FACULTY OF
PHYSICS AND ASTRONOMY

UNIVERSITY OF HEIDELBERG

DIPLOMA THESIS
IN PHYSICS

SUBMITTED BY
THOMAS GERLACH
BORN IN
HEIDELBERG, GERMANY

SEPTEMBER 2008

Development of an Embedded Linux System for the Global Tracking Unit of the ALICE TRD at the LHC (CERN)

This diploma thesis has been carried out by **Thomas Gerlach** at the

Kirchhoff Institute for Physics

under the supervision of

Prof. Dr. Volker Lindenstruth

Entwicklung eines Embedded Linux Systems für die GTU des ALICE TRD

Der Übergangsstrahlungsdetektor (*TRD*) des *ALICE* am *CERN* ist als Triggerdetektor konzipiert. Er trägt zur Auswahl der für die jeweilige Fragestellung relevanten Ereignisse aus der großen Anzahl der Teilchenkollisionen bei. Die *Global Tracking Unit* (*GTU*) ist für den L1-Triggerbeitrag sowie die Pufferung und Übertragung der Rohdaten an das *Data Acquisition System* und den *High-Level Trigger* zuständig. Als hierachisches System aufgebaut besteht sie aus 109 Modulen.
In dieser Arbeit wird die Entwicklung eines Embedded Linux Systems und der dafür notwendigen Hardware-Komponenten vorgestellt, welches als Plattform für Software zur Steuerung und Überwachung der GTU dient. Das entwickelte FPGA-Design ermöglicht über Schnittstellenkomponenten den Zugriff auf die verschiedenen Untersysteme der GTU. Zur Nutzung der SD-Speicherkarten wurde ein SD Memory Card Controller entwickelt. Die Initialisierung der SD-Karten wird über Software-Routinen realisiert, während leistungsrelevante Lese- und Schreiboperationen in die Controller-Hardware integriert sind. Bei der Entwicklung des Linux-Kernels stellen die spezifischen Hardware-Komponenten der GTU eine besondere Herausforderung dar. Der Kernel-Konfigurationsprozess ist so konzipiert, dass relevante Parameter des FPGA-Designs automatisch eingebunden werden, ohne eine manuelle Anpassung der Kernel-Quellen zu erfordern. Ein für das Laden des Kernels zuständiger Bootloader implementiert grundlegende Zugriffsroutinen auf das Dateisystem der SD-Karten.

Development of an Embedded Linux System for the GTU of ALICE TRD

The *Transition Radiation Detector* (*TRD*) of *ALICE* at *CERN* is designed as trigger detector. It serves to select physical events of interest to the main research questions among the huge number of particle collisions. The *Global Tracking Unit* (*GTU*) is responsible for contributing to its L1 trigger decision. It also buffers and forwards the event raw data to the *Data Acquisition System* and the *High-Level Trigger*. Designed as a hierachical system, the GTU consists of 109 system boards.
This thesis describes the development of an Embedded Linux System and the necessary hardware components, which is used as platform for software to control and monitor the GTU. The FPGA design developed also provides access to the different system components of the GTU. An SD Memory Card Controller was designed to access the SD Memory Cards. Initialization of the cards is done by software, while performance-relevant read and write operations are integrated into the controller hardware. The custom-built hardware components of the GTU represent a particular aspect to the development of the Linux kernel. An auto-configuration mechanism allows to extract and integrate relevant parameters of the FPGA design into the kernel configuration without the need of adapting kernel sources manually. The boot loader responsible for loading the kernel implements routines to access the file system of the cards.

Contents

List of Figures

List of Tables

1 Introduction

The investigation of nature and its innermost workings have been fascinating physicists and philosophers from the beginnings of science. Since J. Dalton renewed Democritus' hypothesis on a particle model of matter, many new discoveries have been made on this subject.
In the 1920s, the high-energy physics era began, and scientists were able to have deeper insight into the structure of matter. As a result of these experiments, new particles have been discovered, which helped to verify or disprove some theories, but also revealed new questions. A theory, which sufficiently describes most of the observed phenomena, is the Standard Model of particle physics. Based on gauge theories of the electroweak and strong interaction, it unifies three of the four known fundamental interactions between the elementary particles and has been approved and refined by those high-energy physics experiments.

Physics Overview According to the Standard Model all known matter is constituted from two types of elementary particles, *leptons* and *quarks*. Interaction among those particles can be ascribed to three fundamental forces: the electromagnetic, the strong and the weak interaction, which are carried by corresponding *gauge bosons*. Both the theory of electroweak interaction and the theory of *quantum chromodynamics* (*QCD*) are considered by the Standard Model.
Table 1.1 lists all twelve known fundamental particles, their electric charges, and their masses. Although there exists an anti-particle to each particle listed below, they have not been included in the table as they have corresponding properties.
In table 1.2 the four fundamental forces are listed along with their characteristic range, relative strength, and the corresponding gauge bosons. The fourth fundamental force, *gravitation*, is not yet integrated in the Standard Model. In fact, it still has to be formulated as a quantized theory. The weak and the electromagnetic force, however, couple to the weak and the electromagnetic charge, respectively. The strong interaction is explained by introducing a color charge, which states the name *quantum chromodynamics*. This charge can be one out of red, blue, green or the corresponding anti-colors, respectively.

In contrast to the electromagnetic and the weak force, the strength of the interaction between quarks increases with growing distance. As a result, solitary quarks cannot be observed. This phenomenon is known as *confinement*. Considering the color charges, confinement can also be expressed as a free particle having to be color-neutral, resulting

Generation	Leptons	q/e	m	Quarks	q/e	m
First	e	-1	511 keV	u	2/3	$\approx 3.3\,\mathrm{MeV/c^2}$
	ν_e	0	$\leq 225\,\mathrm{eV}$	d	−1/3	$\approx 6\,\mathrm{MeV/c^2}$
Second	μ	-1	106 MeV	s	−1/3	$\approx 104\,\mathrm{MeV/c^2}$
	ν_μ	0	$\leq 0.17\,\mathrm{MeV}$	c	2/3	$\approx 1.27\,\mathrm{GeV/c^2}$
Third	τ	-1	1.78 GeV	b	−1/3	$\approx 4.2\,\mathrm{GeV/c^2}$
	ν_τ	0	$\leq 18.2\,\mathrm{MeV}$	t	2/3	$\approx 171.2\,\mathrm{GeV/c^2}$

Table 1.1: All known matter consists of twelve elementary particles. Each particle has an anti-particle with corresponding properties, which are not shown in the table. The electric charge is given as a multiple of the elementary charge, while the mass is given in energy equivalent. Source: [A+08]

Force	Range [m]	Strength (rel.)	Couples to	Gauge Boson
Strong	10^{-15}	1	Color charge	Gluon (g)
Weak	10^{-17}	10^{-2}	Weak charge	$W^{\pm}$, Z^0
Electromagnetic	∞	10^{-14}	EM charge	Photon (γ)
Gravitation	∞	10^{-38}	Mass	Undetected

Table 1.2: Interaction between the elementary particles can be described with the four fundamental forces listed above. These forces have effects on matter at very different scales. While electromagnetism and gravitation have effects on atomic and macroscopic scales, the weak and strong interaction are observable on subatomic scales. Sources: [Stö00, A+08]

in two kinds of composite quark particles, called *hadrons*.
When combining a quark and its anti-quark, a new color-neutral particle forms, since color and its anti-color annihilates coloration. The resulting particle family is called *mesons*. In analogy to the classical theory of colors, there is another way to achieve color neutrality by additively mixing the three colors red, green and blue, or anti-red, anti-green and anti-blue, respectively. Thus, the combination of three pairwise differently (anti-)colored quarks also results in colorless particles, referred to as *baryons*.

The Standard Model sufficiently describes the variety of hadrons and baryons and most of the other phenomena seen in particle physics so far. But there are still some questions left open, such as why there are exactly three generations of fundamental particles or where their physical properties actually come from. These questions are some of the reasons for building the *Large Hardon Collider* (*LHC*) at *CERN*[1].

Specifically, physicists at *ALICE* (*A Large Ion Collider Experiment*) hope to gain more insight into the physics of the universe at a very early stage for the purpose of learning more about how the universe developed. QCD predicts a cancellation of confinement at either high density of hadron matter or high temperatures, when a phase transition from

[1]European Organization for Nuclear Research, situated in Geneva, Switzerland.

the hadronic into a partonic phase occurs, and quarks can move quasi-freely. This partonic sea is referred to as *quark-gluon plasma* (*QGP*). Ultra-relativistic heavy-ion collisions as provided by the *LHC* for *ALICE* are expected to produce very hot QGP via hard initial parton scattering. With extremely high particle densities (several thousand per unit of rapidity for each collision), the demands regarding the resolution of the detector are very high. Furthermore, a massive amount of data is produced for each single event, which can be neither completely analyzed in real-time nor stored. Thus, a dedicated hierarchical 4-stage trigger system preselects physics events of interest for permanent storage and offline analysis.

The *Transition Radiation Detector* (*TRD*) is one of the sub-detectors of *ALICE*. Its main purpose is to trigger on electrons with high transversal momenta. The *Global Tracking Unit* (*GTU*) is part of the trigger chain of *ALICE* and provides the contribution by the TRD to the trigger decisions, but is also responsible for reading out and buffering the event raw data. It consists of several sub-entities, each capable to run control and monitoring software. For reliability and flexibility, it is advisable to provide the *GTU* with an Embedded Linux System. The development of such a framework is subject of this thesis.

Embedded Linux Systems An *embedded system* is a computer system consisting of both hardware and software, which is designed to perform one or a few dedicated tasks. The manifold applications of embedded systems vary from control systems for industrial plants to common home appliances such as washing machines, cell phones or entertainment devices. While in the beginning embedded systems were provided with very application-specific system software (*home-grown system software*), embedded derivatives of modern operating systems nowadays offer support for a large number of embedded platforms and thus replace home-grown software solutions.
Because of its flexibility and portability, *Linux* represents a suitable operating system for embedded purposes. It is written in *C* for the most part, and its open source code allows to adapt the kernel to specific hardware as well as to extend it for new functionality or remove features, which are not relevant for the actual purpose. The portability arises from the support for various processor architectures such as *ARM*, *x86*, *PowerPC* and others provided by both the kernel and the development toolchain used to compile the kernel sources.

In contrast to Linux distributions for desktop systems and servers, *Embedded Linux* is especially designed for systems with limited resources. Thus, the standard C library *glibc* is usually replaced by less memory-consuming yet more limited alternatives. The same applies to system software and user applications. As UNIX-like system, Linux depends on a *root file system*, which provides the directory structure required by the kernel and contains system and user applications. On desktop systems, the root file system is usually located on the hard disk drive. Embedded systems, however, typically do not have such storage devices, but use their main memory to host the root file system instead. The

specific memory region, which is reserved for this purpose, is referred to as *ramdisk* and can be accessed and formatted with any file system just as normal storage media. The root file system, which is appended to the image of the embedded kernel, is copied to the ramdisk during the boot stage. However, any modifications applied to the ramdisk are lost in case of a reboot or power-cycle.

Building an Embedded Linux System implies *cross-compilation* of both kernel and application sources. A cross-compiler, which runs on a platform of a processor architecture different to that of the target system, processes the source code and generates executable code for the target processor architecture. This is necessary, because most embedded systems are not designed to operate as development platforms. Moreover, the embedded kernel usually lacks device drivers to access the boot medium, which is often based on *PROM*s or *flash memory chips*.

An Embedded Linux System typically consists of the following elements:

- The Linux kernel binary image, including the root file system
- A boot loader
- Custom-developed drivers for special hardware
- One or more application processes to provide the functionality required

The kernel image incorporates both the actual Linux kernel and the root file system, which contains the directory nodes as well as system and user applications. The boot loader is responsible for copying the kernel image from the boot medium into main memory and jumps to the kernel entry point. While desktop systems are equipped with a *BIOS*, which offers basic I/O capabilities to access the boot medium, the boot loader of an embedded system must implement those routines by itself due to the lack of a BIOS. Because the Linux kernel does not support special hardware components an embedded system is equipped with, it is necessary to provide custom-developed device drivers to access these components. Those drivers are implemented as *kernel modules* located in a specific directory of the root file system and can be added to or removed from the kernel dynamically during runtime. Finally, one or more applications are required to perform the actual task the system is intended for.

Thesis Overview This thesis focuses on the development of an Embedded Linux System and the necessary hardware components for the Global Tracking Unit of the *ALICE TRD* to serve as a platform for future control and monitoring software.
The following chapter gives an overview of the particle accelerator *LHC* and its experiments, in particular *ALICE*, the *TRD* and the *GTU*. Chapter 3 illustrates technical aspects of the *GTU*. The Embedded PowerPC System, which serves as the basic hardware platform to communicate and interface with the *GTU* and its components, is described in chapter 4. In chapter 5 the development process of the Embedded Linux is discussed.

An essential part of the development was to adjust the kernel to the application-specific hardware. The generic SD Memory Card Controller, also developed in this thesis is described in chapter 6. It allows to access the SD Memory Cards the *GTU* is provided with. The controller is required to enable the use of cards as boot media for the Linux and to host the root file system. Chapter 7 discusses the development of a boot loader responsible for copying the Linux kernel image from the SD Memory Cards into main memory and its execution. It is implements the necessary basic I/O routines to access the card file system. Finally, the last chapter summarizes the results of the thesis.

2 The Experiment

The structure of matter and the fundamental interactions is investigated at the *European Organization for Nuclear Research* (*CERN*)[1]. It provides the *Large Hadron Collider*, a huge particle accelerator to collide particle beams, and the necessary computing resources responsible for analyzing the data produced by the detectors of the experiments, which study these collisions.
At the time of writing this thesis, the accelerator is in the final stage of completion, and *first beam* is expected in early September of 2008. Each of the eight sectors of the accelerator is in state of cooling and has nearly reached its operating temperature of 1.9 K.

The following sections give a summary of the *LHC* and its experiments. After a brief overview of the accelerator and the *ALICE* experiment, the *Transition Radiation Detector* (*TRD*) is described in more detail. The last section discusses the main objectives of the *Global Tracking Unit* as part of the TRD.

2.1 The Large Hadron Collider

The *Large Hadron Collider* (*LHC*) is a particle accelerator located near Geneva, Switzerland, in a circular tunnel[2] approximately 100 m beneath the surface and has a circumference of nearly 27 km. Two counter-rotating particle beams will collide at four specific interaction points, where the experiments are stationed to study the collisions. The center-of-mass energies will be about 14 TeV for proton-proton collisions and 1,150 TeV for heavy-ion collisions. To keep the beams on their circular track, over 1,000 super-conducting magnets are spread along the particle beam tubes, cooled down to a temperature of 1.9 K.
Figure 2.1 shows a schematic view of the *LHC* accelerator complex and its four major experiments *ALICE*, *ATLAS*, *CMS* and *LHCb*.
While *ATLAS*, *CMS* and *LHCb* are primarily designed to study proton-proton collisions, *ALICE* is the only experiment particularly developed to run in heavy-ion mode (Pb-Pb) as well. Its main objective is the detection of *quark-gluon plasma* (*QGP*) and the study of its properties.

[1] CERN was founded by several European states in 1954 and is located near Geneva, Switzerland.
[2] The tunnel was initially created for and used by the LHC predecessor *LEP* (*Large Electron Positron Collider*).

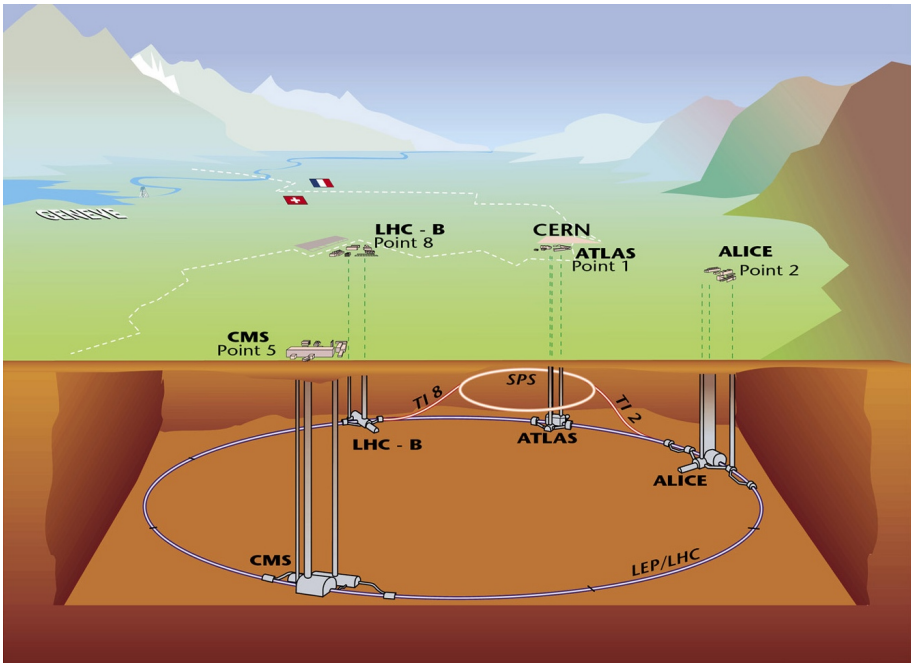


Figure 2.1: Schematic view of the *LHC* accelerator complex. The accelerator ring as well as the four major experiments *ALICE*, *ATLAS*, *CMS* and *LHCb* are situated in a tunnel about 100 m beneath the Franco-Swiss border near Geneva.

2.2 A Large Ion Collider Experiment

Collisions of heavy lead ions (^{208}Pb) are analyzed by *A Large Ion Collider Experiment* (*ALICE*). It is expected that quark-gluon plasma (QGP) is formed from these collisions at the energies provided by LHC. The detection and study of QGP and its properties is the main objective of *ALICE*, which consists of several sub-detectors complementing one another. In figure 2.2 an outline of the *ALICE* detector layout is given.

Particle and Vertex Tracking The innermost detectors of *ALICE* are responsible for capturing the tracks of electrically charged particles.
The *Inner Tracking System* (*ITS*) cylindrically surrounds the beam axis at the point of collision. It is capable of tracking particles at a very high resolution of up to 12 µm in order to localize the vertices.
The ITS is followed by the *Time Projection Chamber* (*TPC*), which is filled with Ne and CO_2 gas. When charged particles travel through the cylindrical gas volume, the gas molecules are ionized along the trajectory of the charged particles. The secondary electrons are accelerated by a strong homogeneous electric field parallel to the beam axis and drift towards the readout chambers. Two of the spatial coordinates of the ionization points are given by the pad position which the charged particles arrive at. The third coordinate is

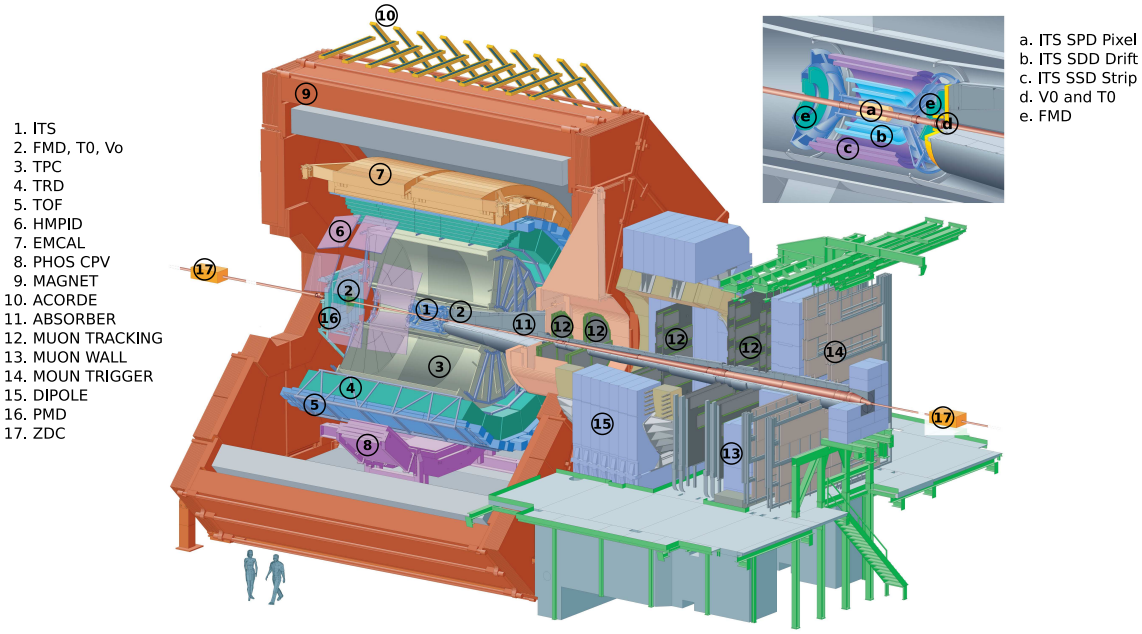


Figure 2.2: Overview of the *ALICE* detector layout. Source: [ALI]

determined by the time the secondary electrons need to reach the readout pads.

Particle Identification Adjacent to the TPC is the *Transition Radiation Detector* (*TRD*). Its main objective is to track and identify electrons with transversal momenta greater than about 3 GeV/c and to form a trigger contribution to decide on the readout of the TPC (*L1 trigger*). A more detailed description of the TRD follows in section 2.3.
The next detector following the TRD is the *Time of Flight* detector (*TOF*), which is capable of measuring the flight time with a resolution of 100 ps to determine the mass of high-energy particles. Additionally, TOF also participates in the L0 trigger decision.
The *High Momentum Particle Identification Detector* (*HMPID*) determines the mass of particles of extremely high energy. It is based on the detection of Cherenkov radiation, which is emitted when a charged particle travels through a dielectric medium at a phase velocity $v_p > c$.
The outer areas of the detector consist of the *Photon Spectrometer* (*PHOS*) and the *Electromagnetic Calorimeter* (*EMCAL*). Aside from contributing to the L0 and L1 trigger decisions, PHOS is designed to determine the temperature of the collision. Last, EMCAL provides the ability to identify photons directly and offers hadron rejection. Furthermore, it adds a jet-trigger and the capability to study medium-induced modification of jet fragmentation.

The entire detector complex is enclosed by the *L3-Magnet*, which produces a homogeneous magnetic field parallel to the beam axis. This magnetic field deflects charged par-

ticles in their trajectory depending on their momentum. The transversal momentum of a particle can be determined by the radius of its circular trajectory, since $r = \frac{p_t}{e \cdot B}$.

2.3 The Transition Radiation Detector

The *Transition Radiation Detector* (*TRD*) is used to identify and trace charged particles. Of special interest is the detection of electrons and positrons with transversal momenta greater than 1 GeV/c, because they indicate a *di-lepton decay* of heavy vector mesons (J/ψ, Y). The analysis of the appearance of those resonances provides a significant method to investigate QGP ([ALI01, MS86, AAA$^+$06]). However, since they are produced rarely (about 10^5 collisions for one Y), it is necessary to trigger on these events, which is performed by the TRD especially designed for this purpose. In the targeted momentum range, the production rate of pions is much higher than that of electrons. To differentiate between those particles, the effect of *transition radiation* is used. The tracks of the particles can be reconstructed by the charges detected at the readout pads of the drift chambers and the chronological sequence the charges arrive at the pads.

The TRD is divided into 18 entities (*Supermodules*) in azimuthal direction, forming a hollow cylinder. Each Supermodule consists of five *stacks* lined up in z-direction. A stack is made up of six *modules*, each consisting of radiator material, drift chamber and readout electronics. A detailed summary of the design and geometry of the TRD is given in [ALI01].

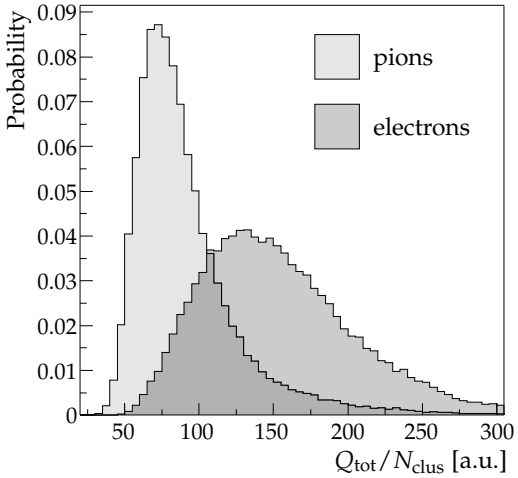


Figure 2.3: Average pulse height distribution for electrons and pions with a transversal momentum of $p_t \geq 3\,\text{GeV/c}$. Source: [ALI01]

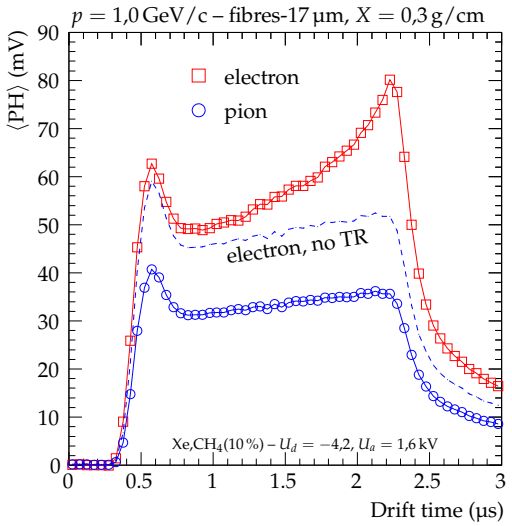


Figure 2.4: Pulse height development as a function of drift time for electrons and pions. Source: [ALI01]

Transition Radiation and Particle Identification Transition radiation is emitted when relativistic charged particles cross the interface of two media of different dielectric constants. A charged particle approaching this boundary represents an electric dipole in combination with its induced image charge. The dipole field intensity varies in time while the particle moves towards the boundary. Thus, electromagnetic radiation is emitted, which is highly collimated in forward direction. The energy of the emitted transition radiation is proportional to the Lorentz factor $\gamma = E/mc^2$ of the particle and typically corresponds to soft X-ray radiation. At same energies, the effect is more distinctive for particles of lower mass. It is used to distinguish between electrons and pions, which have a much higher production rate than electrons in the targeted momentum range. Because $m_{pion} \approx 273 m_{electron}$, pions hardly produce transition radiation compared to electrons. The chronological sequence of the signals triggered by electrons and pions is given in figure 2.4. Unlike electrons and pions, which produce a continuous ionization trail, photons of the transition radiation abruptly lose energy resulting in a spatially confined charge signal. The peak at the end of the drift time reflects the charges produced by the transition radiation.

Particle Tracking The TRD consists of six layers of drift chambers separated by radiator material, which causes crossing particles to produce transition radiation. The chambers are filled with Xe and CO_2 gas and divided into two regions, the *drift region* and the *amplifier region*. The drift region is flushed by a homogeneous electric field. A charged particle crossing the gas volume ionizes the gas molecules and frees electrons, which drift with constant velocity due to the homogeneous field. However, the charge of these secondary electrons is too low to be detected. Thus, the amplifier region dominated by an inhomogeneous electric field of high intensity accelerates those electrons and triggers a cascade of further electrons, which can be measured. The particle track can be reconstructed by capturing the chronological sequence of the charges arriving at the readout pads. Both figure 2.5 and 2.6 show cross sections of a drift chamber of the TRD.

2.4 Objectives of the Global Tracking Unit

The study of quarkonia and jet production in heavy ion collisions provides an effective way to research QGP. The *Global Tracking Unit* (*GTU*) is designed to trigger on both, di-lepton decays of heavy vector mesons (J/ψ and Y) and jet production [dC03, dC, Ret]. In order to associate the particles detected with their emerging point to verify a di-lepton decay, it is necessary to reconstruct their trajectories and transversal momenta [dC03]. Based on the results of these track matching calculations, the *GTU* contributes to the *L1 trigger decision*[3], which is evaluated by the *Central Trigger Processor* (*CTP*). Depending

[3] A L1 trigger is issued by the Central Trigger Processor to indicate an event of interest and start further detector readout. See [Ret07] and [Kir07] for a detailed explanation of the TRD trigger hierarchy.

