outline of things that can happen. Usually, the FIFO won't run full when only two data words have been written.

A.3.3 Reading Data

To read data from the DDR SDRAM, pull `DIR` low, drive a valid frame base address to `AIN` and the desired frame length to `LEN` and assert `FRM`. The user application will receive data through the FIFO read interface.

As soon as `EMPTY` is deasserted, the user application can start reading data by asserting `RD_EN`. The data at `DOUT` will be valid the next clock cycle. Similar to the FIFO write interface, there exists a signal, that is asserted one clock cycle early with respect to `EMPTY`, namely `ALMOST_EMPTY`.

The read frame is terminated by the user application by deasserting `FRM` after reading the last word from the FIFO. The frame must not be terminated early, the user application has to read exactly the number of words requested.

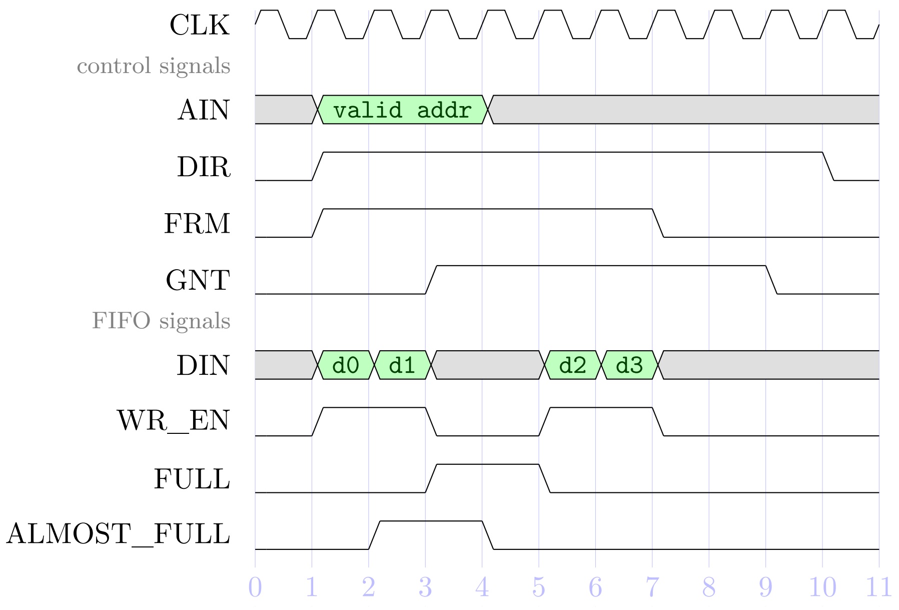


Figure A.5: Writing data using the SC_FIFO interface

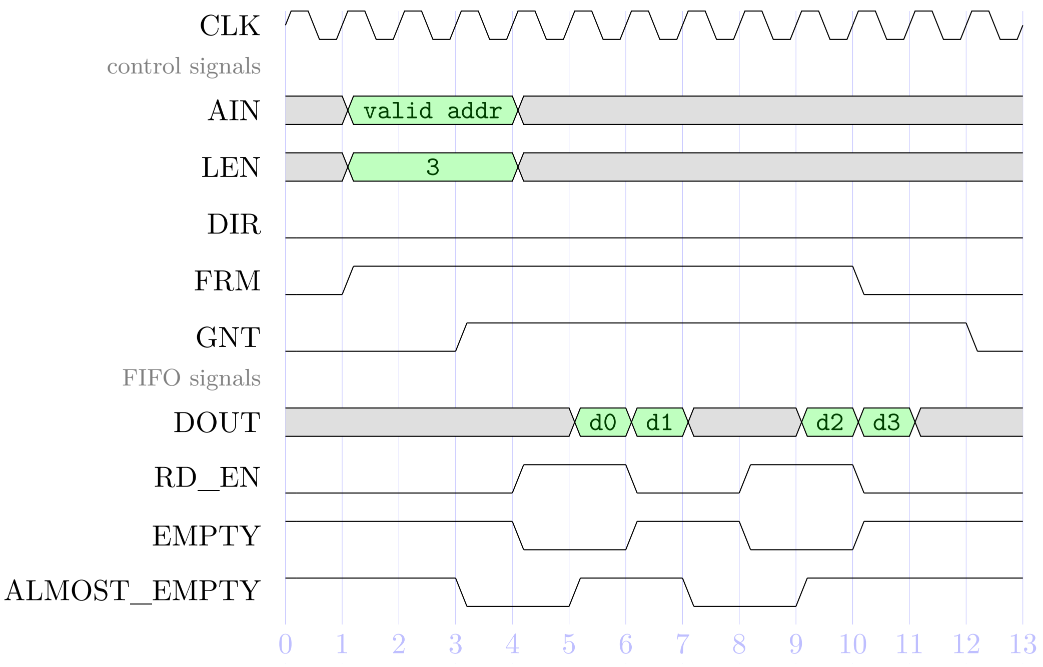


Figure A.6: Reading data using the SC_FIFO interface

B The PSI Tools

This chapter of the appendix is taken from the documentation written for the PSI tools introduced in chapter 4.