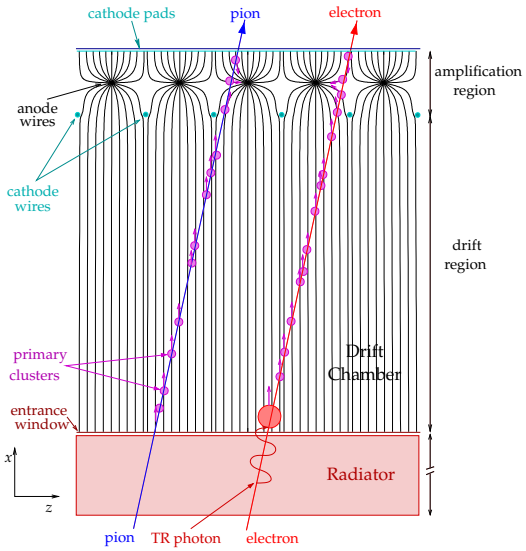


Figure 2.5: Projection of a module segment to the x-z-plane. Source: [ALI01]

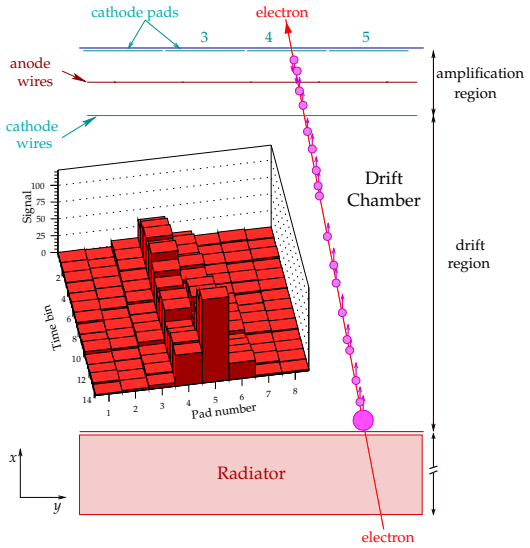


Figure 2.6: Projection of a module segment to the x-y-plane. The histogram shows the pad signal strength distribution for the drift time interval. Source: [ALI01]

on the trigger sequence, the *GTU* reads out and buffers the event raw data from the detector for further processing and forwarding to the *High Level Trigger* (*HLT*) and the *Data Acquisition System* (*DAQ*) [Ret07, Kir07].
Figure 2.9 gives an overview of the TRD readout chain and the *GTU* data path. Data from the detector is transmitted by the TRD *front-end electronics* via 1,080 optical fibers, which provide an aggregate bandwidth of 2.7 GB/s per detector stack (12 fibers). Because the trigger contribution must be calculated within less than 2 µs, track matching and trigger processing demand for a low-latency implementation. To minimize the dead time of the detector, the large amount of event raw data is read out at the full bandwidth provided by the optical fibers (green), which connect the TRD font-end electronics to the *GTU*.

Reconstruction of Particle Track Segments Projected on a plane perpendicular to the beam axis (x-y-plane of the TRD), a charged particle passes through the detector on a circular path due to the longitudinally directed, homogeneous magnetic field of the L3-Magnet. The transversal momentum p_t of the particle can be calculated by determining the radius r of the circular particle track, since radius and transversal momentum are correlated by $r = \frac{p_t}{e \cdot B}$. Considering the transversal momenta of interest ($p_t \geq 3\,\mathrm{GeV/c}$), the radius is of the order of 25 m. That is, the track segments detected within the drift chambers are weakly bent and can be approximated with segments of a straight line (*tracklets*). If the particle is assumed to have emerged from the primary interaction point, gradient and axis intercept are sufficient to describe the particle track within a detector module. The tracklets are fitted and parameterized by the TRD front-end electronics. It

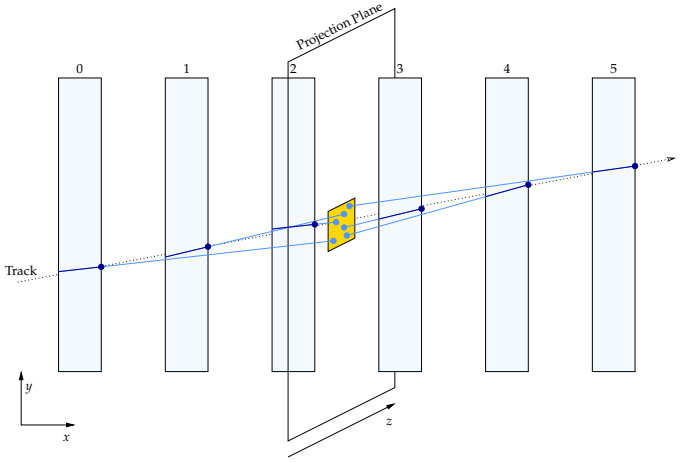


Figure 2.7: Illustration of the *window criterion* for matching a particle track. Source: [dC03]

consists of *readout boards* supplied with the *Multi-Chip Modules* (*MCM*s), which buffer and preprocess the event raw data [Gut02, Gut06, Sch08].

Track Matching The *Track Matching Units* (*TMU*s) of the *GTU* are responsible for reconstructing the track of a particle, which passed the associated detector stack. The reconstruction is based on the tracklets received. The tracklets of all detector layers of a stack are extended straightly and projected to an imaginary plane in the middle of the detector stack (see figure 2.7). Tracklets from different layers, which have both intersection points in close vicinity on the projection plane and similar gradients, are considered as a track (*window criterion*). The transversal momentum of the particle is estimated from the track radius and compared to a threshold (see [dC03], [dC]).
Projected on the x-y-plane, both track segments and interaction point of a particle of interest are on a circular arc. According to calculations by J. de Cuveland, it is sufficient to fit a line to the tracklets, which can be seen as a secant of the circular arc. The radius of the track can be determined by the line parameters a and b (see figure 2.8) and the following formulas [dC03]:

$$r = \frac{d_{12}/2}{\sin(\alpha)} \tag{2.1}$$

where

$$\alpha = \varphi_2 - \varphi_1 = \arctan\left(\frac{y_2}{x_2}\right) - \arctan\left(\frac{y_1}{x_1}\right) \tag{2.2}$$

$$d_{12} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad , \qquad y_i = a + b \cdot x_i \quad , \qquad i \in \{1, 2\} \tag{2.3}$$

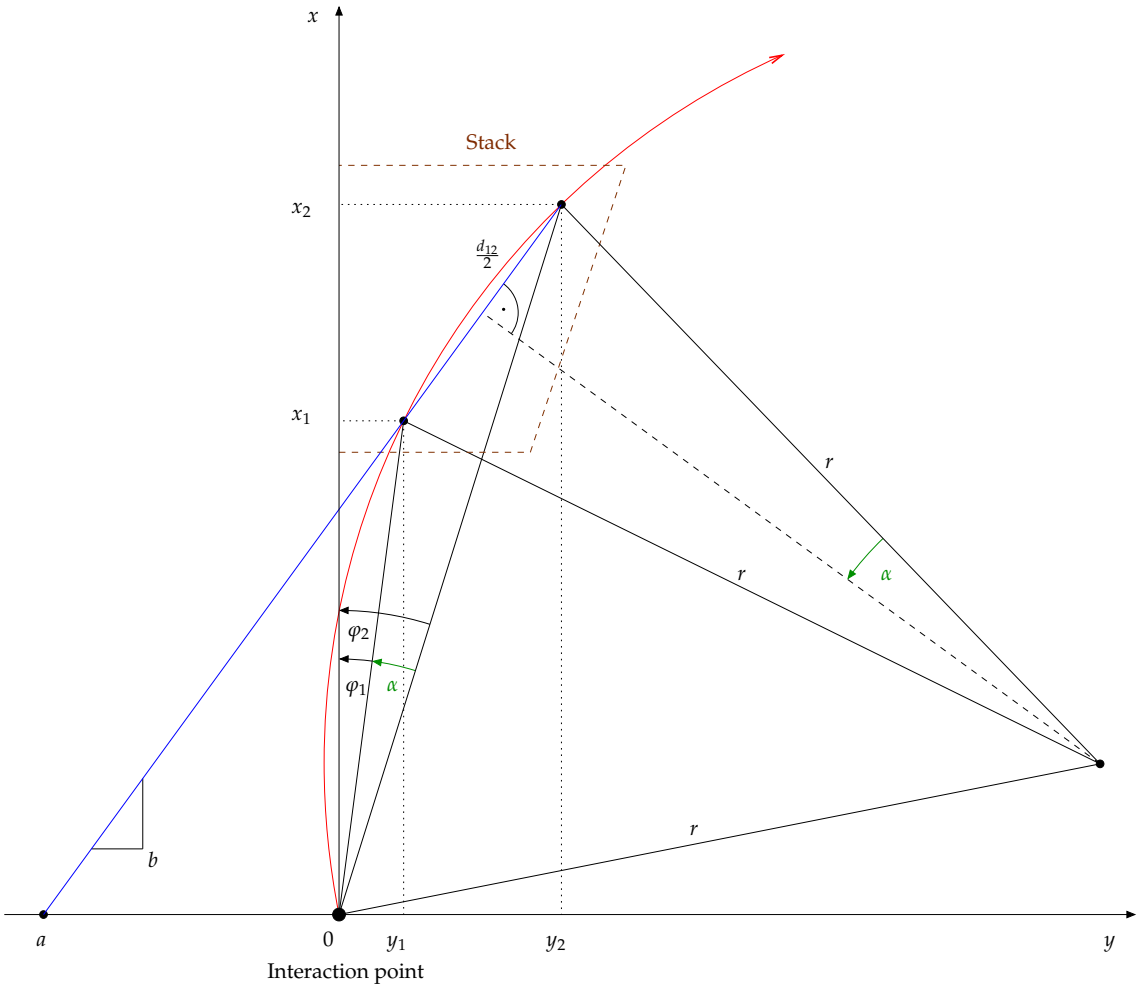


Figure 2.8: Reconstruction of the transversal momentum of a particle. Assuming the particle to have emerged from the interaction point in the beam axis (z-axis of the detector), another two points of the circular track are sufficient to determine the track radius. The track segment inside a detector stack can be approximated by a straight line because of the high transversal momenta of the particles. Considering the line as a secant of the circle, the radius can be determined from the intersection point a of the line with the y-axis and the gradient b of the secant. Source: [dC03]

The results of the TMUs are processed by the *Supermodule Units* (*SMUs*) and the *Trigger Unit* (*TGU*), which contribute to the L1 trigger decision of the CTP.

Event Readout When the CTP signals a L1 trigger, the *SMUs* issue control signals to the *TMUs* to start the readout of the event raw data. The data is transmitted by the TRD front-end electronic via 12 optical fibers per detector stack at a high bandwidth to minimize the dead time of the detector. The data is buffered in the local SRAM of the *TMUs*, which is capable to capture the data at the full bandwidth provided by the optical lines. Each *TMU*

handles the data of one detector stack, meaning a Supermodule is completely covered by five *TMU*s [Ret07]. The buffered data of all stacks of a Supermodule is merged and formatted by the corresponding *SMU* and forwarded to the *DAQ* and the *HLT* [Kir07].

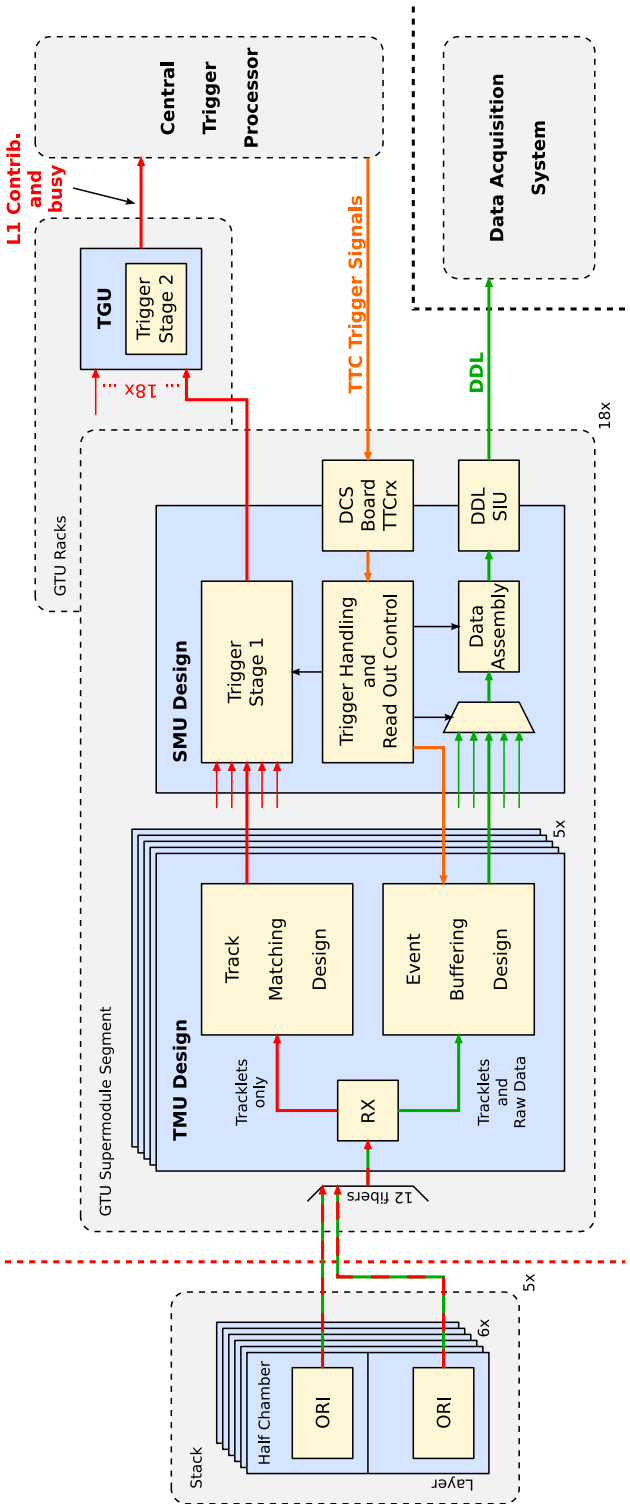


Figure 2.9: Overview of the GTU data path. Track matching and trigger processing (red) require a low-latency implementation, because the trigger contribution must be delivered 6.2 µs after the interaction. The dead time of the detector is minimized by using the full bandwidth provided by the optical fibers to read out the event data (green).

3 The Global Tracking Unit

The main objective of the *Global Tracking Unit* (*GTU*) is to determine the transversal momenta of the detected particles in order to contribute to the L1 trigger decision. The particle tracks are reconstructed from the parameters of the tracklets fitted to the track segments detected in the drift chambers of the Transition Radiation Detector (TRD). For reconstruction and the subsequent decision on the contribution to the L1 trigger, a time period of less than 2 µs is available. The *GTU* also reads out and buffers the event raw data of the detector at high data rates (approx. 2.7 GB/s per detector stack) for forwarding to the Data Acquisition System (DAQ) and the High Level Trigger (HLT).

In this chapter, an overview of the layout and the modules of the *GTU* is given. While the first section describes technical details, the second one discusses how communication with the system is established for monitoring and controlling purposes.

3.1 Technical Aspects

The *GTU* is located in the *ALICE* pit outside the L3-Magnet, below the muon detector arm. According to the layout of the TRD, it consists of 18 sub-entities (*segments*), each associated with one Supermodule[1] of the detector. The segments are placed in groups of two in a 19"-crate. The resulting nine crates are distributed over three racks. A segment is composed of five *Track Matching Units* (*TMU*s), which receive data from one Supermodule via optical fibers and reconstruct the particle tracks (see section 2.4). Further, each segment is supplied with one *Supermodule Unit* (*SMU*) responsible for receiving trigger signals from the Central Trigger Processor (CTP) and controlling a the *TMU*s of the segment based on the trigger sequence. An *SMU* also merges and formats the buffered event data of all five *TMU*s of a segment and forwards it to the HLT and DAQ. An *LVDS* backplane is installed on the back of each segment and allows high-speed communication between an *SMU* and the five *TMU*s. A *CompactPCI* backplane is used to supply power, as well as for control and monitoring purposes. In addition, one crate is equipped with the *Trigger Unit* (*TGU*). It evaluates the trigger-relevant information of all 18 *SMU*s to decide on the L1 trigger contribution to be sent to the CTP. Figure 3.1 gives an overview of the physical layout of the *GTU* and its segments.

[1] 18 Supermodules are subsequently arranged around the beam axis and form a hollow cylinder. Each Supermodule consists of five detector stacks. A stack is composed of six layers of modules. A module is

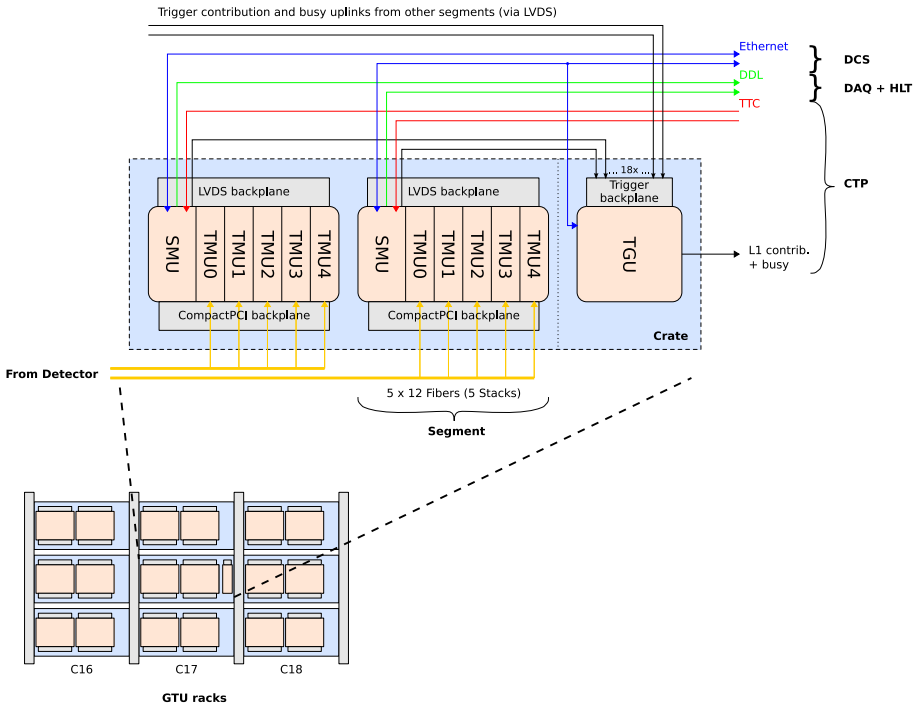


Figure 3.1: Physical layout of the *GTU*. Signal connections and other details are shown as an example for the crate housing the Trigger Unit. The *GTU* and its necessary infrastructure is located in racks C16, C17 and C18 below the muon detection arm inside the *ALICE* pit.

Board Layout The three board types (in the following referred to as *XMU*s) have been developed by J. de Cuveland within the scope of his dissertation [dC]. They are based on a 14-layer CompactPCI board with an identical *PCB*[2] layout of six units of height and only differ in the components equipped. In figure 3.2 a simplified schematic view of the *TMU* board is given as an example. Figure 3.3 shows a photography of the front side and the back side of the *TMU*. The most relevant components are labeled.
The central element common to all boards is a *Xilinx Virtex-4 FX100 FPGA*[3]. This powerful device provides 94,896 *Look-Up Tables* (*LUT*s) and 768 freely usable I/O pins. It also offers a number of specialized functional blocks like *Digital Clock Managers* (*DCM*s) to derive clocks from a reference clock frequency and 20 *Multi-Gigabit Transceivers* (*MGT*s) capable of processing serial data at gigabit rates. The MGTs are used to make the incoming detector data available in the FPGA. Furthermore, two *embedded PowerPC 405* processor cores are integrated into the FPGA and used in a software-hardware co-design for monitoring and configuration purposes.

made up of drift chamber, radiator material and readout electronics. See [ALI01, dC03].
[2]PCB: Printed Circuit Board
[3]FPGA: Field Programmable Gate Array. A semiconductor device containing a large number of configurable logic blocks, which are connected by a programmable switch matrix.

Each *XMU* is supplied with a flash PROM[4] to store the FPGA configuration, which is uploaded to the FPGA device on either power-on, reset or on request. The FPGA configuration is programmed to both FPGA and PROM via *JTAG*[5]. A detailed description of the JTAG programming mechanism is given in [Kir07].
Data from the detector is transmitted to the *GTU* by the detector front-end electronics via optical fibers. Each *XMU* is supplied with several slots for *Small-Form-Factor-Pluggable* (*SFP*) modules[6] to connect the fibers. These modules convert the optical gigabit signals into electrical signals and vice versa and connect them to the MGTs. Two fast 512 Kb x 36 DDR2 SRAM chips[7] are used to buffer the incoming event data from the detector for further processing.
Most important to this thesis are the two embedded PowerPC 405 processor cores, which are the central elements of the Embedded PowerPC System described in the next chapter. A 64 MB DDR2 SDRAM chip[8] serves as main memory for the processors. To provide the boards with non-volatile memory as well, each *XMU* has a slot for an SD Memory Card. Finally, several sensors are available to monitor operating voltages and temperatures for both PCB and FPGA. The sensors are accessed via *I^2C buses*.
Compared to the *TMU*s, the assembly of both the *SMU*s and the *TGU* is slightly different. *SMU*s and *TGU* are provided only with four SFP slots preserved for future applications such as *multi-gigabit Ethernet* (see chapter 4 and 5). Both board types are also equipped with a daughter board, the *Detector Control System* (*DCS*) board. It is used for monitoring and administration purposes throughout *ALICE* and represents the connection of these *XMU*s to the *TRD network*. Additionally, the *SMU*s are supplied with another board called *Source Interface Unit* (*SIU*). It is optically connected to the Data Acquisition System (DAQ) and the High Level Trigger (HLT) and used for the event data transfer. To communicate with the CTP, the *TGU* is provided with a special adaptor board, which transmits the trigger contribution signals via LVDS lines. A more extensive discussion of the board components listed above is given in [Ret07] and [Kir07].

Embedded PowerPC System The Xilinx Virtex-4 FX100 FPGA is supplied with two embedded PowerPC 405 processor cores allowing to provide the boards with software, which is capable to interact with the *GTU* components. The hardware design containing the processor and its peripheral components is referred to as *Embedded PowerPC System* and is described in the next chapter. Although it is an integral part of the *GTU* design, it is developed separately with different design tools. The Embedded PowerPC System

[4] Xilinx XFC32P
[5] JTAG: Joint Test Action Group. JTAG names the IEEE 1149.1 standard (Standard Test Access Port and Boundary-Scan Architecture) for test access ports designed for testing PCBs.
[6] These modules are standardized devices to transmit optical data. Each module is equipped with a sender and receiver unit.
[7] Cypress CY7C1320AV18
[8] Samsung K4T51163QC-ZCD5. The chip is organized as a device of 32 Mb x 16.

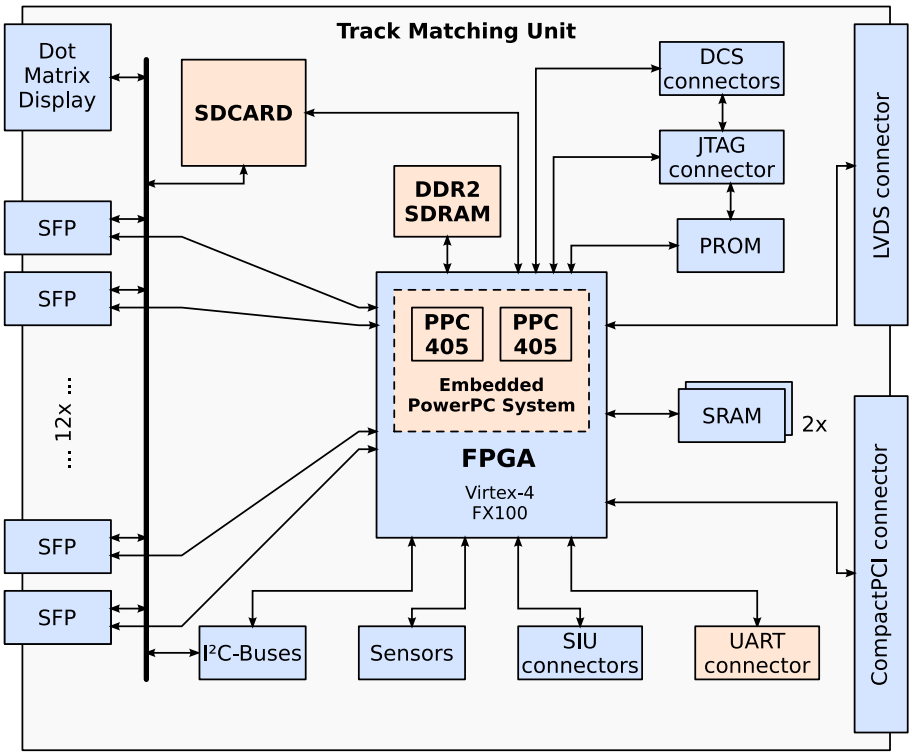


Figure 3.2: Simplified diagram of a *TMU* board. The components most relevant to this thesis are colored orange.

represents the basic hardware design to monitor and administrate the *GTU* via *slow control*.

3.2 GTU Remote Access

The *Detector Control System board* is used for remote accessing objectives throughout *ALICE* and provides the necessary capabilities to monitor and control the *GTU* over the TRD network.

The Detector Control System Board The *Detector Control System* (*DCS*) board represents the interface to monitor and administrate the *GTU* and is located on the back of each *SMU* and the *TGU*. Its central element is an *Altera Excalibur* FPGA, which is provided with an embedded *ARM RISC* processor core [Alt02a, Alt02b]. The board also supplies 32 MB SDRAM and networking hardware, which makes it capable of running an Embedded Linux System and being accessed via the TRD network. Moreover, the board is equipped with a *TTCrx* chip to decode trigger signals received from the CTP via optical fibers and forward them to the *SMU*s. The JTAG interface permits to remotely reprogram

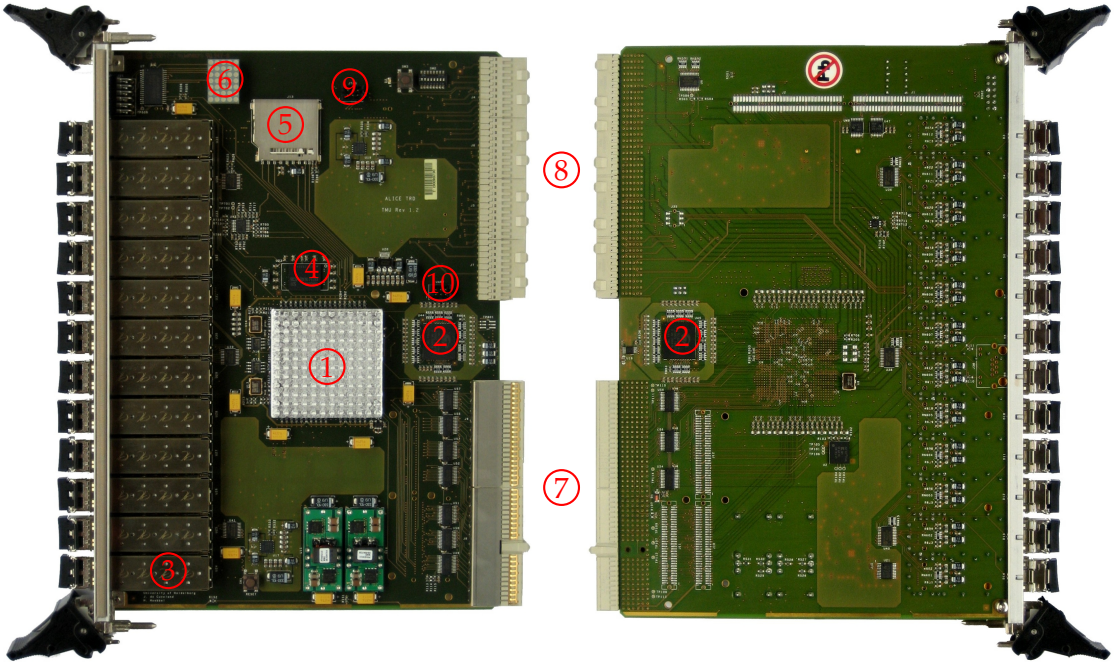


Figure 3.3: The photography shows a TMU board fully equipped with 12 SFP modules as installed at *CERN*. The labeled parts are: (1) FPGA, (2) 2 x 18Mb DDR2 SRAM, (3) SFP module, (4) 64 MB DDR2 SDRAM, (5) SD Memory Card slot, (6) LED dot matrix, (7) CompactPCI connector, (8) LVDS backplane connector, (9) JTAG connector, (10) UART connector.

and reconfigure both flash PROM and FPGA of the *XMU*s, respectively. However, in order to communicate with the Embedded PowerPC System, the standard hardware and software of the DCS board had to be extended. The extensions included to implement an additional *UART*[9] core into the DCS FPGA design to serve as communication interface as well as to develop a Linux device driver to access this core. The necessary modifications were carried out by S. Kirsch and M. Schuh (see [Kir07] and [Sch07]).