B.1 Prerequisites

Since the core component of PSI is designed as a kernel module, you might want to make sure you have a kernel with support for loadable modules, unless you want to hack the driver and patch your kernel. If you end up compiling a custom kernel, you might also want to enable support to forcibly remove a kernel module, just in case something goes wrong.

The PSI driver supports the bigphysarea patch, which is available separately. If you need large memory areas for DMA or want to share data between applications, you might want to compile your kernel with this patch applied. Please refer to the documentation that comes with the patch for information on how to install and use it.

B.2 Building and Installing

Once you have obtained a copy of the PSI sources, you just have to `cd` into the PSI top-level directory, which I will refer to as the *working directory*, and call `make` to build everything. The build process creates several additional subdirectories to hold the driver, libraries and tools, so after a successful build, the working directory should look something like this:

```
> ls
bin  doc  include  lib  Makefile  Prefix.make  README  Rules.make  src
```

If you want to install the PSI driver, library and tools, you can call `make install`. This will place the driver into the modules subdirectory of the current kernel and register it with `modules.dep`, so you can `modprobe` it. The tools, header files and libraries will be placed in `/usr/local/bin`, `/usr/local/include` and `/usr/local/lib`, respectively. Note that you will need administrator privileges if you want to install to the default location.

If you want to install the PSI library and tools somewhere else than the default location, you can set `PREFIX` for `make install`. For example, if you want to install to `/opt/local/psi`, just call `make PREFIX=/opt/local/psi install`. Note, however, that the driver will still be installed inside the modules subdirectory of the current kernel.

Of course, you don't have to install everything. You can just use the library and tools inside the working directory. You will have to tell your compiler where to find header files and libraries for PSI, though. Note that all platform dependent files are placed in platform subdirectories. If, for example, you are working on a i686 Linux system with kernel 2.6.12-plain, `make` will create the platform subdirectories `bin/Linux-i686` and `lib/Linux-i686` for the tools and libraries, and `modules/Linux-i686/2.6.12-plain` for the driver. (You can use `uname` to get the identifiers for your system.) You can still install the driver into the modules subdirectory of the current kernel by calling `make install-module`.

B.3 Region Handling

The PSI library works mostly on *regions*. Regions are simply handles to blocks of memory. But while the term memory usually refers to the RAM installed in your computer, regions can also point to a block of memory or I/O ports of a PCI device. This means, there are different types of regions.

Most functions of the PSI library return a result of type `PSI_Status`. Be sure to check for errors and act appropriately after each and every call because things may eventually fail.

B.3.1 Opening a Region

First, you have to open a region. To open a region, you need a region path which identifies the region. Depending on what type of region you want to open, the region path varies. A region is opened with a call to `PSI_openRegion()` which is passed a pointer to a region handle and the device path as parameters. Region handles are of type `tRegion`.

Named memory regions are identified by their name. You can choose the name to be whatever comes to your mind, as long as the region path doesn't exceed 100 characters, the terminating zero included. The region path is `/dev/psi/mem/<name>`. If you want to specifically get memory from the bigphysarea, the region path should be `/dev/psi/bigphys/<name>`.

Physical memory regions are identified by their physical address. You have to take care yourself, that the physical address exists and cannot be used by the kernel or user space processes other than through PSI. Usually you will reserve a block of the memory installed in your computer by passing the kernel a `mem` argument. See the kernel documentation for details. The region path is `/dev/psi/physmem/<address>`.

Config space regions point to the configuration space of a specific PCI device. They can be identified by either the PCI address (bus, slot and function number) or by the device id (vendor, device and index, where index is used to identify a specific device if more than one devices use the same vendor and device id).
Use either `/dev/psi/bus/<bus>/slot/<slot>/function/<function>/config` or `/dev/psi/vendor/<vendor>/device/<device>/<index>/config` as region path, depending on whether you use the PCI address or the device id to identify the device.

Base address register regions point to either memory or I/O ports on a PCI device. Like config space regions, they can be identified by either the PCI address or the device id. Additionally you have to provide the number of the base address register which points to the memory or I/O ports on the device.
Use either `/dev/psi/bus/<bus>/slot/<slot>/function/<function>/base<bar>` or `/dev/psi/vendor/<vendor>/device/<device>/<index>/base<bar>` as region path, depending on whether you use the PCI address or the device id to identify the device.

B.3.2 Sizing a Region

After successfully opening a region, you have a valid region handle. While config space and base address register regions have fixed sizes, named and physical memory regions don't. You have to size a memory region before you can actually use it. Once the size of a region has been set, it is fixed and can't be changed throughout the lifetime of the region. So another point of view would be, that regions for PCI devices are sized during opening, while memory regions have to be sized manually. You size a memory region by calling `PSI_sizeRegion()` which is passed the region handle and a pointer to an unsigned long which holds the requested size before the call and the granted size after the call. Note that the granted size may differ from the requested size since memory is allocated with a granularity of the page size defined by the system.

Note that regions can be opened by different processes. Of course only the first process that opens a region has to size it. So if a call to `PSI_sizeRegion()` fails with `PSI_SIZE_SET`, it has already been sized by another process and you don't have to size it. Beware that you are sharing the region with another process, though, and you might have to take measures to synchronised shared access to the region. See the documentation that comes with your development tools for details on inter application communication.

B.3.3 Closing a Region

When you are finished using a region, you should always close it. The region will be disposed of after all processes that have used it called `PSI_closeRegion()`. This function is passed a region handle as parameter.

B.3.4 Preventing Disposal of Regions: Locking

Sometimes you might want to make sure that a region is not disposed of even after the last process that has used it closed it. To lock a region, you call `PSI_lockRegion()`. To unlock it, call `PSI_unlockRegion()`. You can also check the number of locks set on a region with `PSI_checkRegionLock()`. All of these functions are passed a region handle as parameter. While the first and second function return a result, `PSI_checkRegionLock()` returns the number of locks set on the region or a negative value if the region handle is invalid.

B.4 Region Access

Now that we have discussed the basics of handling regions, you may want to do something useful with it. For most region types, you basically have two choices to access the contents of the region.

B.4.1 Read and Write

All types of regions support read and write access through `PSI_read()` and `PSI_write()`, respectively. Both functions are passed a region handle, the byte offset to the start of the memory the region is pointing to, the size of a single data element in bytes, the size of the block in bytes and finally a pointer to the block of data as parameters. The size of a single data element can be `_byte_`, `_word_` or `_dword_`, the block size and offset must be aligned accordingly.

B.4.2 Mapping Regions

Named, physical memory and memory base address register regions support mapping of the region into user space. After successfully mapping the region, you can access the memory the region is pointing to through a pointer. Beware that there are side effects on how you access the memory, though. If you access memory base address register regions byte-wise, the PCI bus will translate this to a 8-bit memory cycle and so on. This means you have to carefully choose how to type the pointer you are using. It is highly recommended not to use standard C types as these are not guaranteed to have any fixed size. Rather you should include `sys/types.h` in your program and use `int8_t`, `u_int8_t` and the likes.

You can map a region by calling `PSI_mapRegion()`. This function is passed a region handle and a pointer to a pointer variable which will hold the virtual address of the mapping after the call. To unmap the region again, call `PSI_unmapRegion()`, which is passed the region handle and the pointer value returned by the call to map the region.

You can also map only a part of the region by calling `PSI_mapWindow()`. You pass this function a region handle, the byte offset to the start of the memory the region is pointing to, the size of the window you want to map and, again, a pointer to a pointer variable. To unmap the window, you call `PSI_unmapWindow()`, which is passed the region handle, the pointer value and the size of the window.

Config space and I/O base address register regions do not support mapping.

B.5 Advanced Hardware Access

While the previous section introduced the basic means by which hardware can be accessed, this section delves into some of the more advanced topics, such as interrupts, direct memory access and timing issues.

B.5.1 Interrupts

When communicating with a device the need for synchronisation soon becomes evident. You can of course implement a register in your device design that your program checks regularly, but the use of active waiting means unnecessary load on both the processor and the PCI bus. This can be solved by implementing interrupts in your device design.

With the advent of PSI2, the driver supports basic use of interrupts from user space. There are, however, some things you have to consider when implementing interrupts in your device design and writing the program that will make use of the interrupts fired by your device.