The Shared UART Interface Accessibility from anywhere inside the TRD network and the set of standard tools of the Embedded Linux make the DCS boards ideal hosts to remotely control and monitor the *GTU*. However, only two data lines are available per DCS board for communication with the *XMU*s of one *GTU* segment due to the limited number of FPGA I/O pins. Thus, a serial communication protocol was chosen and implemented by using a UART interface.
Because the fixed UART peripheral of the Excalibur device is already in use, an open source UART soft core from *opencores* [JGM] was added to the DCS FPGA design [Kir07].

[9]UART: Universal Asynchronous Receiver/Transmitter. A data word of the UART data stream consists of one start bit, five to nine data bits, an optional parity bit, and one stop bit.

Furthermore, the Embedded PowerPC System of the *XMU* FPGA designs needed to be supplied with a UART soft core as well.

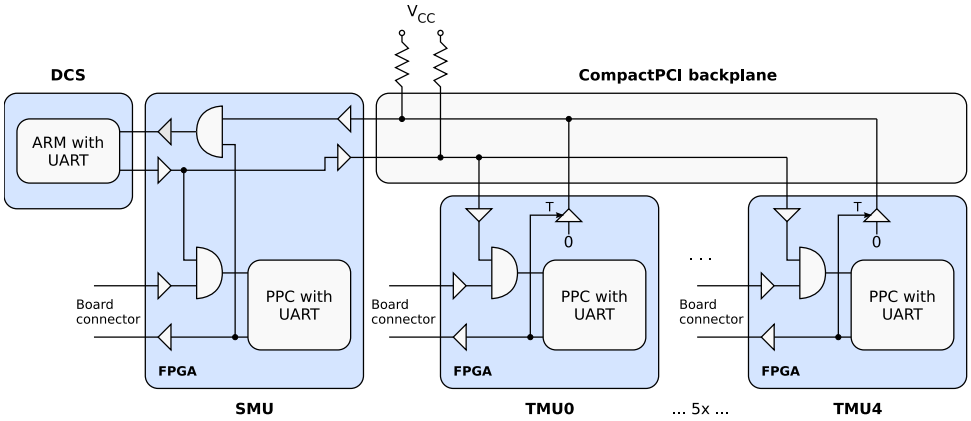


Figure 3.4: Serial communication between a *GTU* segment and the DCS board. The RX and TX data lines of a DCS board are shared by both the *SMU* and the *TMU*s of a *GTU* segment.

Figure 3.4 gives an idea about the UART connection between a DCS board and a *GTU* segment. As illustrated, it is characteristic for the DCS UART soft core interface to be shared by both *SMU* and *TMU*s of a segment. The RX and TX data lines of the DCS UART are connected to the FPGA of the *SMU* and distributed to the *TMU*s via two data lines of the CompactPCI backplane. Due to the presence of *pull-up resistors*, these two data lines are only driven low actively. Thus, in order to merge the DCS UART TX signal with the input signal of the local UART connector of each board, it is necessary to use a logical **AND** for signal combination. The same applies to the local UART TX signals of all *TMU*s, which are combined with the local *SMU* TX signal and connected to the DCS UART RX port.

The Control Software The software to communicate with the *XMU*s is part of the Embedded Linux of each DCS board and consists of two applications. One is responsible for remotely reconfiguring and updating both flash PROM and FPGA via JTAG. The other provides the software interface to communicate with the Embedded PowerPC System via UART for monitoring and controlling purposes. The applications each comprise a Linux device driver, which manages the access to the JTAG and UART interface, respectively, and a main program providing the necessary interfacing commands.

Remote FPGA configuration is done by the JTAG programming tool (*xsvfplayer*). It executes the high-level IEEE 1149.1 (JTAG) bus operations given in an XSVF[10] file to program the PROM or FPGA device with the corresponding FPGA configuration [Kir07].

[10]XSVF: Xilinx Serial Vector Format. Vendor-specific file format used to record JTAG operations by describing the information, which needs to be shifted into the device chain. See [Xil07d].

To communicate with the *GTU* via the UART interface, another application (*gtucom*) is used. It provides commands to interface with the *GTU* system software running on the Embedded PowerPC System of the individual boards [Sch07]. The *GTU* system software is capable of configuring and monitoring a large number of system parameters of the *GTU* at runtime. In particular, all *GTU* status registers and diagnostic entities can be accessed by the system software. It also retrieves information about the board status such as temperature or supplied voltages. This allows the *GTU* to be monitored and administrated from anywhere on the TRD network.

4 The Embedded PowerPC System

The Virtex-4 FX100 FPGA provides two embedded PowerPC 405 processor cores, which are used in a software-hardware co-design to monitor and control the *GTU*. Several peripheral components are attached to the processor bus and form the *Embedded PowerPC System*. The great benefit of this co-design is the ability to read and modify operating parameters of the *GTU* at runtime in a straightforward manner by using flexible software. The hardware part of the embedded system is developed separately from the *GTU* main design. In the present context, some of the peripherals are important to a basic working system, such as communication interface or memory controller. The others are responsible for interfacing system components of the *GTU*.

This chapter describes the layout of the Embedded PowerPC System and the most important components. First, a brief overview of the PowerPC 405 processor core is given, followed by a description of the peripheral components. Finally, a short summary with suggestions for future extensions regarding the capabilities of the system concludes this chapter.

4.1 The PowerPC 405 Processor

The IBM PowerPC 405 processor is a 32-bit implementation of the *PowerPC embedded-environment architecture*, which is derived from the PowerPC architecture[1]. In particular, the processor core integrated into Xilinx Virtex-4 FX100 devices is a PowerPC 405F6 core[2]. Figure 4.1 shows a high-level block diagram of this processor core and its most important internal components.

Central Processing Unit (CPU) The *Central Processing Unit* is based on a 32-bit RISC Harvard architecture and implements a five-stage instruction pipeline as well as *branch prediction* to improve the efficiency of the execution of instructions. Its execution unit contains the *General Purpose Register* file (*GPR*), which consists of 32 32-bit registers, the *Arithmetic-Logic Unit* (*ALU*), and the *Multiply-Accumulate Unit* (*MAC*). An integrated *Floating Point Unit* (*FPU*) is not provided.

[1] The PowerPC architecture originally provides a 64-bit memory model.
[2] See [IBM06] for more detailed information about this specific processor core.

Memory Management Unit (MMU) The *Memory Management Unit* provides address translation from logical to physical address space, protection functions and control of storage-attributes for this address space. It supports various page sizes from 1 KB up to 16 MB. A fully associative *translation look-aside buffer* (*TLB*) is used to improve the performance of address translation. TLB contention between data and instruction accesses is prevented by a 4-entry instruction and an 8-entry data shadow-TLB maintained by the processor transparently to software.

Instruction-Cache and Data-Cache Units Despite using a Harvard architecture, the processor accesses the memory by two separate two-way set-associative cache units, the *instruction-cache unit* (*ICU*) and the *data-cache unit* (*DCU*). Both ICU and DCU are each 16 KB in size and supplied with cache arrays, a cache controller and a *Processor Local Bus* (*PLB*) master interface to fetch data and instructions, respectively, from memory devices attached to the PLB.

Timer Resources The processor is provided with two timers, the 64-bit incrementing *time base*[3] and the 32-bit decrementing *programmable interval timer*, which are both clocked at the processor clock frequency. In addition, three *timer-event interrupts* are supported:

- Programmable Interval Timer (PIT) event interrupt
- Fixed Interval Timer (FIT) event interrupt
- Watchdog Timer (WDT) event interrupt

When the PIT register is set to a non-zero value, it starts decrementing immediately and triggers a PIT event interrupt as soon as its contents reaches zero. In contrast, a FIT interrupt occurs if a specific bit in the time base lower register changes from zero to one. A WDT event is triggered in a manner similar to FIT, but can additionally cause a hardware reset. Watchdog Timers are of special interest for embedded systems, which need to be self-reliant. Normally, the operating software of such a system periodically clears the WDT event. However, in case of a software failure, which prevents the event from being cleared, a Watchdog-timeout occurs and causes a hardware reset to bring the system back to normal operation.

Debug Logic The PowerPC 405 *debug logic* supports both an *internal* and an *external* debugging mode. The internal mode is used during normal program execution for debugging system software and applications. However, to test system hardware as well as software, external debuggers can be interfaced via the JTAG port of the processor. The *Xilinx Microprocessor Debugger* (*XMD*) is a debugging software, which makes use of the

[3]The time base is actually implemented as two 32-bit registers. At a clock frequency of 400 MHz, a time base *roll-over* occurs about every 1,500 years.

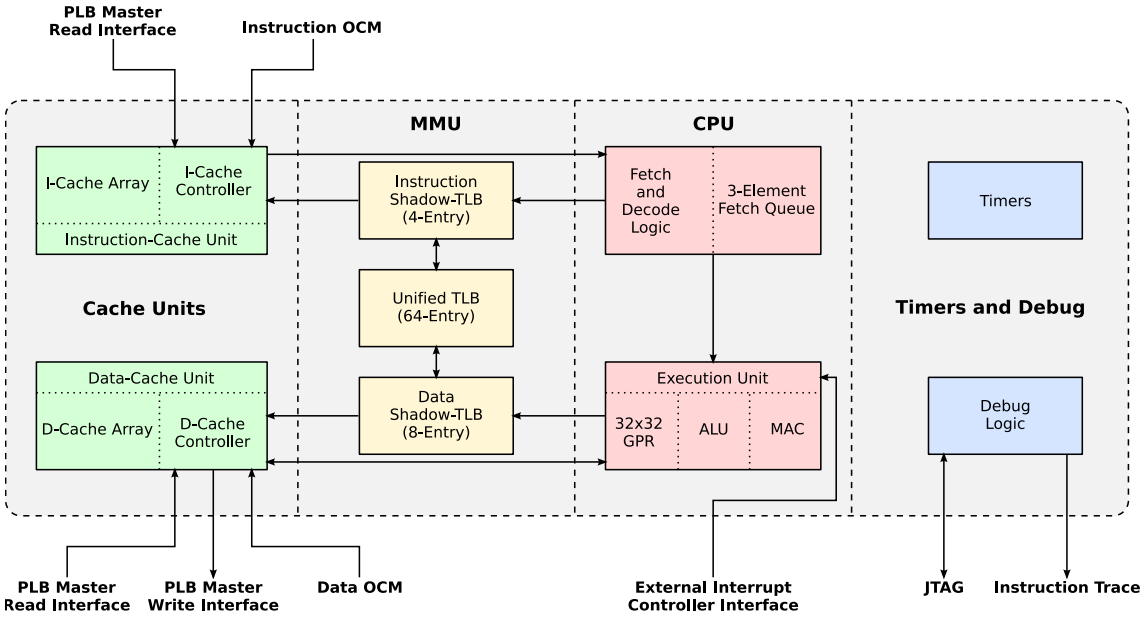


Figure 4.1: PowerPC 405 processor core block diagram. Source: [Xil08b]

external debugging mechanism. It has been constantly used for development during this thesis to initialize the boot memory of the test setup with software and to monitor the program execution.

Interfaces To support the attachment of peripherals, the PowerPC 405 processor provides a set of interfaces as follows:

- The *Processor Local Bus (PLB) interface* provides a 32-bit address bus and three dedicated 64-bit data buses attached to the instruction-cache and data-cache units. One of the 64-bit buses is connected to the *instruction-side PLB interface* (ISPLB) and enables the ICU to fetch instructions from any memory device connected to the PLB. The other two 64-bit buses are attached to the DCU through the *data-side PLB interface* (DSPLB) to provide read and write access to PLB memory devices via separate buses.
- The *Device Control Register (DCR) bus interface* provides a mechanism to configure, control and hold status of peripheral devices, which are not part of the processor block but reside on the same FPGA chip. The devices can be attached to the device control registers, which are accessible by dedicated processor instructions.
- The *On-Chip Memory (OCM) interface* is used to attach additional memory to the instruction and data caches. The additional cache memory can be accessed at performance levels matching the cache arrays.
- The *JTAG port* interface supports attachment of external debugging devices.

- The *On-Chip Interrupt Controller interface* combines asynchronous interrupt inputs from sources both inside and outside the FPGA device and passes them to the PowerPC 405 processor core.
- The *clock and power management interface* provides several methods of clock distribution and power management. It enables power-sensitive applications to control the processor clock using external logic.

For further and more detailed information about the internal components and interfaces of the PowerPC 405 processor, please refer to [Xil08b] and [Xil07c].

4.2 Embedded PowerPC System Configuration

The present Embedded PowerPC System for the *GTU* consists of one PowerPC 405 processor and the peripheral components attached to the PLB. In order to make them accessible to the processor, they are mapped into its address space. The peripherals of the present embedded design are split into three groups. A schematic overview of the Embedded PowerPC System is shown in figure 4.2.
The *core components* provide a basic configuration for communication with the system and debugging purposes. These components are:

- A PowerPC 405 processor core
- A BRAM instance, which serves as the boot memory, and a controller to access it
- A *reset block* used to generate appropriate reset signals
- An *interrupt controller*, which passes interrupt inputs from sources both inside and outside the FPGA device to the CPU
- A *JTAG controller* to attach external debugging hardware
- A *UART component*, which provides basic communication over a serial interface

On systems, which are not equipped with additional on-board memory, BRAM cells of the FPGA device are used to serve as instruction and data memory for the processor. In case of the *GTU*, each of the *XMU* boards is supplied with a DDR2 SDRAM chip to be used as processor memory instead. However, at least one BRAM instance is still required as boot memory, which is initialized with boot code by the FPGA configuration. Although the Virtex-4 FX100 FPGA provides two embedded PowerPC 405 cores, only one processor is currently in use, while the second remains unconnected.

The *board-specific peripherals* are related to hardware, which is supported by corresponding *IP cores*[4] supplied by the development tools. This class currently includes the DDR2

[4] IP core: Intellectual Property core. A design entity with dedicated functionality. See *http://www.xilinx.com/ipcenter/*.

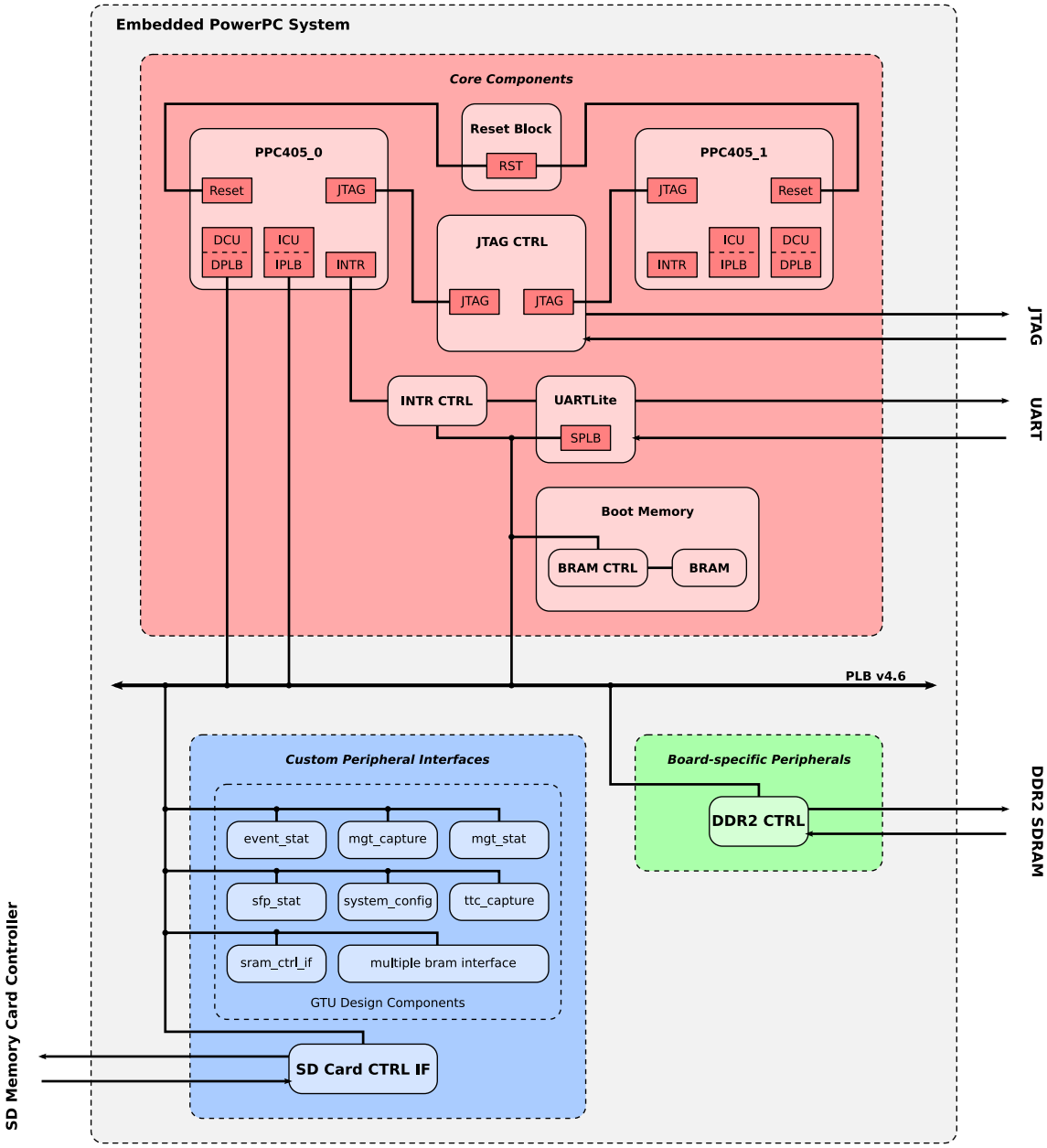


Figure 4.2: Block diagram of the Embedded PowerPC System. The individual components are split into three groups. The *core components* form the basic configuration to communicate with the embedded system and provide debugging capabilities. *Board-specific peripherals* are additional components, which are already supported by corresponding IP cores. The *user peripherals* provide custom-developed hardware peripherals to attach the *GTU* system components.

Multi-Port Memory Controller (MPMC) only. Other components such as the *TEMAC*[5] are to be added for future extensions.
Finally, support for custom-made hardware peripherals is provided by the *custom peripheral interfaces*, including:

- Multiple instantiations of an interface to attach both registers and BRAM instances of *GTU* system components to the PLB and make them accessible to software. The BRAMs and registers are used to configure, control and hold status information of the system components.
- An interface, which allows to connect the custom-built SRAM controller to the PLB.
- An interface providing access to SD Memory Cards via the SD Memory Card Controller described in chapter 6.

The following subsections describe the most important peripherals of the Embedded PowerPC System.

4.2.1 The Processor Local Bus

Most of the components shown in figure 4.2 are attached to the *Processor Local Bus* (*PLB*) of the PowerPC 405 processor[6]. The bus entity consists of a control unit, a watchdog timer and separate address, write and read data path units. The PLB v4.6 as used for this design supports up to 16 master and eight slave devices. Furthermore, it has a 32-bit wide address bus and a 32, 64 or 128-bit wide data bus. For the present design, the PLB is driven at a clock frequency of 100 MHz and a bus width of 32 bits is used.

4.2.2 The UART Communication Interface

Communication with the *XMU*s is currently done via the UART cores provided by the FPGA design of the DCS boards. These boards are used for administration and controlling purposes and are situated on the back of each *SMU* and the *TGU*. Each DCS board is responsible for communication with one *GTU* segment. A more detailed description of the DCS boards and their application is given in chapter 3.
To interface with a DCS board, the Embedded PowerPC System implements a UART IP core configured to transmit and receive serial data streams at a *baud rate* of 57,600 Bd. Each data word consists of a start bit, eight data bits and one stop bit, while a parity bit

[5] TEMAC: Tri-Mode Ethernet Media Access Controller. A core well suited for the development of high-density Gigabit Ethernet communications and storage equipment. See *http://www.xilinx.com/products/ipcenter/TEMAC.htm*.
[6] The reset block as well as the JTAG controller and the interrupt controller are directly connected to the CPU via dedicated interfaces. In addition, the interrupt controller is also attached to the PLB.

is not supported (57600/8N1). Figure 3.4 shows the physical connection layout between a DCS board and the *XMU*s of one segment.

4.2.3 The DDR2 SDRAM Controller

The DDR2 SDRAM controller is used to access the 64 MB DDR2 SDRAM chip each of the *GTU* boards is provided with and serves as instruction and data memory for the PowerPC 405 processor.
The *Multi-Port Memory Controller* (*MPMC*) is a highly parameterizable IP core supporting SDRAM, DDR and DDR2 memory (see [Xil08a]). In addition, the controller configuration tool provides a collection of pre-configured settings for various memory chips.
Both controller and DDR2 SDRAM are driven by the global system clock frequency of 200 MHz, which is within range of the memory chip specification[7]. To ensure a reliable DDR2 memory, a long-term test has been run, which continuously writes and reads the entire memory and counts the number of detected bit errors. The test results are listed in section 4.4.

4.2.4 The Custom Peripheral Interfaces

The custom peripheral interfaces connect the *GTU* system components to the Embedded PowerPC System. Due to the upgrade to another bus architecture[8], the existing custom peripherals needed to be re-designed and adapted.
The *GTU* system components provide BRAM instances and register banks, which allow to configure and hold information about the status of the system components. The peripheral interfaces used to access both BRAMs and register banks via control and monitoring software are referred to as *BRAM interfaces* and are provided with a parameterizable design to adjust them accordingly.
Aside from those BRAM interfaces, there are another two interfaces to attach the SRAM controller and the SD Memory Card Controller, respectively. Since they are very specific and performance-relevant, these interfaces are implemented as non-customizable entities.

Single BRAM Interface The *single BRAM interface* is primarily designed to access continuous address regions, which are not part of the Embedded PowerPC System, such as BRAM instances or register banks of the *GTU* system components. The address and data bus width is set to 10 and 32 bits, respectively, by default but can be adjusted within

[7] The minimum and maximum clock frequency of the memory chip specification are 125 MHz and 266 MHz, respectively. See [SAM05].
[8] With version 10.1 of the development tools, the *On-Chip Peripheral Bus* (OPB) architecture has been declared obsolete by Xilinx.

a specific range by using *generic parameters* (*generics*) to match particular requirements. Another customizable property is the address alignment, which is typically set to byte-alignment or word-alignment. A list of the generics can be found in table 4.1 below.

Generic	Range	Default	Description
U_AWIDTH	0 to 32	10	Width of address bus
U_DWIDTH	0 to 32	32	Width of data bus
U_ALIGNMENT	0, 1, 2	2	Address alignment 0: byte, 1: half-word, 2: word

Table 4.1: The BRAM interfaces provide generic parameters for customizing the width of the address bus and the data bus, respectively. Another generic is used to designate the address alignment. The default values are adjusted to a 18 Kb BRAM with 32-bit aligned addresses.

A diagram of the data path of the interface is shown in figure 4.3. Address, read and write data from the PLB (`Bus2IP_Addr` `IP2Bus_Data` and `Bus2IP_Data`) are directly connected to the BRAM interfacing ports. Several control signals are derived from `Bus2IP_CS` (chip select) and `Bus2IP_RNW` (read-write) signals of the PLB, including the BRAM write enable signal `wr_en` and the PLB write acknowledge signal `sig_wr_ack`. In case of a read transaction, data is available on the BRAM output port registers one clock cycle after the address has been assigned. Thus, the PLB read acknowledge signal `sig_rd_ack` has to be delayed by one clock cycle.

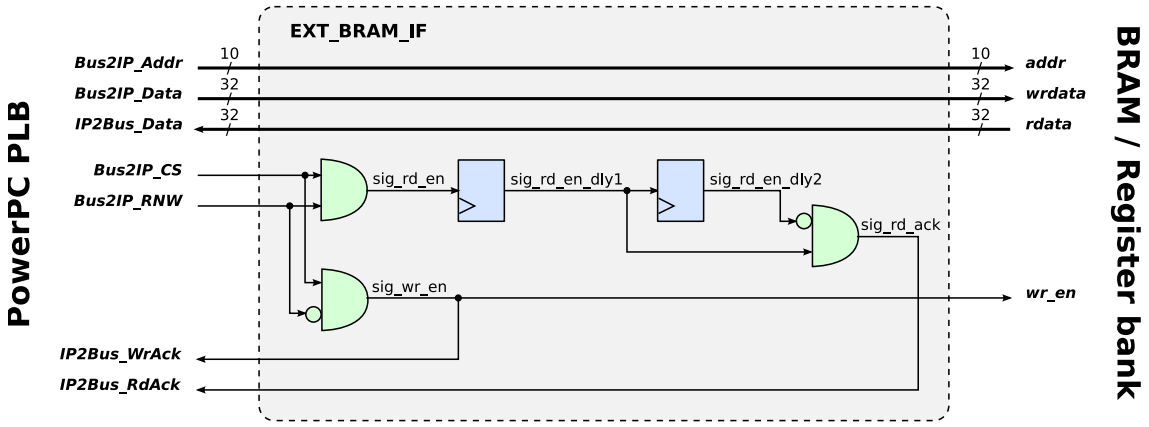


Figure 4.3: Diagram of the single BRAM interface. Bus address, read data and write data are directly connected to the BRAM interfacing ports. The parameterizable bus widths are given for an 18Kb BRAM entity as an example. Because data is available at the BRAM output ports one clock cycle after an address has been assigned, the PLB read acknowledge signal is delayed.

Multiple BRAM Interface The *multiple BRAM interface* allows to attach up to four BRAM entities per interface instance. It was developed to avoid unnecessary consump-

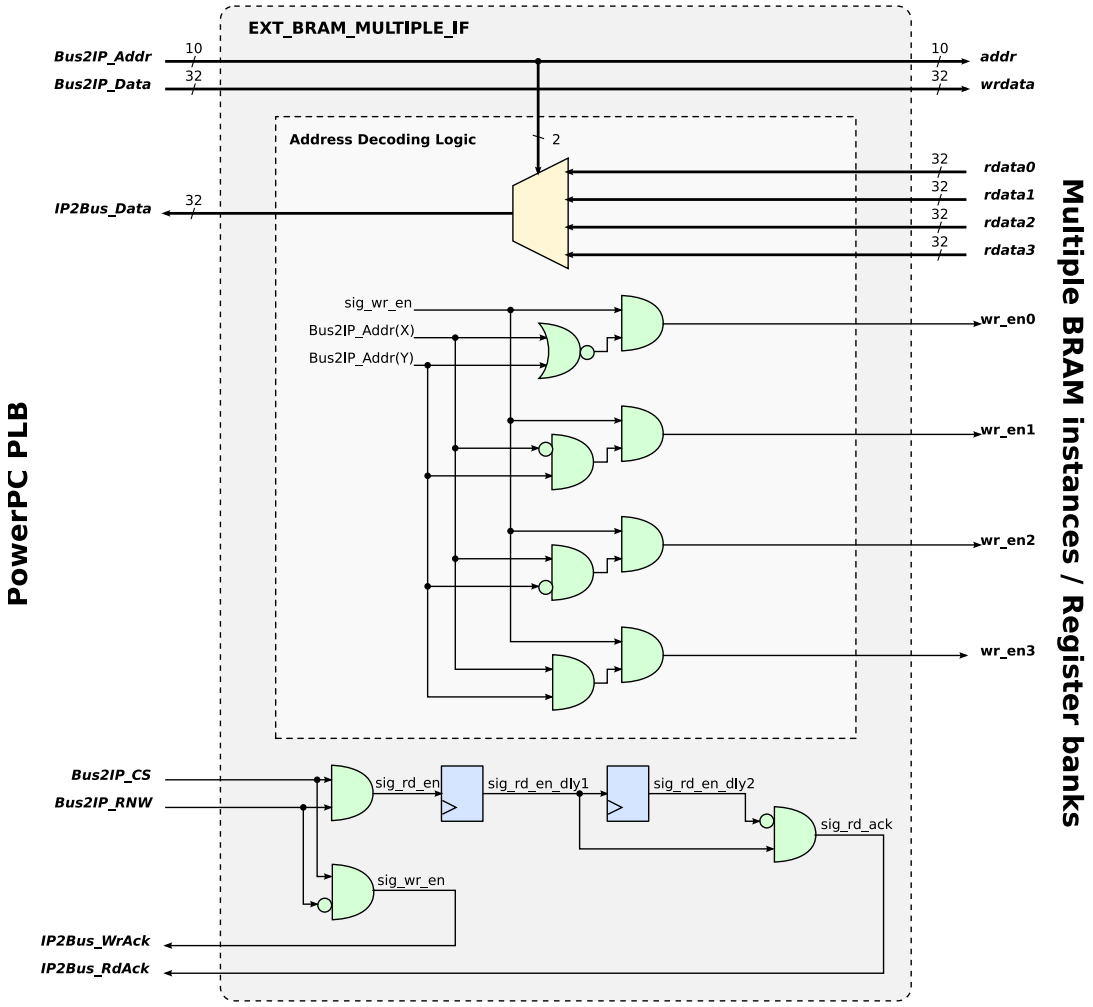


Figure 4.4: Schematic view of the multiple BRAM interface. The substantial difference to the single BRAM interface is the presence of an address decoding logic, which allows to select between the individual BRAM units. The interface provides four separate input data buses (`rdata0..3`) and write enable signals (`wr_en0..3`) for each BRAM connected. The parameterizable widths address, read data and write data bus are given for an 18Kb BRAM entity as an example.

tion of PLB arbiter logic as is the case when dealing with multiple instances of the single BRAM interface. The implementation is similar to the single BRAM interface. In addition, an *address decoding logic* selects between the individual BRAM instances connected. Separate data input ports and write enable signals are available for each of the BRAM entities. The multiple BRAM interface also provides the same generics as listed in table 4.1. Figure 4.4 gives an overview of the data path of the multiple BRAM interface.