First of all, it is convenient to implement a register in your device design that will control how the device drives the PCI interrupt pins. There should at least be one flag to completely prevent interrupts and one flag to reset the interrupt once fired. Your device should by default not fire any interrupts unless the flag to allow interrupts is explicitly set, since unhandled interrupts may confuse the system.

That said, the procedure for using interrupts for synchronisation is as follows: open a config region, tell your device it is allowed to fire interrupts, wait for an interrupt to occur, reset the interrupt and tell your device not to fire any more interrupts, then flush all pending interrupts. If you need to wait for interrupts in a loop, start over with telling your device it is allowed to fire interrupts again, wait for the interrupt to occur, and so on.

As you can see from the above discussion it is indeed convenient to implement both flags within a single register, as you won't need to write more than one word through the PCI bus.

To wait for an interrupt, simply call `PSI_waitInterrupt()`. This function must be passed a handle to a config space region for the device that fires the interrupt your program wants to listen to. Your program will be put to sleep until the interrupt occurs. When your program receives a signal or the config space region is forcibly closed, waiting will be aborted and an error will be returned as result. So again, be sure to check the result before assuming you got an interrupt.

To flush all pending interrupts, you call `PSI_flushInterrupts()` and pass it the handle to the same config space region you use for waiting. This is necessary because the interrupt handling is divided into two parts, one part in the kernel driver and one part in your user program. While the kernel driver tells the kernel to wake up the waiting process and exits, some time will pass until your program actually wakes up and is rescheduled—and resets the interrupt. Since the interrupt handler of the kernel driver exited but the interrupt pin is still driven by your device, the kernel thinks it sees another interrupt. So there will be spurious pending interrupts before your program regains control.

B.5.2 Direct Memory Access

If you need fast data transfer between the memory in your computer and your device without imposing a heavy load on the processor, you may choose to implement direct memory access, or short *DMA* in your device design.

While PSI supported DMA in the first generation, things have become somewhat more complex over time. While on most x86 machines there was no difference between the physical address of a block of memory in your computer and the address of the same block as seen from the PCI bus, modern systems sport a dedicated memory management unit for bus access and you can't just simply tell your device the physical address of some memory block you have allocated. You have to get a valid bus address that is assigned for a link between a specific device and a specific block of memory.

To further complicate the matter, the block of memory, or *DMA buffer*, can only be used by either the device or your program, not by both at a time. So basically to transfer data from your device to memory in your computer, you open both a memory region that will be used as a DMA buffer and a config space or base address register region for the device, map the DMA buffer to the device, tell your device to start the transfer, wait for the transfer to finish, unmap the buffer from the device and remap it into user space. The transfer in the other direction works accordingly, first mapping the buffer into user space, preparing the data to be transferred, unmap the buffer from user space and map it to the device, tell your device to start the transfer, wait for the transfer to finish and unmap the buffer from the device.

It is important that the mapping of the DMA buffer to the device is held only as long as needed, as DMA mappings may be limited on some systems. So while there exists a solution to leave a buffer mapped but suspend DMA to allow access by your program, you should consider twice before implementing a series of transfers that way. If you suspend DMA, you have to explicitly resume before the device accesses the buffer, again.

To map a buffer to a device, call `PSI_mapDMA()` and pass it the handles to the memory and the device regions, the direction of the transfer and a pointer to an unsigned long which will hold the bus address you can pass along to your device if the call succeeds. The direction can be `_bidirectional_`, `_fromDevice_` or `_toDevice_`. While it may be tempting to always use `_bidirectional_`, this is not recommended as there may be more overhead on some systems. You can unmap the buffer from the device by calling `PSI_unmapDMA()`, passing it the handle to the memory region that is used as DMA buffer.

Suspending DMA for a buffer that is mapped to a device to allow access by your program is done by calling `PSI_suspendDMA()`. To resume DMA again, you have to call `PSI_resumeDMA()`. Both functions expect a handle to the memory region that is used as DMA buffer.

B.5.3 Timing Issues

When dealing with hardware access you will often encounter timing specifications that have to be met. While waiting for a few microseconds without imposing serious load on the processor can be achieved through `usleep()`, this function can't guarantee you sub-millisecond granularity or even microsecond granularity as the name suggests. The granularity of any function that sleeps is given by the system tick, which is in the range of a few milliseconds. Note that in older kernels there was the possibility to get realtime privileges and thus a much finer granularity through `sched_setscheduler()` and `nanosleep()`, but this functionality seems to be dysfunctional as of kernel 2.6.