SRAM Controller Interface The *XMU*s are each provided with two fast 18 Mb DDR2 SRAM chips, which serve as buffer for event data. The custom-made controller to access the SRAM is attached to the Processor Local Bus via the *SRAM controller interface*.
The SRAM controller[9] uses a number of separate *FIFO*s to buffer both write and read addresses as well as write and read data. In case of a write transaction, write address and data are loaded into corresponding FIFOs when the write enable signal `wen` is asserted. A read operation is implemented in two steps. The read address is loaded into the read address FIFO by asserting the read enable signal `ren`. The controller performs the read operation and stores the data into the read data FIFO, which can be read out by asserting the `qen` signal. The latency of data available from the SRAM is not specified precisely due to several arbitration processes, but is typically 14 clock cycles after initiation of the read access. Thus, the SRAM controller provides a signal to indicate valid read data (`data_valid`), which is derived from the `empty` signal of the read data FIFO.
A schematic overview of the SRAM controller interface is shown in figure 4.5. Read and write enable (`rd_en` and `wr_en`) as well as the write acknowledge signals are derived from the chip-select and read-write signal of the PLB. The read acknowledge signal is obtained from `data_valid` because of the latency of SRAM data. If no valid data is indicated within 128 clock cycles, the PLB master aborts the data request with a timeout error.

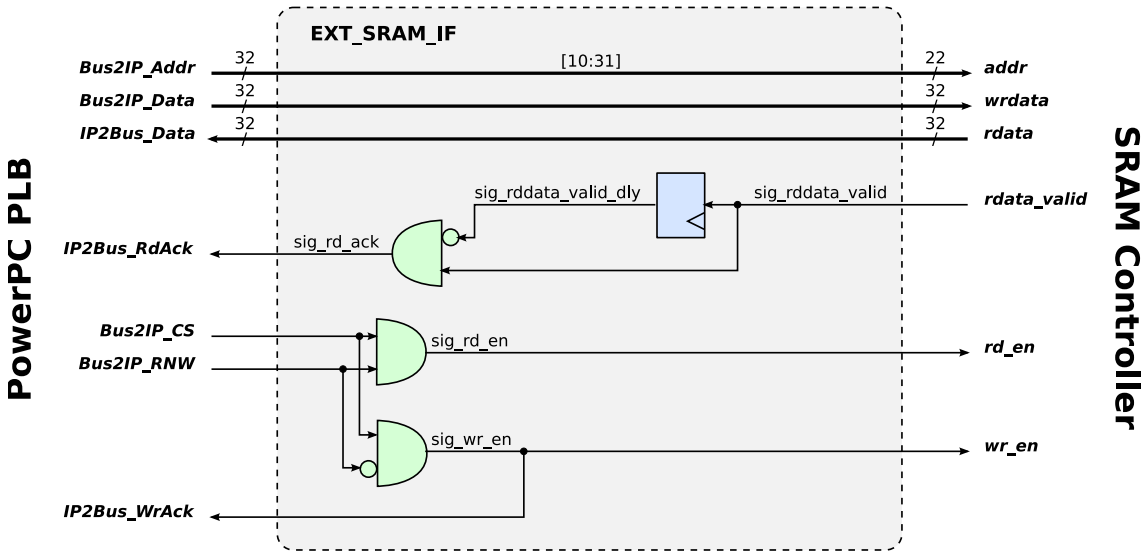


Figure 4.5: The SRAM controller interface connects the SRAM controller to the Embedded PowerPC System. Valid data from the SRAM is indicated by the signal `data_valid`, which is used as read acknowledge signal for the PLB. If this signal is not asserted within 128 clock cycles after initiation of the read operation, the PLB master aborts the read request with a timeout error.

[9] For a detailed description of the controller, please refer to [Ret07].

SD Memory Card Controller Interface During the development of the SD Memory Card Controller described in chapter 6, a PLB interface was required to connect the controller to the PowerPC design and make it accessible to software. Aside from a control register to operate the controller, the interface also provides registers to store a command to be transmitted to the card. An address decoding logic arbitrates the read access to the registers and the BRAM used to buffer card data. Figure 4.6 shows the data path of the interface.

Because the PowerPC 405 processor is based on a 32-bit memory model, the 40-bit card commands sent by the software are split into the 8-bit command index and the 32-bit command argument (see section 6.2), which are transmitted separately over the PLB. Both command index and argument are stored in different registers (`CMD_REG0` and `CMD_REG1`). The two registers are merged and assigned to the command input port of the controller. Another 8-bit register (`CTRL_REG`) combines the control signals used to operate the controller. For detailed information about these registers, please refer to table 4.2 and B.2. Write data and address are directly connected to the interfacing ports of the controller BRAM, which buffers the data exchanged between card and host.
An *address decoding unit* selects the corresponding data source, which is assigned to the PLB input data bus. A 6:1 multiplexer selects between the different command and control registers, the card response registers of the controller or status data. Another 2:1 multiplexer arbitrates between data of the 6:1 multiplexer and BRAM data.

Register	Address (Offset)	Length [Bits]	Description
CMD_REG0	0x00	8	Command index
CMD_REG1	0x04	32	Command argument
RESP_REG0	0x08	8	Higher 8 bits of 40-bit response
RESP_REG1	0x0C	32	Lower 32 bits of 40-bit response
CTRL_REG	0x10	8	Control signals
STAT	0x14	8	Controller and card status signals

Table 4.2: This table lists the registers of SD Memory Card Controller interface along with their size in bits and their offset relative to a base address.

4.3 PowerPC System Build Flow

The Embedded PowerPC System is built with the *Xilinx Embedded Development Kit* (*EDK*), an integrated software solution for designing embedded processing systems for Xilinx FPGAs with embedded PowerPC hard processor cores and/or MicroBlaze soft processor cores. A standard Xilinx EDK project contains a large number of files, which are not all essential to the hardware design or its generation process. Many of them are configuration files generated by the graphical user interface (GUI). However, the EDK hardware build flow is totally independent of the GUI, for it is based on plain text source code and

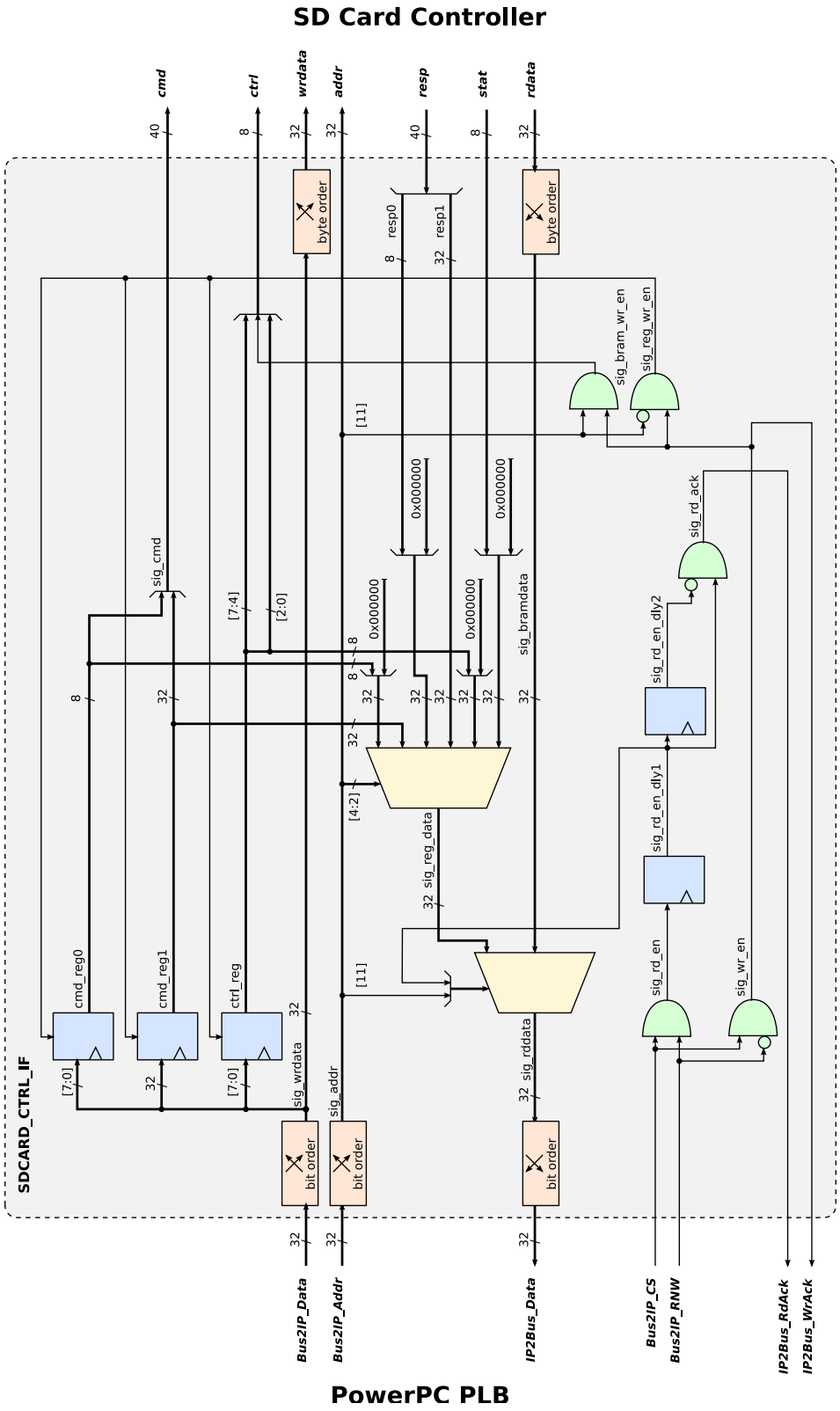


Figure 4.6: The interface shown above connects the SD Memory Card Controller to the Embedded PowerPC System. Signals on the left side are connected to the PowerPC PLB, while the controller is attached to the signals on the right side.

script files (Makefiles). Because the synthesis flow of the *GTU* design is also based on Makefiles, it is desirable to narrow the PowerPC EDK project to its relevant source files and combine both build flows into one single build pass.
Plain text files also benefit from being manageable by a *revision control system*. Actually, the entire *GTU* project tree is under control of such a system, in this case *SVN*[10].
The script-based *GTU* console build flow permits to be automated and executed periodically at a specific time. In combination with the revision control system, it forms the *GTU Nightly Build System*, which provides the most recent hardware design created from the latest source revision.

A minimalistic EDK project consists of the following components:

- A Microprocessor Hardware Specification (MHS file)
- A Microprocessor Software Specification (MSS file)
- A hardware peripheral directory (*pcores/*) (contains MPD, PAO and VHDL/Verilog source files)
- A software driver directory (*drivers/*) (contains MDD, MLD and C source files)
- Two Makefiles (<*project-name*>*.make* and <*project-name*>*_incl.make*)

The MHS file defines the top level of the hardware design on an abstract level. It provides information about the processor and bus architecture as well as peripherals, connectivity and address space segmentation. The components are described by Xilinx-specific *meta-code*, which is a generic layer on top of *VHDL*[11] or *Verilog*[12] and defines generic parameters and port assignments of the individual entities.
The MSS file contains directives for customizing operating systems, software libraries and drivers for the embedded system. Its syntax is similar to that of the MHS file and describes generic parameters to configure the software on an abstract level.
The hardware peripheral directory contains the source code of additional custom peripherals. The code is located in separate subdirectories for each component and is a combination of both abstract meta-code (MDD and PAO) and VHDL/Verilog sources. The MDD (Microprocessor Peripheral Definition) file defines the interface of the peripheral, while the PAO (Peripheral Analyze Order) file contains a list of HDL files, which are needed for the synthesis, and defines the analyze order for compilation.
The source code of the software drivers for the custom peripherals is located in the individual subdirectories of the software driver directory. While the MDD (Microprocessor Driver Definition) and MLD (Microprocessor Library Definition) files contain directives for customizing the software drivers and libraries, respectively, the actual driver sources

[10] SVN: Subversion. See *http://subversion.tigris.org*.
[11] VHDL: Very High Speed Integrated Circuit Hardware Description Language. Commonly used as a design-entry language for FPGAs and ASICs (Application-Specific Integrated Circuits) in electronic design automation of digital circuits.
[12] Another common hardware description language.

are available in plain C.
Last, the Makefiles are supplied with the necessary instructions to generate the *design netlists* for the target platform. A detailed description of the EDK project structure and syntax of the meta-code files is given in [Xil07a] and [Xil07b].

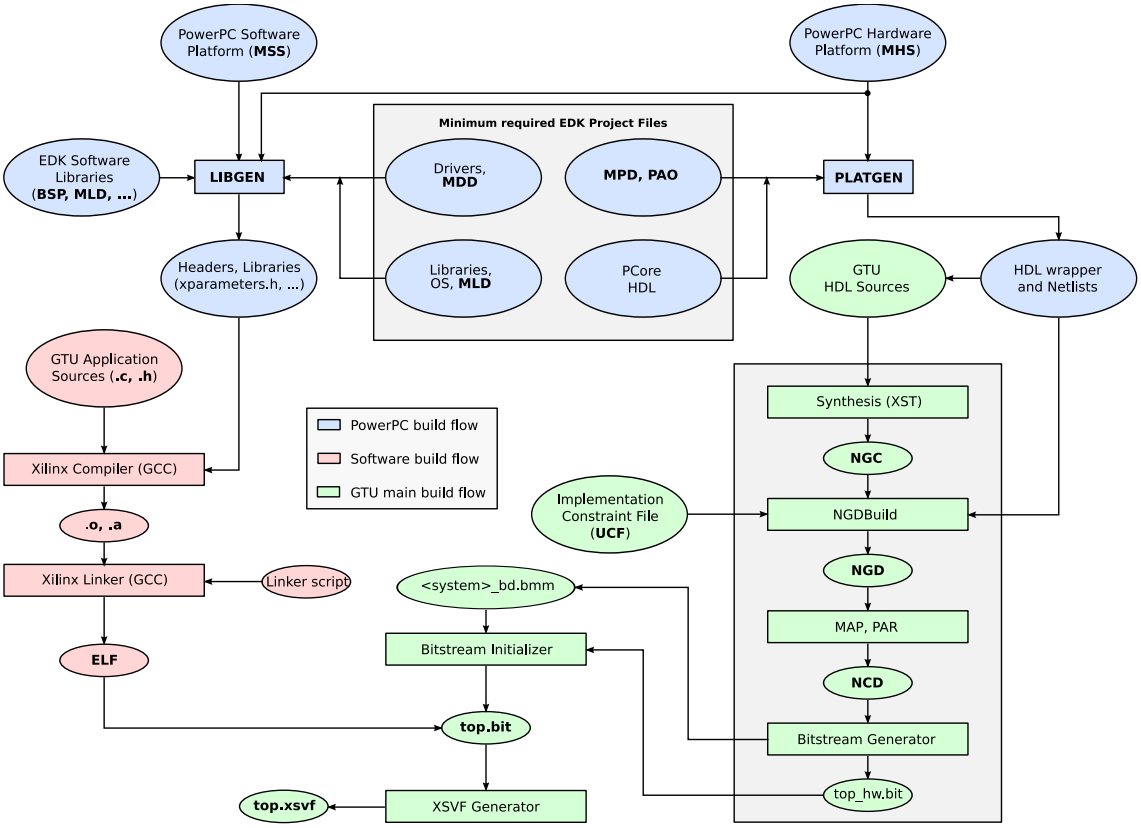


Figure 4.7: Global build flow of the *GTU* design. The three individual build parts are represented by corresponding Makefiles.

Figure 4.7 illustrates the global *GTU* design build flow. It is divided into three parts: the Embedded PowerPC System build flow (blue), the *GTU* hardware synthesis (green) and the system software generation process (red). Each of them is represented by a corresponding Makefile. However, the different build parts are not independent from each other. The PowerPC netlists are required by the *GTU* hardware design, while the *GTU* system software depends on the software headers and libraries generated by the EDK software build flow. In a final step, the software bitstream is integrated into the FPGA configuration file to initialize the boot memory (BRAM) of the Embedded PowerPC System with the software when the FPGA is programmed with the resulting XSVF file (see section 3.2).

4.4 Test Setup and Results

The test setup, which has been used to verify the functionality of the Embedded PowerPC System consists of a crate similar to the ones installed in the *ALICE* pit and one *TMU* board. A personal computer interfaces with both the JTAG and the UART connector of the test board to program the FPGA and establish communication with the Embedded PowerPC System. The PowerPC design is integrated into and tested with all three *XMU* FPGA designs.

The DDR2 SDRAM Controller The DDR2 SDRAM controller has been tested extensively to ensure a reliable access to the memory. The test software continuously writes pseudo-random test data to the entire memory and counts the number of bit errors detected when the readout is performed. It also measures the time to completely write and read the data. Table 4.3 summarizes the test results. During the test period of about $ttw_{tot} + ttr_{tot} = 4 \cdot 10^5 s$, a data volume of $dv_{tot} \approx 1{,}529\,\text{GB}$ has been transferred between controller and DDR2 SDRAM. This corresponds to an average data transfer rate of $dr_w = \frac{1}{2} \cdot \frac{dv_{tot}}{ttw_{tot}} = 3.66\,\text{MB/s}$ (write access) and $dr_r = \frac{1}{2} \cdot \frac{dv_{tot}}{ttr_{tot}} = 4.02\,\text{MB/s}$ (read access). The tests have indicated no errors yielding a bit error rate $BER \leq \frac{1}{dv_{tot}} = 0.76 \cdot 10^{-13}$, which is about industry standard. The very data transfer rates probably result from the test method and underestimate the actual performance of the controller. The test software measures reads and writes of single data words, which implies a large overhead when calling the access and timer routines.

$\mathbf{dv_{tot}}$ [GB]	$\mathbf{ttw_{tot}}$ [s]	$\mathbf{ttr_{tot}}$ [s]	$\mathbf{dr_w}$ [MB/s]	$\mathbf{dr_r}$ [MB/s]	**BER**
1,529	$2.14 \cdot 10^5$	$1.95 \cdot 10^5$	3.66	4.02	$< 0.76 \cdot 10^{-13}$

Table 4.3: Test results of DDR2 SDRAM controller. $\mathbf{dv_{tot}}$ represents the total data volume transferred between the processor and the memory. $\mathbf{ttw_{tot}}$ and $\mathbf{ttr_{tot}}$ are the total time to write and read $\mathbf{dv_{tot}}$ **/ 2**, respectively. The average data transfer rate of write and read access is given by $\mathbf{dr_w}$ and $\mathbf{dr_r}$, respectively. Finally, **BER** represents the bit error rate calculated from $BER \leq \frac{N_{err}}{dv_{tot}[bit]}$, where $N_{err} < 1$ is the number of bit errors.

4.5 Status and Future Prospects

The Embedded PowerPC System at its present stage offers the capability to control and administrate the *GTU* and its system components, which are attached to the Processor Local Bus via several custom-designed interfaces. It implements a memory controller (MPMC), which allows to access the DDR2 SDRAM chip and use it as data and instruction memory for the PowerPC processor core. The interface for the SD Memory Card Controller developed for this thesis and described in chapter 6 allows software to access

the SD Memory Cards installed. Still, some enhancements and extensions can be applied to the embedded design to improve its performance and capabilities.

The DCS boards currently represent the interface to communicate with the *XMU*s via the UART soft core implemented into the DCS FPGA design. Each DCS UART soft core is shared by the six *GTU* boards of a segment. However, the serial communication protocol in use allows data transfer at low rates of 56 Kb only. On the other hand, the Virtex-4 FX100 FPGAs are supplied with *Multi-Gigabit Transceivers* (MGTs), which can be used for *Gigabit Ethernet* purposes. In combination with the MGTs and the Xilinx *TEMAC* core, the SFP modules (see chapter 3.) could serve as *PHY*s[13] and provide Gigabit Ethernet using the *1000BASE-X* standard for optical fibers. However, only *SMU*s and the *TGU* have SFP modules reserved for networking purposes. To connect the *TMU*s to the network as well, it would be necessary to use the *SMU*s as switches, which establish point-to-point connections to the *TMU*s via the common backplane.

Activation of the second PowerPC processor provided by the Virtex-4 FX100 devices would increase the computing power of the Embedded PowerPC System and offer new abilities to administrate the *GTU* while simultaneously performing time-critical tasks. While one CPU of each FPGA would be responsible for monitoring and controlling purposes only, the other one could concentrate on time-critical statistics on event or trigger rates, respectively. However, some of the peripheral components (e. g. main memory and communication interface) would have to be shared by both processors. Thus, a number of precautions would have to be taken in order to avoid access conflicts.

The Embedded PowerPC System as presented has become an inherent part of the *GTU* design and is already successfully in use at *CERN*. Its major task is to provide the hardware infrastructure to monitor and control the individual system components of the *GTU*. Although there are some extensions to improve the performance and capabilities for future applications, the system is ready-to-use as-is.

[13] PHY: Physical layer of the *OSI model*. A PHY connects a link layer device to a physical medium (optical fiber or copper cable).

5 The Embedded Linux System

At present, the *GTU* is controlled and monitored by standalone system software, which runs on the Embedded PowerPC System of each *XMU*. Via an UART core, the DCS boards provide the interface to communicate with this software (see section 3.2). The *GTU* system software offers various commands to perform configuration and monitoring tasks such as setting configuration parameters of *GTU* system components or reading out temperatures and supply voltages.
On the other hand, the Embedded PowerPC System as described in the previous chapter is an ideal platform not only to run task-specific control software, but it is powerful enough to host an *operating system* like Linux, which offers a number of advantages over a task-specific system software.
The crucial advantage of an operating system like Linux over a task-specific system software is the capability of *multi-threading*. While the current *GTU* software can execute only one administration command at a time, a multi-threading environment permits to simultaneously perform various monitoring and controlling tasks.
In contrast to the *GTU* system software, a Linux kernel is provided with a TCP/IP stack, which is essential for networking purposes. With the existing Xilinx device driver for the TEMAC[1], the fundamental requirements for a high-speed Ethernet connection via the Multi-Gigabit Transceivers (MGTs) of the Virtex-4 FX100 FPGA are fulfilled. The Multi-Gigabit Ethernet connection is intended to replace the low-performance UART communication (see section 4.5).
Each *XMU* has a slot for an SD Memory Card in order to provide the boards with mass-storage media. The cards are formatted with a *FAT32 file system*, which can be accessed through a corresponding device driver provided by the Linux kernel. However, the *GTU* system software cannot benefit from the file system, since the cards can currently be accessed in raw data mode only. The necessary I/O routines to access the file system properly still need to be integrated manually.

Considering the preceding discussion, it is desirable to supply the *GTU* with an operating system rather than a task-specific system software. Linux has established itself as a standard platform for embedded purposes and represents an appropriate choice in order to reach more flexibility regarding the development of custom device drivers and administration features. While system commands are integrated as inherent part of the *GTU*

[1] Xilinx Tri-Mode Ethernet Media Access Controller. See chapter 4.

system software and thus require re-booting of the PowerPC processor when changed, under Linux control software can be changed in a straightforward manner.

In this chapter, the development of an Embedded Linux System for the *GTU* is described. The following section discusses the advantages and disadvantages of current Embedded Linux distributions. Subsequently, the process of configuring the kernel as well as building and testing the Embedded Linux System is illustrated. The last section gives an overview of the current status and an outlook to future extensions.

5.1 Embedded Linux Distributions

When choosing a suitable Linux distribution for the *GTU*, several requirements have to be taken into account. The kernel provided has to be compliant with the PowerPC 405 processor architecture and support vendor-specific hardware design components like the Xilinx UART core or the TEMAC for Ethernet purposes. The kernel configuration should offer a mechanism to extract and integrate system parameters of customized hardware designs in order to avoid modifying the kernel sources manually. Furthermore, the resulting Embedded Linux System should also provide a complete *root file system* supplied with system tools and *userland*[2] software.

The main focus of this diploma thesis is to develop an Embedded Linux System, which suits the demands stated above. Several Linux projects[3] and *mailing lists*[4] have already attended to port Linux on embedded PowerPC processors. A diploma thesis about a basic approach written by A. Sinsel concentrates on fundamental preparations and modifications of the kernel sources for this purpose [Sin03]. Consulting these sources, an extensive research has limited the large variety of Embedded Linux distributions to the following four candidates:

- Linux From Scratch (*LFS*) (*http://www.linuxfromscratch.org*)
- uClinux (*http://www.uclinux.org*)
- MontaVista Linux (*http://www.mvista.com*)
- Petalinux (*http://www.petalogix.com*)

Aside from LFS, which is built from the very bottom and can be applied to almost any system architecture, the other three distributions are especially designed for embedded purposes. Unnecessary features have been removed in order to narrow the kernel, since embedded systems mostly have limited resources. The following paragraphs describe different aspects of those four distributions.

[2] Operating system software, which does not belong to the kernel.
[3] For a detailed list please refer to *http://www.penguinppc.org*.
[4] linuxppc-embedded@ozlabs.org, linuxppc-dev@ozlabs.org

Linux From Scratch Building a *Linux From Scratch*[5] starts from the very bottom, because the system is created step-by-step, entirely from source code. The the big advantage is that the emerging system is adapted precisely to the needs of the user, resulting in a very tiny kernel.
On the other hand, building a Linux system from the very bottom is a rather time-consuming task. Typically, it starts with building an appropriate *toolchain*, a set of programming tools required to compile software source code [Yag03]. However, there are several sets of tools available, which allow to easily generate a toolchain for the target processor architecture that can be used alternatively[6].
Once the buildchain is created, the Linux system is built in several passes. First, a temporary Linux environment is set up, which contains the essential tools to build the final LFS system. Based on this environment, the kernel can be successively configured and compiled. Finally, the necessary system and user applications are installed on the root file system.
Regarding the present Embedded PowerPC System, the kernel source code has to be modified manually in order to integrate the configuration settings of the hardware design. Because a standard Linux kernel does not support Xilinx-specific peripheral components, the necessary device drivers have to be developed for the kernel as well.

uClinux *uClinux* is a Linux distribution particularly designed for *microcontrollers* and microprocessors without a *Memory Management Unit* (*MMU*). It provides both kernel version 2.4 and 2.6 and comes with a selection of software applications (including the GNU Core Utilities), which can be integrated into the kernel root file system according to requirements. The standard C library *glibc* is replaced by the much smaller *uClibc*[7], which minimizes the usage of memory resources.
Aside from several processor architectures including the PowerPC 405 family, uClinux also supports various Xilinx development boards and other embedded platforms. Furthermore, the distribution offers a mechanism to extract relevant system information from custom hardware designs and integrate it into the kernel configuration. This *autoconfig* mechanism[8] allows to fit the kernel to custom hardware in a very flexible way. However, it is available for hardware designs based on the *Xilinx MicroBlaze soft processor core*[9] only.

MontaVista Linux *MontaVista Linux* is a commercial Linux distribution based on kernel version 2.6, which offers full support for *uClibc* and comes with a large number of user-

[5] To be exact, LFS is not a Linux distribution. The name refers to the process how the system is built.
[6] See *http://www.kegel.com/crosstool* or *http://www.buildroot.org*.
[7] uClibc is compatible to glibc and primarily designed to handle the memory management for systems without an MMU, but can be used for systems provided with an MMU either.
[8] The parameters of the hardware design are extracted and stored in a file called *auto-config.in*, which is included by the kernel build flow.
[9] *http://www.xilinx.com/microblaze*

land applications. In addition, it provides a platform and application development kit, which are both fully graphical integrated development environments.
MontaVista Linux supports different processor architectures like *ARM*, *X86* or *PowerPC* as well as various development platforms of Xilinx and other vendors. However, system parameters of custom-developed hardware have to be integrated manually into the kernel configuration.
There is a large overhead due to the project files of the graphical development environment, which are not directly related to the actual kernel sources. Since the *GTU* project tree is under control of a revision control system, it is desirable to integrate the kernel sources into the repository as well. In contrast to the plain text kernel build flow of typical Linux distributions, the MontaVista Linux build flow is hardly suitable for being controlled by a *revision control system*.