This leaves you with little other choice than busy waiting if you need a better resolution than a few milliseconds. Since this will probably be done by many a program that uses the PSI library, you can simply call `PSI_delay()` and pass it the number of nanoseconds to wait. There is some overhead, so don't expect it to give you nanosecond granularity, but it will most likely deliver sub-microsecond granularity on modern systems.

Beware, however, that there's still the possibility that the scheduler chooses to suspend your program at any time and select another process for execution, so every now and then your program will fail if you need to guarantee timing to be within a certain margin, so be prepared for this.

B.6 Miscellanea

There are some functions exported by the PSI library which did not fit in the above categories. Some of these are supported by the PSI tools and can be issued from the command line.

You can get a textual representation of mostly any error that can occur during the use of the PSI library by calling `PSI_strerror()` and passing it the result of the operation which failed.

By calling `PSI_getRegionStatus()`, you can get information about a region. Pass it a region handle and a pointer to a tRegionStatus structure. Note that only those fields have a defined value which are valid for the type of region you passed a handle on. For example, the `bus` and `devfn` fields will be undefined for memory regions and a config space region has neither a `name` nor `address`.

You can save and restore the config space of a PCI device by calling `PSI_saveState()` and `PSI_restoreState()`, respectively. Both functions expect, of course, a handle to a config space region and a pointer to a buffer of at least 256 bytes to hold the config space data. If you pass `PSI_restoreState()` zero as pointer to the buffer, the base address registers and IRQ line register will be refreshed from the internal data the kernel stores about a device and the device is re-enabled.

This is what comes closest to *hot-plugging* a PCI device. As long as the interface of your device to the PCI bus doesn't change, you can re-program your device design on a running system and re-enable the device without rebooting. Note that this is not real hot-plugging, as this is only an addition to the PCI specification and requires support from the system BIOS. There is no easy way to re-enable your device if you dropped, changed or added any base address registers and there will probably never be.

The PSI driver does a lot of house-keeping and cleans up after processes that leave anything open. However, if a region is locked, for example, PSI can't guess when to release it, so the region is kept. Even worse, if a process hangs and can't be killed, PSI can't clean up. You can call `PSI_cleanup()` to force clean up. The function expects a process id to clean up after, but it will close all regions that are currently open if you pass zero as the process id.

The second generation of PSI still supports the call `PSI_getPhysAddress()`, which was used to provide DMA support. The use of this function is strongly discouraged. Please use the new DMA layer discussed previously.

B.7 Reference

To use the PSI library, make sure the driver is installed properly, include `psi.h` and `psi_error.h` in your program and link with `-lpsi`.

B.7.1 Region Handling

```
/* open a region */
PSI_Status PSI_openRegion( tRegion * pRegion, const char * regionPath );

/* size a memory region */
PSI_Status PSI_sizeRegion( tRegion region, unsigned long * pSize );

/* close a region */
PSI_Status PSI_closeRegion( tRegion region );

/* region locking */
PSI_Status PSI_lockRegion( tRegion region );
PSI_Status PSI_unlockRegion( tRegion region );
int PSI_checkRegionLock( tRegion region );
```

B.7.2 Region Access

```
/* data sizes */
#define _byte_  1
#define _word_  2
#define _dword_ 4

/* reading/writing from/to a region */
PSI_Status PSI_read( tRegion, unsigned long offset, int dataSize,
                     unsigned long bufferSize, void * buffer );
PSI_Status PSI_write( tRegion, unsigned long offset, int dataSize,
                      unsigned long bufferSize, void * buffer );
```

```
/* mapping regions to user space */
PSI_Status PSI_mapRegion( tRegion region, void ** pPtr );
PSI_Status PSI_unmapRegion( tRegion region, void * ptr );
PSI_Status PSI_mapWindow( tRegion region, unsigned long offset,
                          unsigned long size, void ** pPtr );
PSI_Status PSI_unmapWindow( tRegion region , void * ptr,
                            unsigned long size );
```

B.7.3 Advanced Hardware Access

```
/* dma mappings */
PSI_Status PSI_mapDMA( tRegion buffer, tRegion device, int direction
                       unsigned long * pAddress );
PSI_Status PSI_unmapDMA( tRegion buffer );
PSI_Status PSI_suspendDMA( tRegion buffer );
PSI_Status PSI_resumeDMA( tRegion buffer );

/* interrupts */
PSI_Status PSI_waitInterrupt( tRegion device );
PSI_Status PSI_flushInterrupts( tRegion device );

/* delay execution for given number of nanoseconds */
PSI_Status PSI_delay( unsigned long nsecs );
```