Petalinux *Petalinux* is specifically developed for usage with Xilinx FPGAs and particularly supports the Xilinx MicroBlaze soft processor core. It is derived from uClinux and comes with both kernel version 2.4 and 2.6 as well as the set of uClinux userland software.
Apart from MicroBlaze and PowerPC, no further processor architectures are supported by the distribution. However, the provided PowerPC kernel is not operational, and the development of the Petalinux PowerPC kernel has been suspended for the time being. In addition to the support for various development platforms of different vendors, Petalinux also provides the uClinux *auto-config* mechanism for customized MicroBlaze platforms.

Considering the demands stated at the beginning of this section, none of the distributions previously discussed seems to be a suitable solution on its own. Building a Linux bottom-up is unreasonable, because any modification on the design requires to adapt the kernel sources manually. Furthermore, vendor-specific hardware components are not supported by corresponding kernel device drivers. On the other hand, even Linux distributions particularly developed for embedded purposes and providing support for Xilinx platforms are unusable with custom-developed PowerPC designs. However, the auto-config mechanism provided by both uClinux and Petalinux represents a basic approach, which can be extended to build an Embedded Linux System for embedded PowerPC designs.

The decision was made in favor of Petalinux to serve as the basic development platform, because it provides both the auto-config mechanism and is particularly developed to work with Xilinx FPGAs. The uClinux-based PowerPC kernel provided is replaced by a regular, but modified MontaVista PowerPC kernel. Furthermore, several patches are applied to the kernel as well.

5.2 Building the Embedded Linux System

A promising way to implement the auto-config mechanism for PowerPC processors as well has been described by *J. Williams*, architect and maintainer of the port of uClinux to the Xilinx MicroBlaze soft processor core [Wil06]. The instruction guide explains the steps necessary to prepare a uClinux distribution to support custom PowerPC platforms and integrate the required kernel sources into the build flow. As a uClinux derivative, Petalinux can be modified in the same way. Within the scope of this diploma thesis, *Petalinux v0.20-rc3* has been chosen as the reference.
To set up the Petalinux environment accordingly, the PowerPC kernel provided by the Petalinux distribution has to be replaced by a regular, but modified kernel version 2.4.29[10]. The kernel modifications were carried out by *MontaVista* and concentrate on design-specific source code referring the PowerPC 405 family in order to get a first Linux kernel running on an *IBM PPC 405GB* reference platform[11] . Furthermore, Petalinux lacks a PowerPC buildchain to *cross-compile* for this processor. Thus, the toolchain proposed by J. Williams is used. Support for custom hardware designs is provided by adding a new entry in the Petalinux *vendor tree* (*powerpc-auto*). It is the equivalent of the *petalinux-auto* platform used for MicroBlaze systems and enables the auto-config mechanism for the PowerPC in the first place.

The proceeding subsections give an overview of how to configure the PowerPC hardware project and kernel settings to build a Linux kernel image for the Embedded PowerPC System.

5.2.1 Petalinux and Xilinx Embedded Development Kit

The auto-config mechanism links the hardware synthesis flow of the *Xilinx Embedded Development Kit* (*EDK*) with the Linux kernel build flow. As shown in figure 5.1, relevant data from the *Microprocessor Software Specification* (*MSS*) file is extracted and merged into the *auto-config.in* file. This file is included into the kernel configuration and represents the actual connection between both build processes. To enable the auto-config mechanism for the hardware design, the project files *powerpc.mss* and *powerpc_incl.make* have to be adapted accordingly. A description of the necessary modifications is given in appendix A.
When the synthesis of the Embedded PowerPC System is completed, *auto-config.in* is available in the target output directory specified in *powerpc.mss* and has to be copied to the corresponding directory in the Petalinux vendor tree [Wil06].

[10] Actually, in the case of this diploma thesis kernel version 2.4.30-pre1 has been used.
[11] More precise information about the kernel modifications are given in [Sin03].

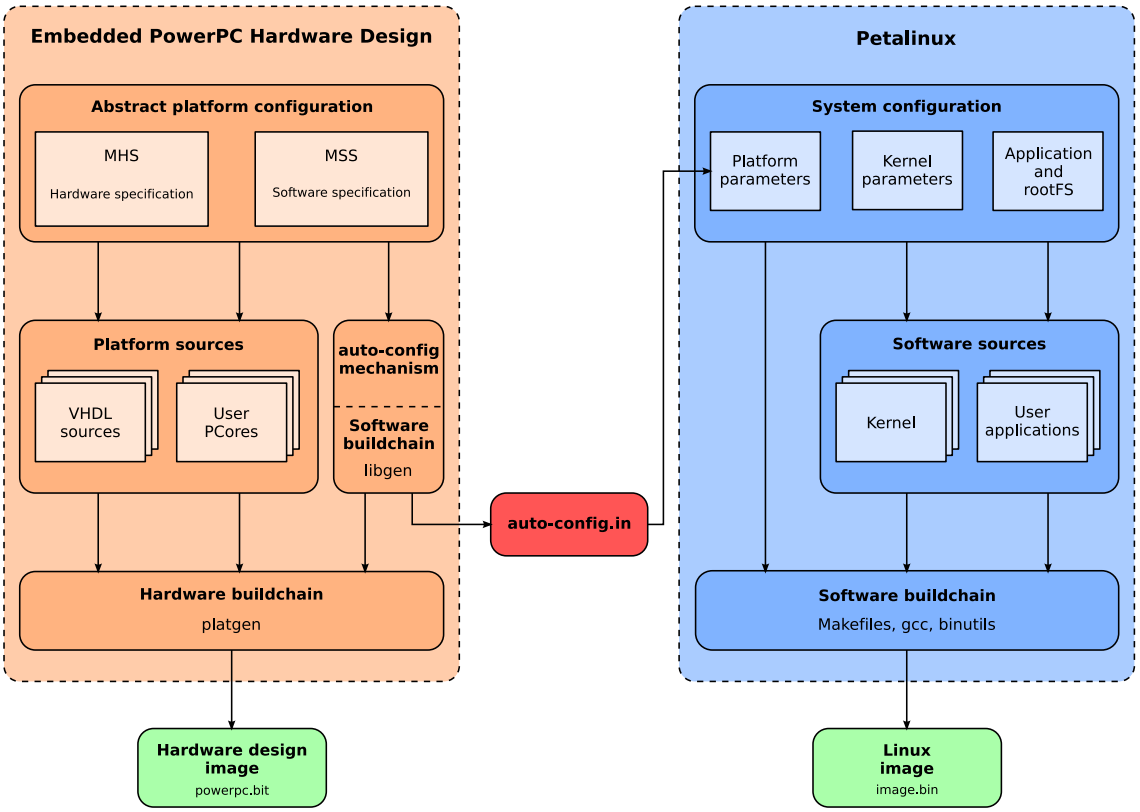


Figure 5.1: The auto-config mechanism of Petalinux forms the connection between Xilinx EDK and Petalinux. Relevant hardware system parameters are extracted from the design and stored in the configuration file *auto-config.in*, which is included by the kernel configuration.

5.2.2 The Petalinux Toolchain and Build Flow

A *toolchain* (also: *buildchain*) is the set of tools, which is necessary to build software from its source files. It consists of several programs to *compile*, *assemble* and *link* the source code. Typically, these are the *GNU C compiler*, the *GNU binary utilities*[12] and the *GNU C library*.

The Toolchain of Petalinux Usually, software is developed on a host platform with a processor architecture corresponding to that of the target platform, which the software is applied to. However, when dealing with embedded systems, host and target system architecture will differ in most instances. Thus, a *cross-compiler* is needed, which is capable to generate executable code for a processor architecture other than the one which the compiler runs on.

[12] Aside from the *GNU assembler* and *GNU linker*, the binary utilities provide several more tools for manipulation of object code generated by the compiler.

Although it is possible to manually create a cross-compilation buildchain bottom-up [Yag03], a more fail-safe and less time-consuming alternative is to use a prebuilt toolchain. For the development of the present Embedded Linux System, the buildchain recommended in the instruction guide of J. Williams has been used. It was created with *buildroot*, a set of several Makefiles and patches, which allow to easily generate a cross-compilation toolchain and a root file system for the target Linux system using the uClibc library. The toolchain consists of the *GNU gcc 3.4.2*, the *GNU binutils 2.16.1* and *genromfs 0.5.1*[13], which allows to create a *ROMFS*[14] root file system from a dedicated directory tree.

The Petalinux Build Flow Before the compilation of the sources can be started, it is necessary to configure both kernel and root file system properly. Configuration is started with the command **make menuconfig**. First, general specifications regarding the system environment such as information about the hardware design (vendor and target platform), the desired kernel version (if there is more than one available) and the standard C library to be used (usually uClibc) must be given.
When the system environment has been specified, the actual kernel configuration menu is entered. Depending on the system platform, different configuration options are provided to fit the kernel to the target hardware. For a basic working kernel, it is necessary to specify the processor type and the communication interface. Furthermore, a device driver to access the storage device, which hosts the root file system, and support for the file system type of the root file system must be enabled.
The last configuration menu allows to select the userland applications, which the root file system will be provided with. While core applications like a *TTY daemon*[15] and a *shell* are mandatory, the GNU Core Utilities provide tools for other purposes like networking or file system management and can be added as required.

The configuration settings are stored in three separate files:

- General specifications: *<petalinux_dir>/software/petalinux-dist/.config*
- Kernel configuration: *<petalinux_dir>/software/petalinux-dist/linux-2.4.x/.config*
- Userland configuration: *<petalinux_dir>/software/petalinux-dist/config/.config*

After configuration, the kernel sources can be compiled resulting in the binary Linux image file, which combines both kernel and root file system. This image is ready-to-use and can be executed on the target platform.

[13] *http://romfs.sourceforge.net*
[14] ROMFS is a simple read-only file system often used for embedded purposes, when the root file system has to be placed within the memory. For more elaborate information please refer to *http://lxr.linux.no/linux/Documentation/filesystems/romfs.txt*.
[15] A TTY daemon manages physical or virtual *terminals* and provides user interaction with the system console.

5.2.3 Configuration of Kernel and Root File System

The kernel of the Embedded Linux System for the *GTU* only implements device drivers for both communication via the UART interface and accessing the root file system. For embedded systems, a binary image of the root file system is usually appended to the kernel image. During boot stage, the root file system image, located on the boot medium is copied to a *ramdisk*, which is located in main memory. The kernel accesses the root file system of the ramdisk.

Ramdisk Embedded systems are often not provided with non-volatile storage media such as hard disk drives. Instead, they use memory devices based on *flash technology* to store the images of both kernel and root file system. However, flash chips have some significant limitations. Generally, the access time of flash chips is higher compared to RAM chips, particularly regarding write operations. A write operation on flash memory implies both an erase and a write cycle of a memory block[16] at once. Erasing a data block before writing is necessary to reset all bits to one, because a write operation allows to set corresponding bits to zero only.
The most inconvenient limitation of flash memory is the finite number of cycles a memory block can be erased and written, which for current devices is of the order of 100,000. Extensive write operations as performed by the kernel when operating on the root file system would soon damage the flash chip, if no precautions are taken to prevent a block from being frequently accessed[17].
Thus, the kernel does not operate on the root file system stored on the flash memory, but on a copy created during the boot stage and located in main memory. This specific region of memory, which serves as block-oriented storage device, is referred to as *ramdisk*. A ramdisk is accessed through a special *ramdisk driver* provided by the kernel and can be formatted with a file system as any other storage medium.

Kernel Configuration The following settings specify the PowerPC 405 processor core and map the Linux system console to the UART interface in order to provide communication with the system:

- ***Platform support***
  - (40x) Processor Type
  - TTYS0 device and default console

[16] Typically 32 KB or 128 KB.

[17] Common methods are *wear leveling* and *bad block management* (BBM). While wear leveling counts the writes and dynamically remaps blocks in order to spread write operations between sectors, BBM performs write verification and remapping to spare sectors in case of write failure.

- ***Character devices***
  - Xilinx UART Lite
  - Console on UART Lite port

The ramdisk required for the root file system is created and accessed by using the *Memory Technology Device* (MTD) drivers[18]. Furthermore, support for the file system type of the root file system must be enabled, which is ROMFS in this case.

- ***Memory Technology Devices (MTD)***
  - Memory Technology Device (MTD) support
  - MTD partitioning support
  - Caching block device access to MTD devices
  - *RAM/ROM/Flash chip drivers*
    - Support for RAM chips in bus mapping
  - *Mapping drivers for chip access*
    - Generic uClinux RAM/ROM filesystem support
- ***File system***
  - ROM file system support

Although the Embedded PowerPC System is not provided with networking hardware at present, **Networking support** in the submenu ***General setup*** must be activated. Some kernel sources rely on these settings and cannot be compiled otherwise. For the same reasons, it is necessary to enable the options **Packet socket** and **TCP/IP networking** of submenu ***Networking options*** as well, while all other items of this submenu must be deactivated to avoid errors during the build process. However, ***Networking device support*** is not required and can be disabled. Other kernel configuration settings can be left as default.

Root File System Configuration The **Root filesystem type** is specified in the submenu ***System settings*** and must be set to ROMFS. Mandatory *core applications*, which have to be integrated into the root file system, are *init*, *gettyd* and a *shell* (bash). Other **Miscellaneous applications** (GNU Core Utilities) can be added as required. Although there is a *busybox* available as well, it is not recommended to make use of it, since its sources are flawed and cause the build flow to aborts with a large number of errors.

After configuring the root file system, the compilation of the kernel sources can be started with the following commands:

[18] Originally developed to provide access to flash memory, the MTD drivers can be used for RAM and similar chips as well.

- **make clean** ensures a proper clean-up of the development tree to avoid misconfigured remainings of former build processes. In case of a modification of the *autoconfig.in* hardware parameter file, it is necessary to invoke **make distclean** instead, since this file is read only once at the initial configuration pass of a clean environment. However, this command also deletes the configuration files of **make menuconfig** (see subsection 5.2.2).
- **make dep** evaluates the kernel dependencies.
- **make** finally starts the actual compilation process of kernel and applications.

Kernel Patches Since several source files of the Petalinux environment are flawed, a number of patches have to be applied, which were created during the development of the Embedded Linux System. A full list including a brief description of the patches is given in appendix, section A.2. The patch sources are located in the directory */petalinux-v0.20-rc3/patches* of the DVD, which is attached with this diploma thesis.

After applying the patches, the build flow can be executed without errors. The resulting kernel image is stored in the following directory:

<petalinux_dir>/software/petalinux-dist/images/

It contains the same kernel image in different file formats. *linux.bin* is the pure binary kernel image resulting from the kernel build process. A binary image of the root file system including the system and userland applications is stored in the file *romfs.img*. These two files are concatenated to *image.bin* and *image.elf*, an *Executable Linux File* (ELF). ELFs are preferred for debugging purposes, because they contain additional information about the code sections, while *image.bin* is suited to be directly copied into memory and executed. The size of *image.bin* corresponds to the size of memory used by the kernel and is about 2.7 MB.

5.3 Test Results and Status

The setup described in section 4.4 has been used in order to test the generated Linux kernel. It has been tested on different *GTU* boards and with all three designs, which implement the Embedded PowerPC System described in the previous chapter. The Xilinx Microprocessor Debugger (XMD) is used to connect to the JTAG interface of the PowerPC processors via the JTAG controller shown in figure 4.2. It provides commands to load the kernel image file into the DDR2 memory and to execute and control the program flow. The necessary command sequence is listed below.

- **connect ppc hw -debugdevice devicenr 2 cpunr 1** connects to the JTAG controller of the Embedded PowerPC System. Since the JTAG chain contains two devices (XCF32P PROM and Virtex-4 FX100 FPGA), the FPGA must be selected by the parameter `devicenr 2`. The parameter `cpunr 1` is required to address the first PowerPC 405 processor core of the FPGA device.
- **dow image.elf** loads the kernel image into the DDR2 memory and sets the Program Counter (PC) of the processor to the kernel entry point.
- **con** finally starts execution from the actual PC address.

5.3.1 Results

Figure 5.2 shows the output of the system console during the boot stage. Information about the initial memory allocation is given in lines one to five. The two columns specify the physical start and end address, respectively. The messages up to line nine result from the boot code, which uncompresses the Linux kernel, while the actual kernel boot stage begins at line ten. Subsequently, various information about the kernel and the system is given. The modified MontaVista PowerPC kernel used in this system is of version 2.4.30-pre1 and was originally ported to run on Xilinx Virtex-II Pro FPGAs (line 10 and 11). Line 19 gives information about the current memory usage. The output messages of ramdisk creation and initialization is shown in lines 31 to 35. After mounting the root file system (line 41 to 45), the *init* process is entered and the shell is spawned (line 50 and 51).
In order to perform a simple test of the runtime-stability of the Embedded Linux System under long-term conditions, an infinite loop has been run, which executes the *uptime* command to display the system uptime. At the present time, the system is running stable for more than 80 days.

5.3.2 Problems Encountered and Solutions

During the development of this Embedded Linux System, a major problem has been encountered, which seriously affected the runtime-stability of the Linux kernel. The uptime of the operating system varied from a few seconds to several hours.

The Embedded PowerPC System, which was used to test a first working version of the Linux kernel, differs from the current hardware design. Most of the peripheral components[19] listed in chapter 4 were implemented differently to attach them to the *On-Chip Peripheral Bus* (*OPB*)[20]. The OPB is intended to access low-speed and low-performance system resources. A "PLB to OPB" bridge is required to access the OPB peripherals.

[19] All except the MPMC and the Multiple BRAM Interface, which have been added during the development of the Embedded Linux System.
[20] The OPB has been declared obsolete by Xilinx. However, it is still supported by the development tools in order to provide compatibility with older designs.

```
[00]
[01]  loaded at:     00400000 005341E0
[02]  board data at: 00531138 00531150
[03]  relocated to:  004052BC 004052D4
[04]  zimage at:     00405885 005308E8
[05]  avail ram:     00535000 04000000
[06]
[07]  Linux/PPC load:
[08]  Uncompressing Linux...done.
[09]  Now booting the kernel
[10]  Linux version 2.4.30-pre1 (gerlach@gtudev) (gcc version 3.4.2) #68 Thu Feb 14 17:55:46 CET 2008
[11]  Xilinx Virtex-II Pro port (C) 2002 MontaVista Software, Inc. (source@mvista.com)
[12]  On node 0 total pages: 16384
[13]  zone(0): 16384 pages.
[14]  zone(1): 0 pages.
[15]  zone(2): 0 pages.
[16]  Kernel command line:
[17]  Xilinx INTC #0 at 0x41200000 mapped to 0xFDFFF000
[18]  Calibration delay loop... 398.95 BogoMIPS
[19]  Memory: 61928k available (988k kernel code, 1800k data, 44k init, 0k highmem)
[20]  Dentry cache hash table entries: 8192 (order: 4, 65536 bytes)
[21]  Inode cache hash table entries: 4096 (order: 3, 32768 bytes)
[22]  Mount cache hash table entries: 512 (order: 0, 4096 bytes)
[23]  Buffer cache has table entries: 4096 (order: 2, 16484 bytes)
[24]  Page-cache hash table entries: 16384 (order: 4, 65536 bytes)
[25]  POSIX conformance testing by UNIFIX
[26]  LinuxNET4.0 for Linux 2.4
[27]  Based upon Swansea University Computer Society NET3.039
[28]  Initializing RT netlink socket
[29]  Starting kswapd
[30]  pty: 256 Unix98 ptys configured
[31]  RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
[32]  uclinux[mtd]: RAM probe address=0xc0150f98 size=0x173000
[33]  Creating 1 MTD partitions on "RAM":
[34]  0x00000000-0x00173000 : "ROMfs"
[35]  uclinux[mtd]: set ROMfs to be root filesystem
[36]  NET4: Linux TCP/IP 1.0 for NET4.0
[37]  IP protocols: ICMP, UDP, TCP
[38]  IP: routing cache hash table of 512 buckets, 4Kbytes
[39]  TCP: Hash tables configured (established 4096 bind 8192)
[40]  NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
[41]  VFS: Mounted root (romfs filesystem) readonly.
[42]  Freeing unused kernel memory: 44k init
[43]  Mounting proc:
[44]  Mounting var:
[45]  Populating /var:
[46]  Running local start scripts.
[47]  Setting hostname:
[48]  Setting up interface lo:
[49]  /etc/rc.d/S40network: ifconfig: command not found
[50]  init: Booting to single user mode
[51]  # _
[52]
```

Figure 5.2: Output of the Linux system console during the boot stage of the Embedded Linux System. The kernel has been tested with all three *XMU* designs on different *GTU* boards and shows no errors or warning messages at boot time. The line numbers have been added manually.

Because the system crashed virtually in a statistical manner, first assumptions focussed on an unreliable DRAM controller. A number of RAM tests were run to verify the functionality of the controller. The tests included linear and random access and were applied to randomly chosen memory regions of different size as well as to the entire memory. The refresh cycle logic of the controller was also tested. However, no errors were detected.
The Linux kernel was examined systematically in order to try to trace back the problem where it first occurs. The kernel never crashed during the boot process, and extensive tests yielded that the system becomes instable when the *scheduling mechanism* of the kernel is started. However, the cause of the kernel instability could not be determined further, because the kernel machine code would have been to be debugged in detail.
Since both uClinux and Petalinux are known to run stable on several Xilinx development platforms, an approach from the opposite direction was chosen. The Linux kernel described in this chapter was stable on a reference platform board *ML403*[21]. However, an *XMU* design ported to the ML403 caused the kernel to crash. The test results showed that the kernel instability was triggered by the *XMU* hardware design.
The major difference between the Xilinx reference design and the former revision of the Embedded PowerPC System is the memory controller to access the DDR2 SDRAM. While the reference design uses a DDR2 SDRAM controller attached to the PLB, the *XMU* designs used an OPB memory controller. This controller had to be chosen, because the DDR2 memory chips of the *GTU* boards have a 16-bit wide data bus, which was not supported by any PLB memory controller of the hardware development tools at that time.

By upgrading to a new version of the development tools, a new memory controller was available. The *Multi-Port Memory Controller*[22] offers support for a large number of memory chips including the ones used for the *XMU*s. However, the upgrade required to completely re-design the Embedded PowerPC System in favor of the PLB architecture, since the OPB has been declared obsolete by Xilinx.

5.3.3 Status

The Embedded Linux System described in this chapter represents the basic platform for the development of a flexible and expandable system to monitor and control the *GTU*. The system provides a stable kernel version 2.4.30-pre1, which is basically configured to support a serial console and access to a ramdisk for the root file system. The root file system contains several system and user applications, including a shell and a number of GNU Core Utilities. The current Embedded Linux System has been successfully tested on different *GTU* boards with all three FPGA designs for more than 80 days. It is still in development stage, since the necessary device drivers to access the *GTU* system components have to be developed.

[21] The ML403 is a development platform with a Virtex-4 FX12 FPGA device. For more information, please refer to *http://www.xilinx.com/products/devkits/HW-V4-ML403-UNI-G.htm*.
[22] See chapter 4.

5.4 Future Prospects

In order to use the Embedded Linux System within the *GTU* framework, a few extensions to the kernel are still required.

First, device drivers for the *GTU* system components attached to the Embedded PowerPC System have to be developed. The drivers enable future software applications to access status and control registers of the *GTU*.
In order to access the SD Memory Cards, the kernel must be supplied with a block device driver to communicate with the SD Memory Card Controller. Such a driver would also permit to replace the ramdisk-based file system by a common one for non-volatile storage media.
Currently, the DCS boards on the back of each *SMU* represent the interface to communicate with the PowerPC processors via a shared UART connection at low bandwidth only. As stated in the previous chapter, the MGTs provided by the Virtex-4 FX100 FPGA would enable each *SMU* to establish a Multi-Gigabit Ethernet connection, which requires to extend the Embedded PowerPC System by the Xilinx TEMAC core and to integrate the existing TEMAC device driver into the kernel. High-speed data transfer is of special interest to the *GTU Boot Loader* (GBL) described in chapter 7. The GBL is a boot utility, which not only has to be capable of copying the Linux kernel image from the SD Memory Cards into main memory for execution, but is also responsible for storing new images files on the cards (see chapter 7). Regarding the image file size of about 2.7 MB of a basically configured kernel, it is desirable to have a high-speed communication link provided, since a file transfer at a data rate of 56 Kb/s would last about seven minutes. There are also other applications (*event replay*), which require to transfer huge amounts of user data. At the current bandwidth provided, a data exchange for event replay would last several days.

In summary, the Embedded Linux System developed during this thesis opens up a number of new aspects and perspectives regarding the usage capabilities of the Embedded PowerPC System. As a standardized platform, the operating system can be extended in a flexible manner by additional device drivers and user applications, which allows to adapt to the evolving requirements and purposes of the experiment.

6 The SD Memory Card Controller

All *GTU* boards have been equipped with a slot for SD Memory Cards to provide the system with mass-storage devices for various purposes. The most relevant is to serve as storage medium for the Linux kernel and its root file system of the Embedded Linux System described in the previous chapter.
SD Memory Cards have been chosen over common mass-storage devices like *(S)ATA hard disk drives* because of several significant aspects. Because of the existing magnetic stray field of both the dipole magnet and the L3-Magnet, which is of the order of 20 mT, it is not inadvisable to use magnetic storage devices. Moving parts such as the read-and-write heads of hard disk drives or the storage plates suffering from mechanical wearout are not applicable for long unserviced times at *ALICE*.
In contrast to hard disk drives, the memory content of an SD Memory Card is not affected by magnetic fields, since the cards are based on *flash technology*. There are also no loose mechanical parts suffering from mechanical wearout. Furthermore, the small dimensions of the cards make them ideal storage devices for embedded systems such as the *GTU* is.

This chapter discusses the development of a controller to access the SD Memory Cards installed on the *GTU*. After an overview of SD Memory Cards and their functional principle, the implementation of the controller is described. The last sections refer to the test environment and test results as well as possible future extensions.

6.1 SD Memory Card and Specification

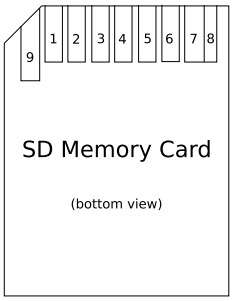
SD Memory Cards (*Secure Digital Memory Cards*) are mass-storage media based on flash memory technology. They are available with different capacities from 16 MB to 32 GB and more[1]. Figure 6.1 shows an SDHC Memory Card as installed on each *GTU* board.

SD Memory Cards are supplied with an integrated controller to access the flash memory chip. It is responsible for reading and writing the flash memory as well as optionally encrypting the data before storing. The cards provide several configuration and status registers, which hold information about the capabilities and operation conditions of a card. Table B.1 summarizes the registers.
SD Memory Cards can operate in two different modes. The *SD mode* enables access to all

[1] Cards with a capacity larger than 2 GB are usually referred to as SDHC Memory Cards (High Capacity).

Figure 6.1: *Kingston* SDHC Memory Card with a capacity of 4 GB as installed on each *GTU* board. Source: [Com]

card features including encryption support. The underlying communication protocol is called SD Memory Card protocol. Data is transferred via four parallel, bi-directional data lines a a maximum clock frequency[2] of 50 MHz, giving a maximum data transfer rate of 200 MB/s.
The second operation mode is based on an *SPI*[3] compatible communication protocol. Data is transmitted via two uni-directional data lines (one for each direction), and the clock frequency is fixed to 25 MHz. In contrast to SD mode, encryption of data is not supported. Figure 6.2(a) and table 6.2(b) illustrate the pinout of an SD Memory Card and the SPI signal assignment.



(a)

Pin No.	Name	Description
1	$\overline{CS}$	Chip select (active low)
2	DataIn	Host-to-card commands and data
3	V_{SS1}	Supply voltage ground
4	V_{DD}	Supply voltage
5	CLK	Clock
6	V_{SS2}	Supply voltage ground
7	DataOut	Card-to-host data and status
8	RSV	Reserved
9	RSV	Reserved

(b)

Figure 6.2: (a) Pinout (bottom view) and (b) pin assignment of an SD Memory Card operating in SPI mode. Source: [San04]

As discussed in section 6.3, the implementation of the developed SD Memory Card Controller is based on the SPI protocol for various advantages. The following section gives a short description of the SPI mode.

[2]The standard clock frequency is 25 MHz.
[3]Single Peripheral Interface: Synchronous serial data link standard, designed by *Motorola* to connect devices in a master-slave-fashioned manner.

6.2 The SPI Mode

When the card is powered up, it operates in SD mode. In order to enter the SPI mode, a reset command must be sent simultaneously to asserting the $\overline{\texttt{CS}}$ signal of the card. Once the SPI mode is active, the SD mode can be re-entered only by power-cycling the card.

SPI Bus protocol Communication in SPI mode between host and SD Memory Card is established in a master-slave-fashioned manner, where the host represents the master, and the card is the slave. Transactions can only be initiated by the host. The serial bit streams of the SPI mode are byte-aligned to the $\overline{\texttt{CS}}$ signal. SPI messages consist of command, response and data block tokens. A command token is always confirmed by a response token. In case of a read or write operation, an additional data block token is sent by the card or the host, respectively. Transmission in SPI mode is done with the least significant byte (*LSByte*) first, while the individual bytes are transmitted with the most significant bit (*MSBit*) first.

Command Token When operating in SPI mode, the flash chip controller of the SD Memory Card supports 25 different command tokens, which are divided into a number of classes[4]. Each command token consists of 48 bits. Table 6.1 explains the individual bit fields of an SPI command token.

Bit index	47	46	[45:40]	[39:8]	[7:1]	0
Width [bits]	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	Start bit	Transmission bit	Cmd index	Arg	CRC7	End bit

Table 6.1: Command token format. Every token consists of 48 bits. Start bit, transmission bit and end bit have fixed values. Source: [Tec06]

Response Token When the SD Memory Card receives a command token or data token from the host, it confirms reception with a response token. Commands are acknowledged with *command response tokens*, while data blocks are confirmed with *data response tokens*. Table 6.2 lists the response tokens for both response types. A description of the individual bit fields of each response token is given in [Tec06].

Data Block Token A data block token is transmitted between host and card in case of a *single block read* or *single block write* operation. It consists of an 8-bit *start block token*

[4] A description of all command tokens and classes is given in [Tec06].

Response type	Response format	Length [Bits]
Response to command	R1	8
	R1b	8 + x
	R2	16
	R3	40
	R7	40
Response to data	Data Response Token	8
	Data Error Token	8

Table 6.2: The SPI mode supports different types of response tokens of variable length, which can be classified into responses to commands and responses to data.

Byte index	514	[513:2]	[1:0]
Value	"11111110"	x	x
Description	Start token	Data block	CRC16

Table 6.3: A data block token typically has a size of 515 bytes and consists of the start token, which has a fixed value, the actual data block and the CRC16 checksum calculated from the upper 513 bytes.

followed by the actual user data block (512 bytes)[5] and a 16-bit *CRC16 checksum token*. The structure of a data block token is shown in table 6.3.

Initialization Sequence After the SD Memory Card is powered on, it operates in SD mode. In order to enter SPI mode, the host has to send a *CMD0* command token and concurrently assert the $\overline{\texttt{CS}}$ signal. The card responds with an *R1* response token and is ready to receive further initialization instructions. Once the SPI mode has been entered, a return back to SD mode is only possible by a power-cycle.
A *CMD8* must be transmitted to determine the card version and voltage range. When this information has been retrieved, it is necessary to repeatedly send the paired command combination (*CMD55*, *ACMD41*), until the card signals it has exited initialization state[6]. To finish the initialization process, the host must transmit *CMD58* and check the *CCS* flag of the OCR register in order to determine the card capacity type, and the SD Memory Card is ready for operation. Figure 6.3 illustrates the initialization sequence of SPI mode.

[5] In case of a standard SD Memory Card, the size of a data block must be at least one byte and as large as one card write block at most. The standard block size is 512 bytes. SDHC Memory Cards have a fixed block size of 512 bytes.
[6] Bit zero of *R1* is cleared.

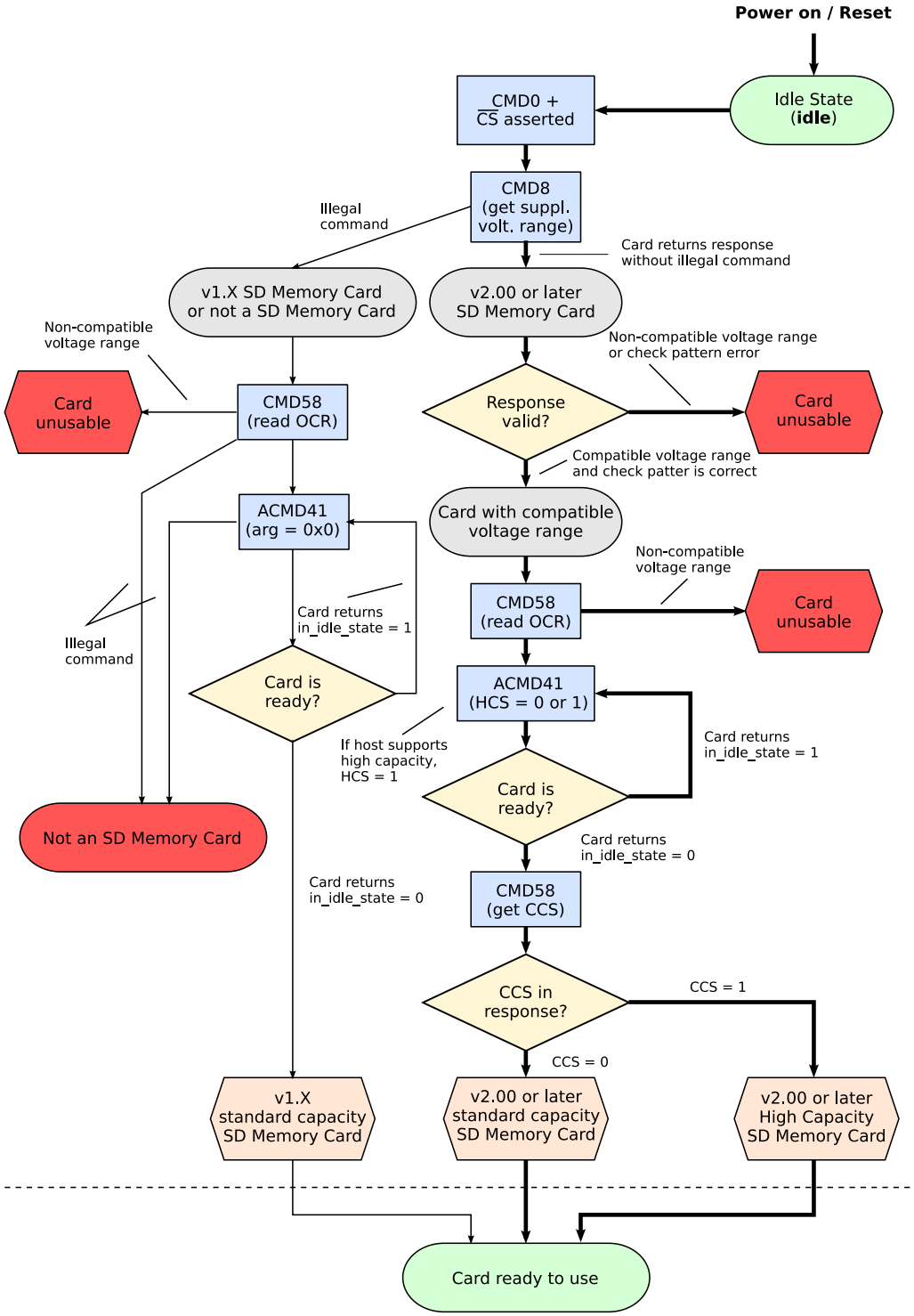


Figure 6.3: This figure illustrates the initialization sequence of an SD Memory Card when operating in SPI mode. Various properties of the card (voltage range, capacity) are checked for in order to identify the card type. The bold arrows represent the initialization branch, which has to be processed for cards compliant with SD Memory Card Specification Version 2.0. Source: [Tec06]

Read and Write Operation A single block read operation is triggered by transmitting *CMD17* with a valid *logical block address* (*LBA*) as the command argument. The *LBA* has to be aligned to the size of a data block and must not exceed the maximum *LBA*, which can be determined from the capacity of the card (in bytes) divided by the data block size. The card responds to the read operation with an *R1* response token, followed by a data block token. If the read access fails, a *data error token* is returned instead providing information about the cause of error. In figure 6.4 the timing diagram of a single block read operation is shown.

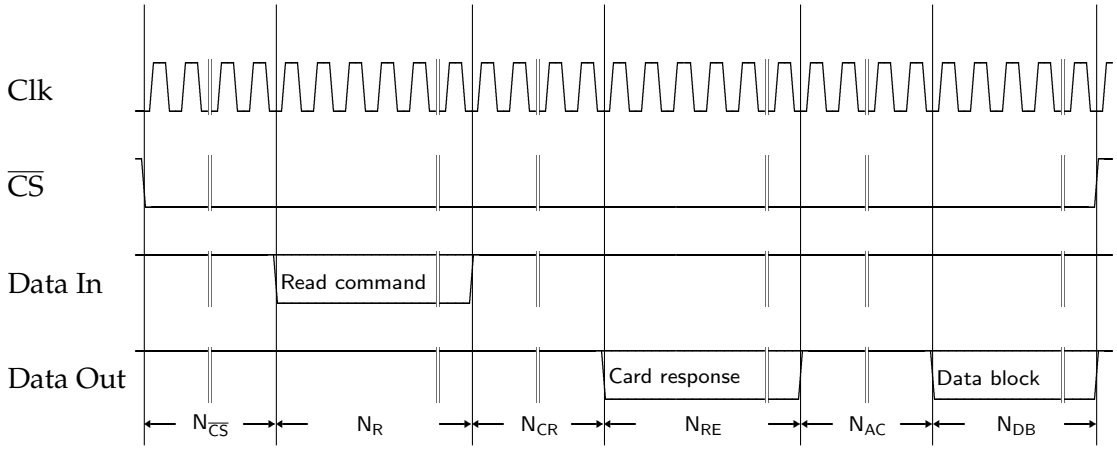


Figure 6.4: Timing diagram of a single block read operation. The host transmits the read command (*CMD17*), which is responded by the card with an *R1* and a data block token. Detailed information about timing values N_x is given in table B.3. Source: [San04]

A single block write operation is initiated by sending *CMD24*. As for *CMD17*, a valid *LBA* is expected as an argument. The card also responds with an *R1* token and waits for a data block token to be sent by the host. After reception, a *data response token* is returned and completes the transaction. Until the card has finished writing the data to the memory, it sends *busy tokens*[7]. Figure 6.5 illustrates the timing diagram of a single block write operation.

6.3 The SD Memory Card Controller

The development of the SD Memory Card Controller aims to design an efficient device, which is compatible with different types of memory cards and does not depend on specific hardware requirements in order to be applicable with different designs and platforms as well.

[7]During a busy token, the output data line is kept down to zero for eight clock cycles.

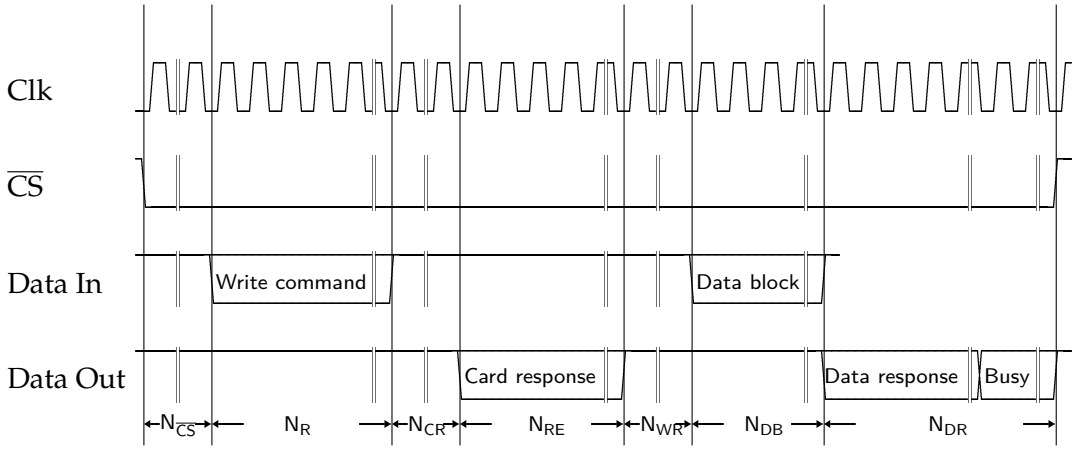


Figure 6.5: Timing diagram of a single block write operation. The host issues the write command (*CMD24*), which is responded with an *R1* token. The card waits for a data block token, which has to be transmitted by the host. The card confirms reception of the data block token with a data response token. As long as the card has not finished programming the data, it continuously sends busy tokens. Detailed information about timing values N_x is given in table B.3. Source: [San04]

A basic question for the development of the controller is whether to operate in SD mode or SPI mode. Since the SD mode underlies a proprietary protocol, it requires a license to make use of it. Furthermore, the SD protocol has a much higher complexity than the SPI protocol. An implementation of the SD mode would consume a lot of valuable logic resources. In contrast, the SPI mode is based on a straightforward protocol and can be implemented in a less resource-consuming manner.
Regarding speed and performance, the SD mode provides a much higher data transfer rate of 200 MB/s in contrast to 25 MB/s of the SPI mode. The SD mode also supports encryption of data before storing on the memory chip. However, both are features of minor interest in the present context. Once the Embedded Linux System is running, the SD Memory Cards are write-accessed only with a low amount of data[8]. Considering the preceding discussion, the decision was made in favor of the SPI mode, which additionally offers a wide compatibility range with other card types such as *Multi Media Card* or *CompactFlash Card*.

Another question, which has to be considered, concerns the actual implementation of the controller. Since the Virtex-4 FX family is supplied with embedded PowerPC cores, a first draft suggests a controller completely written in software, which accesses the card through a basic SPI hardware communication layer. The software would be responsible for command sequencing and the handling of incoming responses as well as transferring and receiving data blocks. However, this approach represents the contrary to the demand for independence of specific hardware requirements, since it completely relies on a con-

[8] Actually, the Linux kernel at present lacks a device driver to access the cards. See section 5.4.

trolling computer.
A controller completely implemented in hardware is inappropriate as well. As illustrated in figure 6.3, the initialization sequence is a complicated and nested process. Dealing with all eventualities would result in a complex and resource-consuming FPGA design. On the other hand, the flexibility of software allows to realize the card initialization in a straightforward manner, while time-critical read and write operations can be implemented in hardware. The considerations above lead to a solution, which combines the best of both approaches: Time-critical read and write operations are implemented in hardware, while initialization and status control of the card is done via software.
The implementation of the *data path* is another subject to be taken into consideration. Because data is transmitted block-aligned between host and card, a buffer of the size of at least one data block is required. The data buffer can be implemented as *FIFO* or *dual-ported* BRAM. FIFOs would have to be read out sequentially, which makes manipulation of a specific word of the data block an inefficient task. Unlike FIFOs, dual-ported BRAMs are randomly accessible, which allows to modify any single data word within the memory.
To increase the performance of the controller, a technique referred to as *bank switching* is used. The BRAM is divided into several regions (banks) of the size of one data block. By switching between those banks, it is possible to process or prepare another data block, while the card is still busy transferring or receiving data. This mechanism is discussed in paragraph 6.3.1.
Typically, host and controller are located in different *clock domains*[9]. That is, the signals between the two clock domains must be synchronized by a so-called *clock domain crossing*. The clock domain crossing of *dual-clock* FIFOs based on *distributed RAM* requires particular attendance because of their critical timing. However, BRAMs are provided with two independent interfaces, which share the same memory and allow to access it from different clock domains, while the clock domain crossing is handled internally.

The SD Memory Card Controller developed and described in the following sections unifies compatibility (SPI mode), flexibility (the generic interface allows to operate the controller by both software and hardware) and performance (bank switching) in a straightforward manner.

6.3.1 Implementation

The implementation of the SD Memory Card Controller comprises three parts. The *finite state machine* (*FSM*) generates the control signals for the SD Memory Card and other subcomponents of the controller core. The *address path* maps the *bank address scheme*[10] of the

[9] In a clock domain, all entities are driven by the same clock frequency.
[10] When using a bank address scheme, linear memory is divided into several equal regions, each referred to as a memory bank. The *bank address* selects the bank to operate on, while an *offset* addresses memory words within a specific bank.

host on a *linear address scheme* used inside the controller to access the BRAM. The *data path* prepares the commands for transmission and processes incoming and outgoing data. A schematic overview of the SD Memory Card Controller is given in figure 6.7.

The Finite State Machine The state machine is the central element of the controller. It uses three user control input signals (`USR_SD_CMD_INIT`, `USD_SD_RW_INIT`, and `USR_SD_RNW`) and three user handshake output signals (`USR_SD_CMD_ACK`, `USR_SD_RW_ACK`, and `USR_SD_BUSY`). Further, it generates control signals for the SD Memory Card ($\overline{\texttt{CS}}$) and the BRAM (`WE`). The remaining incoming and outgoing control signals are for internal use. A diagram of the complete *FSM* is shown in figure 6.6.

Upon reset, the state machine first enters the `IDLE` state and remains unless a command execution is requested. When a command is issued, the state `CS_HOLD` is entered and held for seven clock cycles to ensure a correct alignment to the $\overline{\texttt{CS}}$ signal. Afterwards, the transmission of the command is started. The state machine implies two major command branches to separate between fix implemented read and write commands (*FSM commands*) and external initialization and control/status commands (*USR commands*).

When the host issues a read or write operation by asserting the signal `USR_SD_RW_INIT`, the *FSM command* path is entered. According to the value of signal `USR_SD_RNW`, either a read or write transaction is performed. In both cases, the serial transmission of the command is prepared (`FSM_CMD_START`). The next three states of both sub-branches are identical regarding their functionality. The state machine remains in `FSM_CMD_<OP>`, where `<OP>` designates the corresponding transaction type (`READ` or `WRITE`), until all command bits are transmitted. Afterwards, the state `FSM_<OP>_WAIT_RESP` is entered to wait for the incoming response[11]. The response is captured and stored to a register to be available for further processing by the host (`FSM_<OP>_CAPT_RESP`).
In case of a read operation, the controller waits for the data block and stores it into the selected BRAM bank (`FSM_READ_WAIT_DATA`, `FSM_READ_CAPT_DATA`). The CRC16 checksum of the preceding data block is captured separately (`FSM_READ_CAPT_CRC`). When finished, the state machine returns to `IDLE` state.
When performing a write operation, the controller prepares and sends the start token first to signal the start of data transaction (`FSM_WRITE_DATA_START_TOK`). In the following states `FSM_WRITE_DATA` and `FSM_WRITE_CRC`, the 512-byte data block as well as the CRC16 calculated from the data block is transmitted to the card. The two states `FSM_WRITE_WAIT_DATA_RESP` and `FSM_WRITE_CAPT_DATA_RESP` are responsible for capturing the data response token, which is stored in the response register. The state machine remains in the state `FSM_WRITE_BUSY` until the card has finished the writing data to the memory and stops sending busy tokens. Finally, the state machine returns to `IDLE` state.

[11] A response is triggered by the card dropping its data out line to zero.

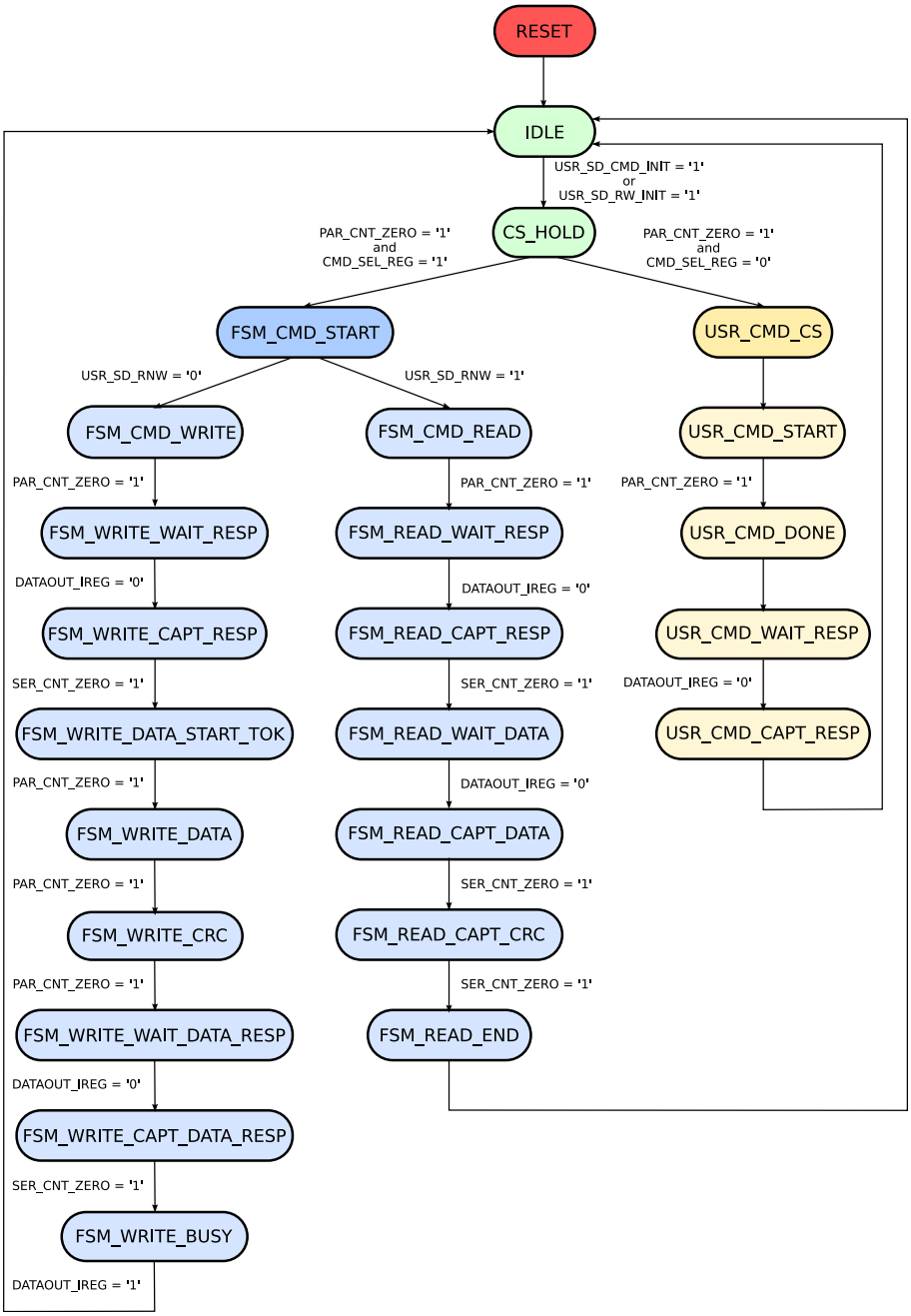


Figure 6.6: The state machine of the SD Memory Card Controller processes several state signals and generates signals for controlling internal and external components. Command processing is divided into two branches. The *FSM command* path is entered if the host triggers a read or write operation. Depending on the operation, the corresponding states are passed. If the host issues any other command, the *USR command* states are processed.

If the host issues a user command, the states of the *USR command* branch are traversed. A USR command is initiated by asserting the signal `USR_SD_CMD_INIT` causing the state machine to enter `USR_CMD_CS`. The transmission of the command is prepared and executed in state `USR_CMD_START`. Waiting for the response and capturing it is done the same way as already described for the *FSM command* path (`USR_CMD_WAIT_RESP` and `USR_CMD_CAPT_RESP`).

During all states except `IDLE`, the controller asserts `USR_SD_BUSY` to avoid being interrupted while a transaction is running.

The Address Path An 18 Kb dual-ported BRAM is used to buffer the data blocks exchanged between host and SD Memory Card. Since a data block has a standard size of 512 bytes, the BRAM can be divided into four *banks*, each capable to store one data block. The host determines the bank to operate on with the two bit wide signals `USR_BANK` and `USR_SD_RW_BANK`, respectively. The former signal is used for data exchange between the host and the BRAM, while the latter indicates the memory bank accessed by the card. Two seven bit wide offset addresses (`USR_OFFSET` and `OFFSET`) enable access to 128 32-bit words within a specific bank. The address path is responsible for mapping the bank address scheme to a linear address scheme as used to access the BRAM. The relation between both address schemes is shown in table 6.4.
Logical block addresses, which are used to access the data blocks of the SD Memory Card, do not require address mapping.

Bit position	14	[13:12]	[11:5]	[4:0]
Value	'0'	x	x	"00000"
Description	Fixed	Bank no.	Data word offset within block	32-bit alignment

Table 6.4: Address mapping between bank addresses and linear addresses.

The Data Path The data path can be divided into two branches, the *command and response path* and the *data block path*. The command and response path is responsible for preparing and transmitting both USR and FSM commands issued by the host. It also captures the card responses in a register for further evaluation by the host. Before a command is transmitted, a CRC7 checksum is calculated from the command index and argument. Index, argument and CRC7 are concatenated and loaded into a 48-bit *shift register*, which serves as a *serializer* and starts transmitting the individual bits. Simultaneously, a decrementing counter is set to the length of the command. A *zero-crossing* of the counter signals the end of command transmission. Another 40-bit shift register (the *deserializer*) captures the incoming response of the card. According to table 6.2, the length of a response token varies with the response type, but is 40 bits in maximum[12]. In or-

[12]Except *R1b*, which can be split up into *R1* and an undefined number busy tokens.

der to avoid the overhead to determine the expected response token from the command, the maximum number of response bits are captured, and the decrementing counter is set accordingly. When the counter reaches zero, the response has been completely captured and is stored into a register for further processing by the host, which is responsible for evaluating the response accordingly.

Incoming and outgoing data blocks are handled by the data block path. A 18 Kb dual-ported BRAM, which is divided into four memory banks, buffers the data blocks exchanged between host and card. To efficiently store the data blocks, the controller is capable to switch between different BRAM banks. While the SD Memory Card processes a write operation and thus reads data from one memory bank, the host can simultaneously store data in another bank for the next write transaction. The same applies to the card in case of a read operation. This switching mechanism is a relevant enhancement of data throughput combined with true random access within a data block.
The signal `USR_SD_RW_ENDIANESS` determines the byte order of the data located in the controller BRAM. If the signal is de-asserted, data is interpreted or stored *little* endian, while asserting the signal corresponds to *big* endian.
In case of reading data from the card, the de-serializer is used to capture the incoming serial data stream. The corresponding counter is set to the length in bits of the data stream[13]. A CRC16 is calculated from every incoming byte resulting in the total checksum of the entire data block. Every 32 bits, the lower four bytes of the shift register are read out[14]. According to `USR_SD_RW_ENDIANESS`, the 32-bit data words are stored to the BRAM. This process is repeated until all 128 words have been received (zero-crossing of the counter). The final checksum is compared to the one transmitted by the card, which has been calculated by the card-internal controller. If no CRC error is detected, `USR_SD_NO_CRC_ERR` is asserted. Otherwise, the signal remains de-asserted to mark the data as invalid, and the host has to react in a corresponding manner. In case the read access fails, a *data error token* is sent by the card, which keeps information about the cause of the error. Error information is stored in the response register for evaluation by the host.
Writing data to the card is slightly different, since a data start token must precede the data block after the card response has been received. A 2:1 multiplexer controlled by the state machine selects between data start token and data words from the BRAM. After sending the start token, data words of 32 bits are sequentially loaded from the designated BRAM bank into the serializer according to the selected byte order. A counter is set to the size of the data block, and the serializer transmits data in a corresponding manner. The total CRC16 checksum of the entire data block is calculated from each single byte and transmitted to the card as well. Finally, the card confirms data with a *data response token*.
Because card data can be accessed one block at a time only, modifying specific words within the memory chip of the card implies a *read-modify-write* operation. The entire block, which the designated words are located in, is loaded into the BRAM. The host

[13] Length of one data block in bits: 512 * 8 bits = 4,096 bits
[14] Both shift registers always hold the data words in big endian order.