B.7.4 Miscellanea

```
/* textual representation of PSI_Status results */
char *PSI_strerror( PSI_Status );

/* get status information about the region */
PSI_Status PSI_getRegionStatus( tRegion region,
                                tRegionStatus * pStatus );

/* save/restore state of PCI device */
PSI_Status PSI_saveState( tRegion region, void * buffer );
PSI_Status PSI_restoreState( tRegion region, void * buffer );

/* cleanup after process */
PSI_Status PSI_cleanup( pid_t pid );
```

References

[1] ALICE Collaboration. Technical Design Report of the Trigger, Data Aquisition, High Level Trigger and Control System. CERN, January 2004

[2] T. Alt. HLT RORC Production Readiness Review. Kirchhoff-Institute for Physics, October 2005

[3] T. Alt. PhD Thesis, University of Heidelberg, not yet published

[4] D. Atanasov. Design and Implementation of a System for Data Traffic Management in a Real-Time Processing Farm Operated at 1 MHz. PhD Thesis, University of Heidelberg, not yet published

[5] M. Balch. Complete Digital Design, A Comprehensive Guide to Digital Electronics and Computer System Architecture. McGraw-Hill, June 2003

[6] P. J. Bryant. A Brief History and Review of Accelerators. CERN, 1994

[7] CERN Library Archive. Chronology of the CERN. `http://library.cern.ch/archives/chrono`

[8] CERN Public Relations. `http://public.web.cern.ch`

[9] J. Corbet, A. Rubini, G. Kroah-Hartman. Linux Device Drivers, 3rd Edition. O'Reilly, January 2005

[10] Data Sheet for the EDD2516AMTA-6B-E 256M bits DDR SDRAM. Elpida Memory, Inc., 2005

[11] Linux Kernel Cross-Reference. `http://lxr.linux.no`

[12] Porting device drivers to the 2.6 kernel. LWN Articles Archive. `http://lwn.net/Articles/driver-porting`

[13] Memory 97. Integrated Circuit Engineering Corp., 1997

[14] C. Lara. PhD Thesis, University of Heidelberg, not yet published

[15] D. Meschede. Gerthsen Physik, 21. Auflage. Springer, 2002

[16] PCI Local Bus Specification, Revision 2.2. PCI Special Interrest Group, December 1998

[17] J. Peschek. Infrastructural Improvements on a FPGA-based Pre-processing System. Diploma Thesis, University of Heidelberg, December 2006

[18] T. M. Steinbeck. A Modular and Fault-Tolerant Data Transport Framework. PhD Thesis, University of Heidelberg, 2004

[19] Virtex-4 Data Sheet: DC and Switching Characteristics (DS302). Xilinx, Inc., September 2006

[20] Virtex-4 User Guide (UG070). Xilinx, Inc., March 2006

[21] Interfacing to DDR SDRAM with CoolRunner-II CPLDs (Application Note XAPP384). Xilinx, Inc., February 2003

[22] DDR SDRAM Controller using Virtex-4 FPGA Devices (Application Note XAPP709). Xilinx, Inc., October 2006

[23] W.-M. Yao *et al.* Nuclear and Particle Physics. Journal of Physics G, Volume 33, July 2006

Acknowledgements

I would like to thank everyone who supported me during the time of my studies and especially during the work for this thesis, by sharing their knowledge, their time or their lunch with me.

First, I would like to thank Prof. Dr. Volker Lindenstruth for providing guidance and valuable advice whenever needed. Much the same goes for Torsten Alt.

Many kudos go to Felix Rettig and Jochen Thäder for proofreading the thesis, and to Jan de Cuveland for all the endless discussions about LaTeX and all sorts of other things. Notwithstanding the tight schedules and lots of work I had a great time at the KIP, so far, and I would be looking forward to spending some more time with all of you.

I would also like to thank my family for their support. It hasn't been exactly the easiest time—for all of us. But I am absolutely positive about one thing: things change.

Special thanks go to Jens Tschritter. I owe him more than I can probably ever give back.

earth (or **Earth**; often preceded by 'the') Mostly harmless.

The Hitchhikers Guide to the Galaxy

Erklärung

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

Heidelberg, den 1. März 2007 Florian Painke