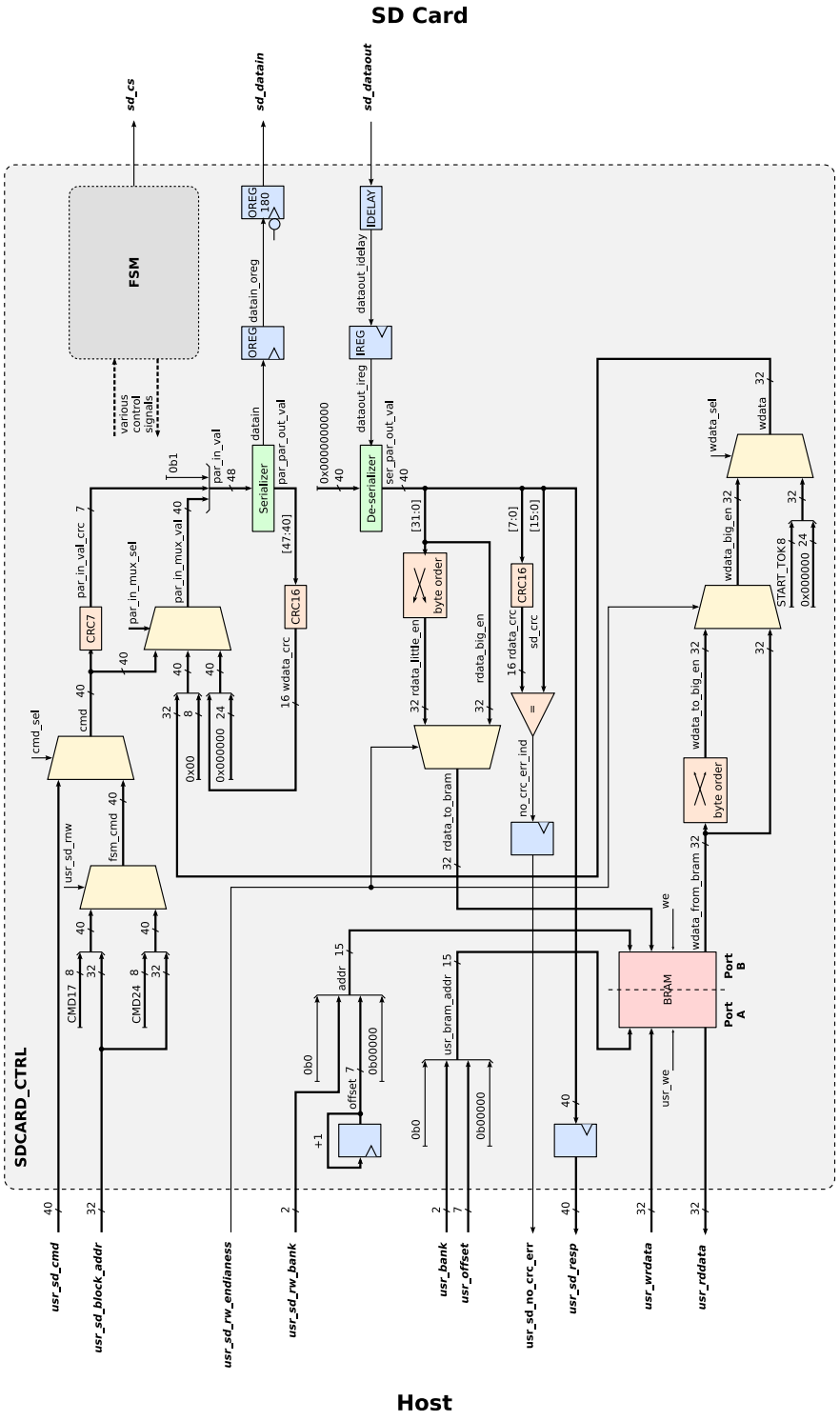


Figure 6.7: Schematic view of the SD Memory Card Controller. Two shift registers are used as serializer and de-serializer. Multiplexers and a dual-ported 18 Kb BRAM allow a fast and straightforward data exchange with the card.

then accesses the specific word addresses by the bank offset. Last, the block is sent back to the card again.

The SD Memory Card Controller PLB Interface In order to access the cards through the controller by software running on the embedded PowerPC processor cores of the Virtex-4 FX FPGA, a special interface is required, which allows to attach the SD Memory Card Controller to the Processor Local Bus (PLB) of the Embedded PowerPC System discussed in chapter 4. The interface provides two command registers and a control register, which are mapped into the address space of the processor. The two command registers are required, because the PowerPC is based on a 32-bit memory architecture and the 40-bit user commands[15] have to be split into the 8-bit command index (seven index bits plus one start bit) and the 32-bit command argument. The control register is eight bits wide and used to combine the control signals. As for user commands, the 40-bit response register of the controller is split into an 8-bit and a 32-bit signal. A detailed description of the interface and its registers is given in chapter 6 and table B.2.

6.3.2 Integration of the Controller into the GTU Design

To integrate the SD Memory Card Controller into the *GTU* design, the interface for the controller is attached to the PLB of the PowerPC design. The PowerPC top entity must be supplied with additional interfacing ports for the controller (see right side of figure 4.6). The SD Memory Card Controller is added to the top-level design of the *GTU* and connected to the corresponding ports of the PowerPC entity. Embedded PowerPC System and controller are driven by different clock signals, derived from a central *clock generation entity*. While the PowerPC processor and the PLB are driven at a clock frequency of 400 MHz and 100 MHz, respectively, the controller runs with a 25 MHz clock.

6.4 The Software Interface

The software interface provides the user with a set of routines for basic card operations such as sending commands and receiving card responses out of which more complex operations are constructed. These include initializing the card or reading and writing data blocks from or to the card. The card responses are evaluated to check for errors signaled by the card. Additionally, the initialization status of the card is summarized in an *initialization status indicator*. Although the hardware part of the controller supports bank switching, it is not implemented yet, since it gets relevant in combination with a Linux block device driver only.

[15]The lower eight bits of the command are appended by the controller.

Fundamental commands are integrated into the *GTU* system software to provide a first working version of interaction with the SD Memory Cards. These commands are summarized in table 6.5.

Command	Options	Description
sdcard init	-	Initialize SD Memory Card
sdcard initstat	-	Get initialization status
sdcard read	<block addr> [<number of blocks>]	Read <number of blocks> beginning from <block addr>
sdcard write	<block addr> <d1> ... <d512>	Write 32-bit data words to <block addr>

Table 6.5: Fundamental commands of the *GTU* system software. Currently, only initialization, read and write operation are supported.

Sending User Commands User commands are transmitted to the SD Memory Card by using the method **sd_send_cmd**(*cmd_struc*). A command structure is passed as a parameter containing the command index and its argument. The values are stored to the registers `CMD_REG0` and `CMD_REG1` of the bus interface, respectively. The command transaction is initiated by setting the corresponding bit of the `CTRL_REG` register (see table B.2). The controller transmits the command to the card and polls for the signal `USR_SD_CMD_ACK`, which confirms command reception. If the assertion of the signal is not detected within a certain time period[16] a timeout error is raised, and the transaction is aborted.

Receiving Response Tokens Responses from the card are fetched by the method **sd_recv_resp**(*resp_buf*). The busy signal is polled until it is de-asserted or a timeout error aborts the transaction after 500 µs with an error message. Otherwise, the response registers `RESP0` and `RESP1` of the controller are read out, and their content is stored in *resp_buf*.

All other functions provided by the software interface are built upon the basic communication methods described above. In the following, the rest of them are described to give an insight into initializing the card as well as loading and storing data blocks.

Software Initialization Sequence Apart from SD(HC) Memory Cards, the *XMU* board hardware supports no other card types. Because SDHC Memory Cards are compliant to *SD Specification Version 2.00*, only the main branch of the initialization sequence (bold arrows in figure 6.3) is processed by the initialization routine (**sd_init()**) illustrated in

[16] The timeout value can be set manually and is 500 µs as default.

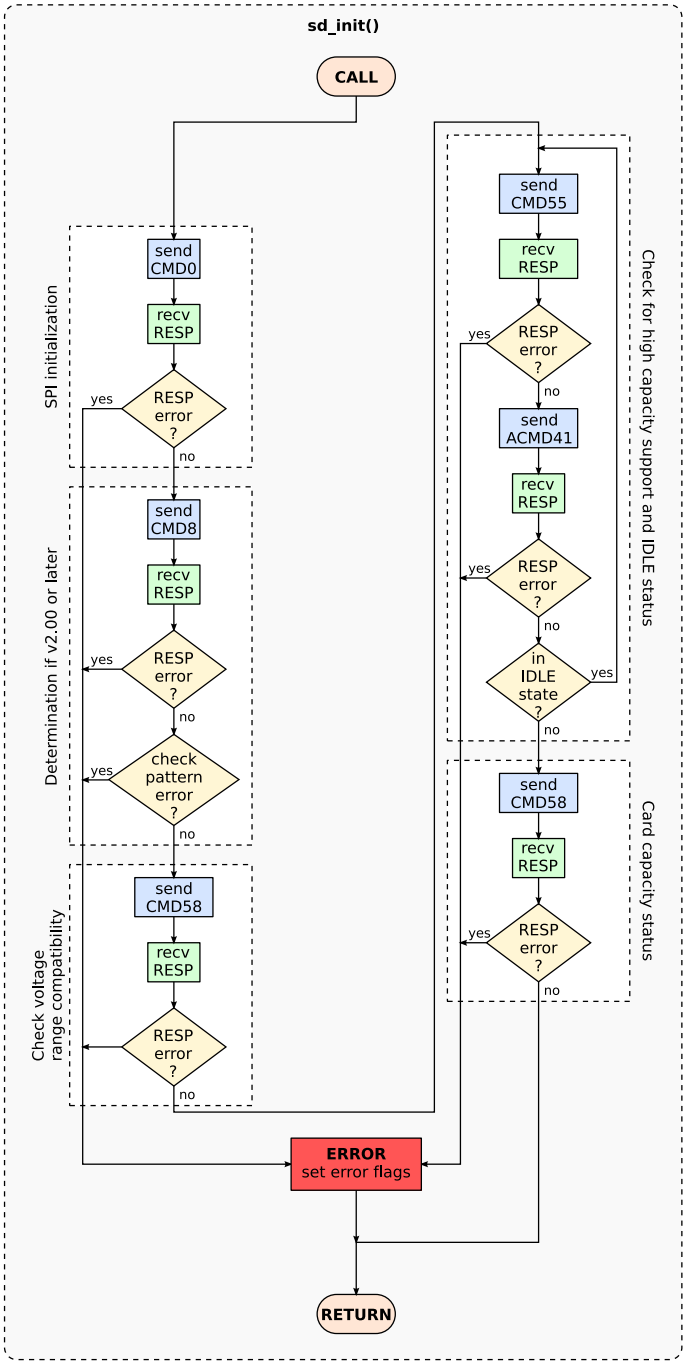


Figure 6.8: This diagram describes the initialization sequence as it is implemented in software. The dashed sections can be identified with the corresponding initialization states of the main branch shown in figure 6.3.

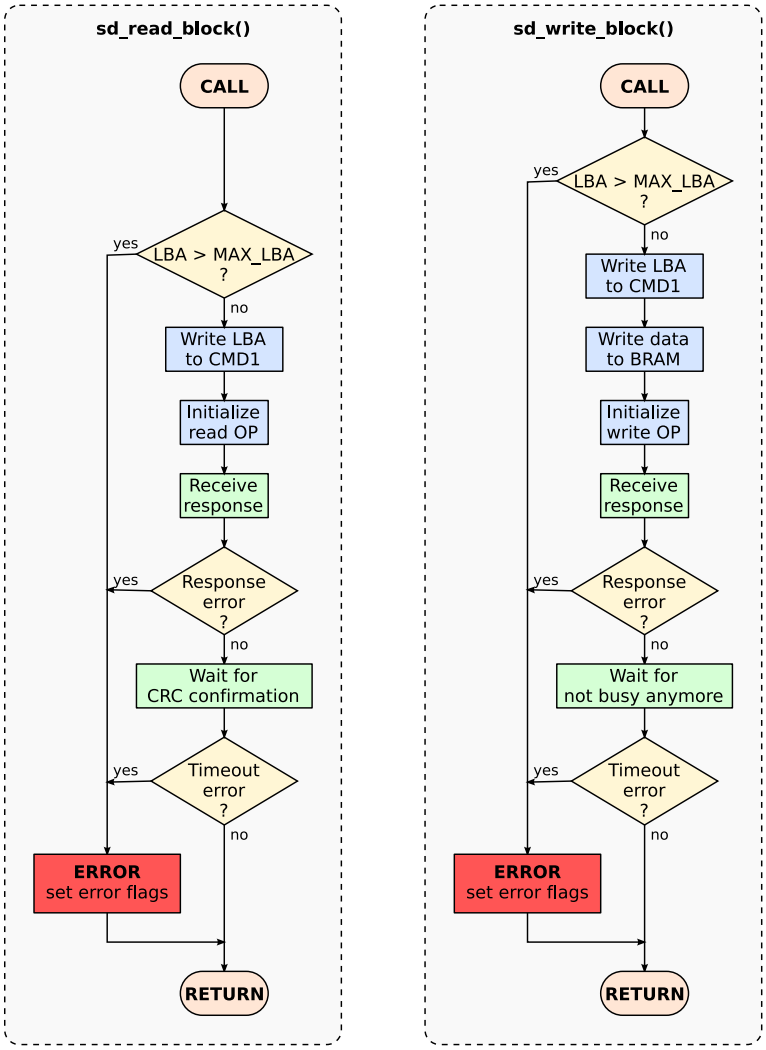
figure 6.8. The dashed sections correspond to the states of the main branch of the initialization sequence. The card responses are evaluated to check for occurring errors. Since this is a first draft of the initialization routine, a detected error is reported only, and the initialization is canceled immediately without further examination. The initialization status of the card is obtained from the output of the responses and the status signals of the bus interface. A description of the individual bits of the status indicator is given in table 6.6.

Bit position	Description
0	Card is present
1	Card is write-protected
2	Card is legacy (ver. 1.x) (CMD8)
3	Card unusable (non-compatible volt. range / chk pattern err) (CMD8)
4	Card unusable (non-compatible volt. range) (CMD58)
5	Ver. 2.00 or later card
6	Error on command (not further specified)
7	Initialization successful

Table 6.6: The initialization status indicator contains information about the initialization process of the card.

Reading Data Reading a data block from the card is done by using the function **sd_read_block**(*block_addr*, *rbuf*, *endianess*). The first parameter is the logical block address *LBA* of the card block to read from. It is checked against the maximal block address permitted, which depends on the capacity of the card. In case of a valid *LBA*, it is written to the `CMD1_REG` register. Further, a buffer to store the data in is required, which has to have at least the size of one data block. The byte order how the data is stored in the BRAM by the controller has to be determined by the last parameter. The read command is issued by setting the corresponding bit of the register `CTRL_REG` of the bus interface (see table B.2). If the response has indicated no errors, the data block stored in the BRAM is copied to the buffer. Finally, the function returns to the main function. In figure 6.9(a) an outline of the read data process is shown.

Writing Data Figure 6.9(b) describes the function **sd_write_block**(*block_addr*, *wbuf*, *endianess*), which is used to write a data block to the SD Memory Card. The parameters are similar to that of the function **sd_read_block**, except that the buffer contains the data to be written to the SD Memory Card. The *LBA* is checked for validity as well and stored to the `CMD1` register. Next, the content of the write buffer is written to the BRAM. As for reading data, the endianess of the data has to be determined. When the data has been written to the card, the incoming response is checked for errors. While the card is busy to



(a) (b)

Figure 6.9: Diagrams of the **sd_read_block** (a) and **sd_write_block** (b) functions of the SD Memory Card Controller software interface. The responses are evaluated to check for possible occurring errors. If an error has been detected, the transaction is aborted immediately, and it is returned to the main function.

store the data block to its memory, the `USR_SD_BUSY` signal remains asserted by the controller. The software polls for de-assertion of this signal and returns to the main function if card and controller are not busy any longer.

6.5 Test Environment and Test Results

To verify the functionality of the SD Memory Card Controller, a test setup similar to that described in section 4.4 is used. Additionally, the board is equipped with a 4 GB SDHC Memory Card as shown in figure 6.1. The controller is integrated into the top level of all three FPGA designs and connected to the PowerPC top level entity. A computer interfaced with both the JTAG and the UART connector of the board is used to program the FPGA device with a *GTU* design. Via the Xilinx Microprocessor Debugger (XMD), a test application is downloaded into the PowerPC BRAMs. It generates pseudo-random data for each data block to be written to the card and stores it to a write buffer. The time to write and read a specified number of card blocks is measured. Each card block, which has been read out, is compared to the write buffer in order to detect possible bit errors. The time to write and to read as well as the number of detected bit errors are added up with the values of the previous test cycle, and the next test cycle is entered. During the test period of about 13 days, a data volume of 341.25 GB has been transferred to the SD Memory Card, which corresponds to a data transfer rate of 0.16 MB/s (write access) and 7.69 MB/s (read access). Within this test period, no bit errors have occurred, leading to a bit error ratio of less than $3.4 \cdot 10^{-13}$, which is close to industry standard. Detailed information about the test results is listed in table 6.7 below.

$\mathbf{dv_{tot}}$ [GB]	$\mathbf{ttw_{tot}}$ [s]	$\mathbf{ttr_{tot}}$ [s]	$\mathbf{dr_w}$ [MB/s]	$\mathbf{dr_r}$ [MB/s]	**BER**
341.2	$1.1 \cdot 10^6 s$	$2.3 \cdot 10^4$	0.16	7.69	$< 3.4 \cdot 10^{-13}$

Table 6.7: The table lists the test results of the SD Memory Card Controller. During the overall test time of $\mathbf{ttw_{tot}}$ + $\mathbf{ttr_{tot}}$ $\approx$ days, a total data volume of $\mathbf{dv_{tot}}$ = X GB has been transferred between card and host. The average data transfer rate of write and read access is given by $\mathbf{dr_w}$ and $\mathbf{dr_r}$, respectively. **BER** represents the bit error rate calculated from $BER \leq \frac{N_{err}}{dv_{tot}[bit]}$, where N_{err} is the number of bit errors.

6.6 Status and Future Prospects

In summary, the present SD Memory Card Controller is provided with all necessary features to meet the demands stated at the beginning of this chapter. Due to the SPI-based communication protocol, which can be implemented in a resource-saving manner, the controller is compliant with different kinds of SD Memory Cards as well as several other memory card types such as MMC or CompactFlash Card. Any hardware, which provides the necessary controlling capabilities and interfacing ports can act as host and access the cards. Efficient data transaction is realized by hardware-controlled handling of data transaction and providing the capability to use a bank switching mechanism. The measured bit error ratio is less than $3.4 \cdot 10^{-13}$.

The control software provides the necessary routines to operate the cards used in the *GTU*. Aside from initializing cards compliant with the SD Specification Version 2.00, the controller supports reading and writing single data blocks.

Although the controller is ready-to-use as-is, there are extensions, which could be made in future. To provide access to older card versions and other card types, the software initialization sequence could be adapted to deal with the other initialization branches shown in figure 6.3. In fact, this is of minor relevance to the *GTU*, because it is equipped with SDHC Memory Cards only. For reasons of future use with platforms supporting other card types, the software could be extended in the corresponding manner.
At present, there is no support to access specific card-internal configuration and status registers (see table B.1), which contain useful information about the SD Memory Card in use. Thus, it might be advisable provide dedicated software routines in order to read, evaluate and write those card registers.
An important improvement to both the controller hardware and software is the implementation of extensive error handling. If the card detects an error (e. g. transmission error), the state machine currently does not abort and enter a dedicated error state. Instead, it passes all following states until the transaction is completed. To handle the possible occurring errors during a read or write transaction, both *data error token* and *data response token* should be evaluated. A data error token is returned, if a read operation fails, and the card can not provide the required data. Every data block written to the card is acknowledged by a data response token. In case of a write error, the host may send further commands to determine the cause of the problem.
Finally, to enable the Embedded Linux System described in chapter 5 to access the cards as well, it is necessary to develop a Linux block device driver. The main purpose of the SD Memory Cards is to serve not only as a boot medium for the Linux, but also to host the Linux root file system.

The SD Memory Card Controller has been extensively tested by a corresponding test configuration (see section 6.5). The results have indicated no errors yielding a bit error rate of $< 3.4 \cdot 10^{-13}$. That is, the controller is close to industry standard and ready-to-use. It has been integrated as a fully operative part into the *GTU* main design currently in use at the *CERN*.

7 The GTU Boot Loader (GBL)

A *boot loader* is a program, which is executed previously to the actual operating system. It is responsible for loading the kernel and other parts of the operating system from the boot medium into the main memory. However, this generally assumes the presence of a *Basic Input Output System* (*BIOS*), which provides the boot loader with the necessary routines to access the boot medium. In case of embedded systems, there is commonly no BIOS available.
The SD Memory Cards installed on the *GTU* serve as boot media for the Embedded Linux System discussed in chapter 5 and are accessed via the SD Memory Card Controller described in the previous chapter. However, the *GTU* does not provide any kind of BIOS, therefore the boot loader must supply routines to access the controller. Thus, commonly used boot loaders like *GRUB*, *LILO* or *U-Boot* are unusable.
The desired boot program not only has to provide typical functionality to load and execute the kernel image, but must also be supplied with administration and recovery capabilities. Because the cards are permanently installed on the *GTU* located in the *ALICE* pit, they can be remotely accessed only. Thus, the boot loader must be capable to write access the cards as well to store a Linux kernel image on the SD Memory Cards. Furthermore, in case of a system failure causing the card file system to be damaged, the boot program must provide recovery capabilities such as reformatting the card. This is very specific functionality common boot loaders do not offer.
The boot program should also be able to allow the user to select between different kernel images on the card. This is of particular interest for testing and development purposes in order to have kernels at different configuration and development stages accessible.

In this chapter, the development of a boot loader for the *GTU* is described. First, the general idea of the boot process is discussed. Subsequently, a description of the boot loader and its basic capabilities is given, followed by a summary of the test setup and problems encountered. The last section is related to future extensions and possible improvements.

7.1 The Boot Process

When a computer system is powered up or reset, the internal registers of the CPU are set to initial values. In particular, the *Program Counter*[1] (PC) is set to a specific address referred to as *reset vector*, from where the CPU starts execution. On non-embedded systems, this address contains an instruction to jump to the entry point of the BIOS, which manages several initialization tasks. This includes identifying and testing different hardware components as well as initializing the *interrupt vector table* and providing basic I/O methods to access the hardware. When finished, the BIOS copies the bootable content of the *Master Boot Record* (*MBR*)[2] of the boot medium to a dedicated address in main memory and continues execution at this address. These instructions are normally part of a boot loader, which accesses the storage device to copy and start the operating system kernel by using the BIOS routines.
Regarding the boot process of the Embedded PowerPC System, the processor starts execution at address 0xFFFFFFFC . As typical for most embedded systems, the *GTU* is not supplied with a BIOS. Instead, the reset vector points to the last 32-bit data word of the BRAM instance, which holds an instruction to jump to the entry point of the main program. While this has been the *GTU* system software up to now, it is replaced by a boot loader, which implements BIOS functionality.

7.2 The GTU Boot Loader (GBL)

The *GTU Boot Loader* (*GBL*) is responsible for loading and booting the kernel images from the SD Memory Cards. Since the *GTU* lacks a BIOS, the GBL must be provided with basic I/O methods to access the cards via the SD Memory Card Controller discussed in the previous chapter. Furthermore, the cards can be remotely accessed only, thus the GBL must implement administration capabilities such as storing kernel images to the cards as well. The software layer, which is used to interface this controller offers both, read and write access to the cards on raw data level. However, SD Memory Cards are usually formatted with a FAT32 file system, which must be accessible by the Linux kernel in order to make use of the cards as common file system-based mass storage media. Thus, the access routines provided by the GBL have also to preserve the integrity of the card file system rather than operating in raw data mode.
Regarding recovery capabilities, it is necessary to supply methods, which allow to format a card with a FAT32 file system. This is of major interest in case the file system of a card is damaged, because it is the only option to bring the system back to an initial state.

[1] Also: *Instruction Pointer* (IP)

[2] The MBR is the first data block (512 bytes) of a partitioned storage medium and holds the *partition table* as well as a boot loader. However, the MBR can optionally reference the boot sector of a specific partition, which contains the actual boot loader.

For testing and development purposes, it is desirable to have several kernel images of different configuration available to boot. This implies a configurable kernel image path, which can be adapted during runtime to select the path a kernel image will be stored to or is loaded from.

7.2.1 Implementation and Integration

A boot loader can generally be implemented following two different approaches, either *single-staged* or *multi-staged*. Single-stage boot loaders are small enough to entirely reside in the boot sector of the storage medium. In contrast, the multi-stage approach is applied, if the boot program code is too large to be stored within the boot sector and must be divided into several parts. The boot sector then only keeps the initial boot stage, which loads further boot program code.
In the present context, a BRAM instance of the Embedded PowerPC System represents the initial boot medium. The reset vector of the processor points at the last data word of the BRAM, which holds an unconditional jump to the entry point of the boot loader code before. Regarding the two boot models, the GBL can be implemented as follows:

- **Single-staged approach**
  The entire GBL is located in the BRAM. To boot the Linux kernel, it copies the kernel image from the SD Memory Card into the DDR2 SDRAM and executes it.
- **Multi-staged approach**
  - A minimalistic boot routine in the BRAM copies the GBL from the SD Memory Card to the DDR2 SDRAM.
  - The GBL loads the Linux image from the SD Memory Card into the main memory and executes it.

Providing that recovery capabilities are entirely hosted by the GBL, the multi-stage approach proves to be inappropriate. In case of a damaged file system, the only option for recovery would gone, if the GBL is also broken. Thus, the decision is made in favor of the single-stage approach. Figure 7.1 illustrates the implementation of the single-stage boot process.

The GBL at its present state is capable of read-accessing the SD Memory Cards and copying a dedicated kernel image into main memory to execute it. The boot routine is implemented in the following steps:

1. Initialize the SD Memory Card.
2. Read the *partition table* located in the *Master Boot Record* of the card.
3. Access the *FAT32*-formated partition which contains the Linux image file.
4. Copy the binary kernel image to the main memory.

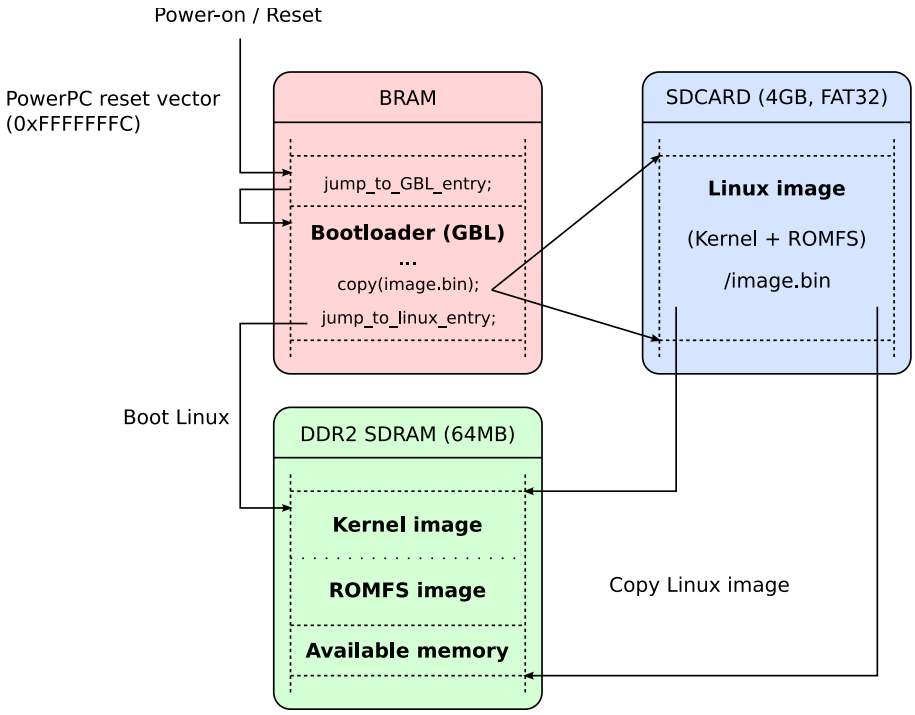


Figure 7.1: Single-stage boot process. When the PowerPC processor is reset, it starts execution of the boot loader located in the BRAM, which copies the binary Linux image from the SD Memory Card into the DDR2 SDRAM. Afterwards, the processor continues execution at the entry point of the kernel to boot the operations system.

5. If any error occurs so far, print an error message and retry.
6. Continue execution at the entry point of the kernel.

Figure 7.2 illustrates the implementation of these steps by the boot routine **bl_load()** of the GBL. For initialization of the card, the corresponding method of the controller software layer is used (**sd_init()**).
Since the present SD Memory Cards are partitioned following the *PC BIOS MBR partitioning scheme*, the *Master Boot Record* (*MBR*) must be read out to retrieve information about the card partitions stored in the partition table (**partition_init()**). A detailed description of both the MBR and the partition table is given in appendix, chapter C.
The cards used are formatted with a FAT32 file system. As stated at the beginning of this section, a special set of routines is required to access the file system properly. With the functionality of these routines based on the I/O methods of the card controller, the boot loader is capable to read and evaluate the partition table of the cards as well as reading data from the existing FAT32-formated partitions (**fat_init()**).
At present, a dedicated kernel image is expected to be located in the *root directory* of one of the primary partitions of the SD Memory Card. The GBL searches for the image file and

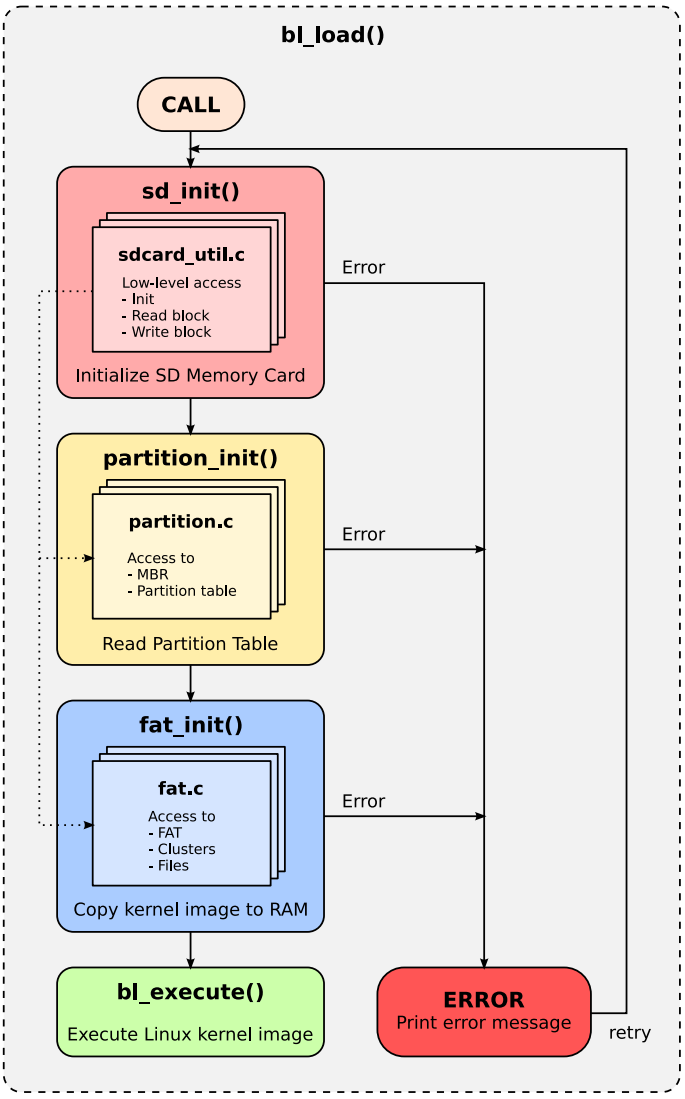


Figure 7.2: The GBL boot routine **bl_load()** encapsulates the individual steps necessary to boot the kernel. Initialization of the SD Memory Card is done by **sd_init()**. Both **partition_init()** and **fat_init()** are responsible for reading and evaluating the partition table as well as accessing the FAT32 file system and copying the kernel image to the main memory. Last, **bl_execute()** branches to the entry point of the kernel to start the operating system.

copies it to the main memory, if it is found (**fat_init()**). If no error has occurred during the previous steps, the processor jumps to the kernel entry point and starts the operating system (**bl_execute()**). In case of an error, the implementation of the boot routine at the present development stage preserves to display an error message and retries the boot procedure.

7.3 Status and Future Extensions

The *GTU Boot Loader* (*GBL*) at present is still under development, but provides basic I/O functionality to access the SD Memory Cards via the controller developed for this thesis. The GBL is capable of read-accessing FAT32-formatted cards and loading the kernel image to be executed. The image file is expected in the root directory of a primary partition of the card.

Administration and recovery functionality is required for updating the kernel image or reformatting the card in case of a damaged card file system. Thus, the boot loader has to be extended by routines, which provide the capability to write-access the cards and preserve the integrity of the FAT32 file system while operating on it.
It is also desirable to provide the opportunity to select between different kernel images to boot. This is of special interest for development and testing purposes, since kernel images of different configuration can be swapped in a straightforward manner. The ability to choose between the image files requires a configurable image path not only when loading, but also when storing a kernel image.

8 Conclusion and Outlook

Designed as fast trigger detector, the *ALICE TRD* (*Transition Radiation Detector*) serves to select only particle collisions of major interest to the main purpose of research. As high-level part of the TRD front-end electronics, the *Global Tracking Unit* (*GTU*) is responsible for reconstructing particle tracks and transversal momenta in order to form a contribution to the L1 trigger decision. The GTU also buffers the entire event data of the TRD and forwards it to the *Data Acquisition System* and the *High-Level Trigger*.
As one of the most complex systems of the TRD readout chain, the GTU requires continuous monitoring and control. The Xilinx Virtex-4 FX100 FPGA devices used in the GTU provide embedded PowerPC 405 processor cores, which can be used in a software-hardware co-design to administrate the system. While the design is currently provided with standalone software of limited flexibility, it is desirable to have an embedded system, which manages system administration in a more flexible and expandable way.

Within the scope of this diploma thesis, the *Embedded Linux System* and the necessary hardware components were developed for the GTU. The Embedded PowerPC System was designed to serve as the basic hardware platform for the Linux kernel. It is capable of interfacing with the GTU system components. To provide the necessary infrastructure for storing and booting the Linux kernel from the SD Memory Cards installed, an *SD Memory Card Controller* and the *GTU Boot Loader* were developed. A substantial aspect of the development of the *Embedded Linux kernel* was the support for custom-designed embedded platforms such as the GTU is.

The Embedded PowerPC System developed in this thesis provides essential system components such as the processor core, memory controller and communication interface, as well as custom-developed interfaces to attach system components of the GTU. In order to provide flexible interfacing capabilities despite the variety of different GTU system components, the custom interfaces have a parameterizable design.

The design of the SD Memory Card Controller provides a resource-saving performance-optimized implementation without limiting compatibility with other memory devices based on flash technology. Initialization and status control of the card is done by a hardware-software combination, while performance-relevant read and write transactions are implemented fully in hardware. A dual-ported BRAM serves as buffer for the data exchanged between host and card. A *bank switching mechanism* divides the BRAM into four independent buffers and provides the opportunity to prepare data blocks while the

card simultaneously processes others. Compatibility with other flash-based memory devices is preserved by using the *SPI communication protocol*. Moreover, the controller has a generic interface to be usable with various host designs. It is not fixed to a specific clock frequency, but implements a *clock domain crossing* between host and SD Memory Card. Additional interfacing logic allows to attach the controller to the *Processor Local Bus* of the PowerPC. The present implementation of the SD Memory Card Controller provides reliable transactions with bit error rates less than 10^{-12} and is integrated into the GTU hardware design currently in use at *CERN*.

The GTU Boot Loader (GBL) is responsible for copying the Linux kernel image from the SD Memory Cards to main memory and starting the kernel boot process. It must also provide administration and recovery capabilities to store new kernel image files on the cards or reformat a card in case of a damaged file system. The cards are accessed by the SD Memory Card Controller and formatted with a *FAT32 file system*. The GBL is supplied with a set of basic routines designed to access the file system based on low-level controller software I/O routines.

The development of the Embedded Linux System required support for the custom-developed hardware design without the need of adapting kernel sources manually to modifications of the hardware design. By combining specific parts of different embedded Linux distributions a kernel buildchain could be developed, which provides an autoconfiguration mechanism to integrate hardware design parameters into the kernel configuration. The Embedded Linux System developed represents an operating system ready to use as a platform for the deployment of a flexible and easily expandable system to monitor and administrate the GTU. It has shown stable operation for more than 80 days on various GTU system boards.

By the time of writing this thesis, the fundamental parts of an embedded administration and control system for the GTU are available. Thoroughly tested, the system is ready to be extended for the specific functionality of the GTU system software for a forthcoming use at *CERN*.
However, there are several extensions to improve the performance and capabilities of the system. The next step could be the implementation of a *Multi-Gigabit Ethernet connection* to provide faster communication with the GTU boards. To use the Multi-Gigabit Ethernet connection, the GBL must be extended by a TCP/IP stack. Its administration and recovery capabilities also need to be augmented. In order to integrate the functionality of the GTU system software into the Embedded Linux System, corresponding *device drivers* for accessing the custom hardware components such as the *BRAM interfaces*, the *SRAM controller interface* and the *SD Memory Card Controller interface* are needed. Finally, the monitoring and control software, which replaces the current standalone GTU system software, needs to be ported.

Appendix A

Embedded PowerPC System and Petalinux

A.1 Example Configuration of EDK PowerPC Project

Integration of the *auto-config mechanism* of Petalinux into the Xilinx EDK hardware build flow requires to adapt the project files *powerpc.mss* and *powerpc_incl.make* of the Embedded PowerPC System. The auto-config mechanism allows to extract relevant information from the hardware design and stores it in the file *auto-config.in*, which is included into the Linux kernel configuration. In the following excerpts of the relevant parts of both project files are listed.

A.1.1 Excerpt from PowerPC MSS File

```
[...]
BEGIN OS
 PARAMETER OS_NAME = petalinux
 PARAMETER OS_VER = 1.00.b
 PARAMETER PROC_INSTANCE = ppc405_0
 PARAMETER TARGET_DIR = ""
 PARAMETER stdout = uartlite
 PARAMETER stdin = uartlite
 PARAMETER main_memory = ddr2_mpmc
 PARAMETER main_memory_bank = 0
END
[...]
```

The parameters `OS_NAME` and `OS_VER` are needed to associate the processor instance with the Petalinux operating system. To include the auto-config scripts, which extract the hardware system parameters, these parameters must be set to *petalinux* and *1.00.b*, respectively. The parameter `TARGET_DIR` specifies the output directory, where the *auto-config.in* file is stored in. Both `stdin` and `stdout` identify the standard I/O communication device of the embedded system. The Petalinux-related parameters `main_memory` and `main_memory_bank` are used to designate and configure the main memory instance.

A.1.2 Excerpt from PowerPC Makefile

```
[...]
XILINX_EDK_DIR = /opt/xilinx/edk9.1
SYSTEM = powerpc
MHSFILE = powerpc.mhs
MSSFILE = powerpc.mss
FPGA_ARCH = virtex4
DEVICE = xc4vfx100ff1152-11
LANGUAGE = vhdl
SEARCHPATHOPT = -lp $(PETALINUX)/hardware/edk_user_repository
SUBMODULE_OPT = -toplevel no -ti powerpc_i
PLATGEN_OPTIONS = -p $(DEVICE) -lang $(LANGUAGE) $(SEARCHPATHOPT) $(SUBMODULE_OPT)
LIBGEN_OPTIONS = -mhs $(MHSFILE) -p $(DEVICE) $(SEARCHPATHOPT)
[...]
```

The parameter `SEARCHPATHOPT` defines the library path for user peripherals and driver repositories. It has to be extended by *-lp $(PETALINUX)/hardware/edk_user_repository* and must be added to both `PLATGEN_OPTIONS` and `LIBGEN_OPTIONS` in order to make the Petalinux scripts accessible to the synthesis tools.

A.2 Petalinux Patches

This section provides a full list of the patches, which had to be created and applied to the Petalinux source files in order to get a flawless kernel build flow. The source code of the patches is located in the directory */petalinux-v0.20-rc3/patches* on the project DVD, which is attached to this thesis. In the following, a brief description of each of the patches is given.

<petalinux_dir>/settings.sh

The patch for the file *settings.sh* originally extends the executable search path `PATH` by the directory of the MicroBlaze toolchain, which comes with the Petalinux distribution. The patch adapts the file and adds the directory of the PowerPC buildchain to the search path instead.

<powerpc_toolchain_dir>/powerpc-linux-ld.sh

Because the options `-fatal-warnings` and `-Wl` are not provided by the GNU linker of the uClinux PowerPC toolchain as used, the kernel build flow aborts with a number of errors.

The script *powerpc-linux-ld.sh* bypasses this problem by calling the linker and omitting these options. The linker has to be renamed to *powerpc-linux-ld**.real***. A *symbolic link* must be created from *powerpc-linux-ld* to *powerpc-linux-ld.sh* in the buildchain directory.

<petalinux_dir>/software/petalinux-dist/user/mount/fstab.c

Some of the source files of Petalinux and uClinux are outdated and contain obsolete code fragments. This patch replaces the line `die (EX_USER, "%s", sys_siglist[sig])` with `die (EX_USER, "%s", strsignal(sig))`, because `sys_siglist` is no longer supported by the kernel sources.

<petalinux_dir>/software/petalinux-dist/linux-2.4.x/include/linux/in.h

The kernel sources are provided with two versions of *in.h*, both required by different applications. Since the header files are of similar content, multiple definitions of constants and data structures exist, causing the kernel build flow to fail. However, the definitions in the file *<petalinux_dir>/software/petalinux-dist/linux-2.4.x/include/linux/in.h* can be removed (or disabled in this case) to avoid further conflicts.

<petalinux_dir>/software/petalinux-dist/linux-2.4.x/arch/ppc/boot/simple/misc-embedded.c

In file *misc-embedded.c*, the kernel boot parameter string `char netroot_string[]` is initially set to *"root=/dev/nfs rw ip=on"* and causes the kernel to try and mount a network file system as root file system. Since there is no networking device available, the kernel fails to mount a root file system and stops. The patch defines the boot parameter as an empty string. Kernel parameters can be explicitly set in the kernel configuration.

<petalinux_dir>/software/petalinux-dist/linux-2.4.x/include/linux/nfs.h

The header file *nfs.h* contains a few lines of code required by the kernel, which is is enclosed by preprocessor directives checking for the definition of the symbol `__KERNEL__`. Because this symbol is not defined, the code is omitted during compilation and causes several errors. The patch for this file disables the preprocessor directives.

<petalinux_dir>/software/petalinux-dist/include/include-linux/linux/nfs_mount.h

The file *nfs_mount.h* lacks including the file <*linux/nfs2.h*>, which contains several essential NFS2 protocol definitions. As described in chapter 5, it is necessary to enable networking support in the kernel configuration, which depends on these definitions. The patch adds a line to include the necessary file.

<petalinux_dir>/software/petalinux-dist/user/mount/nfsmount.c

The file *nfsmount.c* includes the non-existing header file <*gnu/types.h*>. The corresponding line of code is disabled by the patch.

<petalinux_dir>/software/petalinux-dist/user/mount/nfsmount.h

As for the previous file, *nfsmount.h* includes the non-existing header file <*gnu/types.h*>. Additionally, it lacks the constant `NFS_VERSION 3` required by the drivers for networking support. The patch adds the required constant and disables the inclusion of the non-existing header file.

<petalinux_dir>/software/petalinux-dist/Makefile

This file is the top-level Makefile of the Petalinux build flow and contains the necessary build targets. However, the build flow to generate the *U-Boot* boot loader is flawed and not needed in this context. Thus, the `u-boot` target it is removed from the build target `all` specified in the beginning of the Makefile. The non-existing *TFTP*[1] output directory is defined as a root directory (`TFTPDIR = /tftpboot`), but cannot be created without root permission. Furthermore, a number of environment variables are not properly set. The patch fixes the errors in a corresponding manner.

<petalinux_dir>/software/petalinux-dist/linux-2.4.x/Makefile

The PERL script, which calculates the kernel dependencies (*depmod.pl*), is called with flawed command line options. The patch adapts the Makefile to pass the correct arguments to the script.

[1] TFTP: Trivial File Transfer Protocol. A very basic form of FTP. It is neither capable to list directory contents, nor does it provide authentication or encryption mechanisms.

<petalinux_dir>/software/petalinux-dist/vendors/Xilinx/powerpc-auto/Makefile

A number of Makefile directives begin with a tabulator space, which causes them to be considered as a command for a rule. The Makefile also tries to copy the kernel image files to a directory, which does not exist and can not be created without root permissions (*/tftpboot*). Furthermore, a wrong installation path of kernel modules is given. The patch fixes the errors by removing the tab spaces, disabling the copy command and adjusting the module installation path.

Appendix B

SD Memory Card Controller

B.1 SD Memory Card Configuration and Status Registers

SD Memory Cards provide a set of status and configuration registers, which hold relevant information about the capabilities and operation conditions of the card. Table B.1 gives an overview of the card registers.

Name	Width [Bits]	Description
CID	128	Card Identification Number
RCA	16	Relative Card Address (not available in SPI mode)
DSR	16	Driver Stage Register
CSD	128	Card Specific Data
SCR	64	SD Configuration Register
OCR	32	Operation Condition Register
SSR	512	SD Status Register
CSR	32	Card Status Register

Table B.1: Configuration and status registers of an SD Memory Card, which contain relevant information about the capabilities and operation conditions of the card. Source: [Tec06]

`CID` stores an individual number to identify the card. When operating in SD mode, the local system address of a card is held by `RCA`. This address is dynamically suggested by the card and approved by the host during initialization. It is used for the addressed host-card communication after the card identification procedure. The `DSR` register is used to configure the output drivers of the card. Information regarding the operation conditions of the card is stored in `CSD`. In addition to the `CSD` register, `SCR` provides information about special features of the card. The register `OCR` stores operation conditions such as voltage profile or capacity status of the card. `SSD` provides information about the card proprietary features. Finally, the `CSR` register holds the actual card status. A detailed description and example settings are given in [Tec06].

B.2 Control and Status Registers of the PLB Interface

The interface to attach the SD Memory Card Controller to the PLB of the PowerPC processor provides an 8-bit control register and an 8-bit status register.
The control register CTRL is used to initiate both commands transmitted by the host (USR commands) and read/write operations (FSM commands). In case of an FSM command, the type of transaction (read or write) is defined by a corresponding bit of this register. Furthermore, the register enables write access to the BRAM, which buffers the data transmitted between card and host.
The status of both controller and card is captured in the STAT register. It combines the handshaking signals of the controller as well as information about the card presence and protection status[1]. The status registers also captures the busy state of the controller. The assignment of the individual bits of both registers is summarized in table B.2 below.

Register	Bit position	Description
CTRL_REG	[7:6]	Bank index
	5	Reserved
	4	Acknowledge signal NO_CRC_ERR_ACK
	3	Reserved
	2	Type of operation (0: w, 1: r) (USR_SD_RNW)
	1	Initiate r/w operation (USR_SD_RW_INIT)
	0	Initiate user command (USR_SD_CMD_INIT)
STAT	7	Card busy (USR_SD_BUSY)
	6	No CRC error detected (USR_SD_NO_CRC_ERR)
	[5:4]	Reserved
	3	FSM command (r/w) acknowledge (USR_SD_RW_ACK)
	2	User command acknowledge (USR_SD_CMD_ACK
	1	Write protected (SD_WRITE_PROT)
	0	Card present (SD_PRESENT)

Table B.2: Assignment of the individual bits of the PLB interface control and status register to the corresponding interfacing signals of the SD Memory Card Controller.

B.3 Timing Values of Single Block R/W Transactions

The timing values for both single block read and single block write access to an SD Memory Card are listed in table B.3 below. The values are given in units of eight clock cycles, because data transmitted in SPI mode is byte-aligned to the $\overline{CS}$ signal.

[1] Card presence and protection status are retrieved via I^2C bus and routed through the controller.

Value	N_{CS}	N_R	N_W	N_{CR}	N_{RE}	N_{AC}	N_{WR}	N_{DB}	N_{DR}
Min	0	6	6	1	2	1	1	514	1
Max	-	6	6	8	2	See below	-	514	-

Table B.3: Timing values of single block read and write operations, given in units of eight clock cycles. The maximum read access time is calculated by the host as follows: $N_{AC}(max) = 100((TAAC \cdot f_{PP}) + (100 \cdot NSAC))$, where f_{PP} is the interface clock rate, and $TAAC$ and $NSAC$ are given in the *CSD* register.

$N_{\overline{CS}}$ defines the number of cycles the $\overline{CS}$ signal must be asserted before a transaction can be initiated. N_R and N_W correspond to the time required to send a read and write command token, respectively. N_{CR} is the number of cycles between the end of the command token and the beginning of a response token. The response token requires N_{RE} cycles to be transmitted by the card. In case of a read operation, N_{AC} and N_{DB} specify the time to access and transmit a data block token, respectively. When performing a write operation, the number of cycles between the end of the response token and the start of the transmission of a data block token is given by N_{WR}. N_{DR} is the time the card requires to write a data block including the variable number of clock cycles the card is busy to store the data.

Appendix C

Boot Loader

C.1 Master Boot Record and Partition Table

On partitioned data storage devices, which follow the BIOS partitioning scheme common to personal computers, the first logical data block (or sector) is called the Master Boot Record (MBR). Information about the individual primary partitions such as start address and length is stored in the partition table, which resides in the MBR. Further, the first 440 bytes of the MBR are reserved to host the code of a boot loader. The structure of an MBR is shown in table C.1.

Address	Size [Bytes]	Description
0x0000	440	Boot loader executable
0x01B8	4	Disk signature (optional)
0x01BC	2	Null (0x0000)
0x01BE	64	Table of primary partitions (four 16-byte entries)
0x01FE	2	MBR signature (0xAA55, little endian)

Table C.1: Structure of a Master Boot Record. The first 440 bytes host the boot loader (or part of it in case of a multi-stage boot solution). The partition table consists of four table entries, each 16 bytes in size and referring to one primary partition. The table is located at offset 0x01BE . The last two bytes keep the signature to identify the MBR.

The partition table consists of four table entries, each keeping information about one of four primary partitions of the storage medium. Table C.2 shows the structure of a partition table entry. Most relevant are the Cylinder-Head-Sector (CHS) fields, which specify both the first and last sector in the partition. The coding scheme of a CHS field is given in table C.3. A partition table entry also stores the partition type and the number of sectors in the partition. The status field marks a partition as bootable (0x80) or non-bootable (0x00).

The CHS addressing scheme was an early method for addressing each physical data block on a hard disk drive. It is based on the physical geometry of hard disk drives,

Offset	Size [Bytes]	Description
0x00	1	Status
0x01	3	Cylinder-Head-Sector (CHS) address of first partition sector
0x04	1	Partition type
0x05	3	CHS address of last sector in partition
0x08	4	Logical Block Address (LBA) of first sector in partition
0x0C	4	Number of sectors in partition

Table C.2: Structure of a partition table entry. The status field marks a partition as bootable 0x80 or non-bootable 0x00 . Both start and end address as well as the size of the partition are stored in dedicated fields of a table entry.

which are divided into heads, cylinders (tracks) and sectors and is still used for floppy disks. However, CHS has been replaced by the *Logical Block Address* (LBA) addressing scheme. Data blocks are successively addressed by an index, with the first data block being LBA=0. The translation from CHS to LBA is given by $LBA = (C \cdot H + h) \cdot S + s - 1$, where:

- LBA: Logical Block Address
- c: Cylinder number
- H: Number of heads
- h: Head number
- S: Number of sectors
- s: Sector number

Bit position	[23:16]	[15:14]	[13:8]	[7:0]
Description	Cylinder bits [7:0]	Cylinder bits [9:8]	Sector	Head

Table C.3: Cylinder-Head-Sector entry.

Bibliography

[A+08] AMSLER, C. ET AL., **2008**. *Review of Particle Physics*. *Physics Letters B667*, 1:1+. URL `http://pdg.lbl.gov`

[AAA+06] ALICE, ALESSANDRO, B., ANTINORI, F., BELIKOV, J. A., BLUME, C., DAINESE, A., FOKA, P., GIUBELLINO, P., HIPPOLYTE, B., KUHN, C., MARTINEZ, G., MONTENO, M., MORSCH, A., NAYAK, T. K., NYSTRAND, J., NORIEGA, M. L., PAIC, G., PLUTA, J., RAMELLO, L., REVOL, J.-P., SAFARIK, K., SCHUKRAFT, J., SCHUTZ, Y., SCOMPARIN, E., SNELLINGS, R., BAILLIE, O. V. and VERCELLIN, E., **2006**. *ALICE: Physics Performance Report, Volume II*. *Journal of Physics G: Nuclear and Particle Physics*, 32(10):1295–2040. URL `http://stacks.iop.org/0954-3899/32/1295`

[ALI] ALICE COLLABORATION. *Layout of the ALICE Detector*. URL `http://aliceinfo.cern.ch/public/en/Chapter2/Chap2Experiment-en.html`.

[ALI01] ALICE COLLABORATION, **2001**. *ALICE Technical Design Report of the Transition Radiation Detector*. Tech. Rep. CERN/LHCC 2001-021, CERN, Geneva.

[Alt02a] ALTERA CORPORATION, **2002**. *Excalibur Device Overview*. Version 2.0.

[Alt02b] ALTERA CORPORATION, **2002**. *Excalibur Hardware Reference Manual*. Version 3.1.

[Com] COMPANY, K. T.

[dC] DE CUVELAND, J. *Online Track Reconstruction of the* ALICE *Transition Radiation Detector at* LHC *(CERN)*. Ph.D. thesis, University of Heidelberg, Kirchhoff-Institut for Physics, Heidelberg. Publication planned.

[dC03] DE CUVELAND, J., **2003**. *Entwicklung der globalen Spurrekonstruktionseinheit für den ALICE-Übergangsstrahlungsdetektor am LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[Gut02] GUTFLEISCH, M., **2002**. *Digitales Frontend und Preprozessor im TRAP1-Chip des TRD-Triggers für das ALICE-Experiment am LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[Gut06] GUTFLEISCH, M., **2006**. *Local Signal Processing of the ALICE Transition Radiation Detector at LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[IBM06] IBM, **2006**. *PowerPC 405 CPU Core Product Overview*. IBM Corporation.

[JGM] J GORBAN, I. M. and MARKOVIC, T. *Opencores: uart16550*. URL `http://www.opencores.org/projects.cgi/web/uart16550/overview`.

[Kir07] KIRSCH, S., **2007**. *Development of the Supermodule Unit for the ALICE Transition Radiation Detector at the LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[MS86] MATSUI, T. and SATZ, H., **Oct. 1986**. *J/ψ suppression by quark-gluon plasma formation*. *Physics Letters B*, 178:416–422.

[Ret] RETTIG, F. *Development of a Di-Lepton and Jet Trigger for the ALICE Transition Radiation Detector at LHC (CERN)*. Ph.D. thesis, University of Heidelberg, Kirchhoff-Institut for Physics, Heidelberg. Publication planned.

[Ret07] RETTIG, F., **2007**. *Entwicklung der optischen Auslesekette für den ALICE-Übergangsstrahlungsdetektor am LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[SAM05] SAMSUNG, **2005**. *512Mb C-die DDR2 SDRAM Specification v1.4*. SAMSUNG Electronics Co.

[San04] SANDISK, **2004**. *SanDisk SD Card Product Manual, Version 2.2*. SanDisk Corporation.

[Sch07] SCHUH, M., **2007**. *Entwicklung und Implementierung eines Steuersystems für die Trigger- und Datenausleselogik des ALICE-Übergangsstrahlungsdetektors am LHC (CERN)*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[Sch08] SCHNEIDER, R., **2008**. *Entwicklung des Triggerkonzepts und die entsprechende Implementierung eines 200-GB/s-Auslesenetzwerks für den ALICE-Übergangsstrahlungsdetektor*. Ph.D. thesis, University of Heidelberg, Kirchhoff-Institut for Physics, Heidelberg.

[Sin03] SINSEL, A., **2003**. *Linuxportierung auf einen eingebetteten PowerPC 405 zur Steuerung eines neuronalen Netzwerks*. Diplomarbeit, Universität Heidelberg, Kirchhoff-Institut für Physik, Heidelberg.

[Stö00] STÖCKER, H., **2000**. *Taschenbuch der Physik*. 4., korrigierte Aufl.,. Thun; Frankfurt am Main. ISBN 3-8171-1628-4.

[Tec06] TECHNICAL COMMITTEE, **2006**. *SD Specifications Part I, Physical Layer, Simplified Specification, Version 2.00*. SD Card Assostiation.

[Wil06] WILLIAMS, J., **2006**. *uClinux-dist Build Environment for Xilinx PowerPC Devices*.

[Xil07a] XILINX, **2007**. *Embedded System Tools Reference Manual v9.2i*. Xilinx, Inc.

[Xil07b] XILINX, **2007**. *Platform Specification Format Reference Manual*. Xilinx, Inc.

[Xil07c] XILINX, **2007**. *PowerPC 405 Reference Guide v1.2*. Xilinx, Inc.

[Xil07d] XILINX, **2007**. *SVF and XSVF File Formats for Xilinx Devices v2.0*. Xilinx, Inc.

[Xil08a] XILINX, **2008**. *Multi-Port Memory Controller (MPMC) v4.02.a*. Xilinx, Inc.

[Xil08b] XILINX, **2008**. *PowerPC 405 Block Reference Guide v2.3*. Xilinx, Inc.

[Yag03] YAGHMOUR, K., **2003**. *Building Embedded Linux Systems*. O'Reilly, 1st edition ed. ISBN 0-596-00222-X.

Danksagung Ich danke besonders herzlich meinem betreuenden Professor, Herrn Prof. Dr. Volker Lindenstruth, für die interessante Aufgabenstellung und seine engagierte Betreuung. Ebenso gilt besonderer Dank "meinem Doktoranden" Felix Rettig, der mich in vieler Hinsicht unterstützt hat. Die Zusammenarbeit mit ihm hat mich sehr viel gelehrt. Mein Dank gilt auch Florian Painke für seinen fachlichen Rat, mit dem er mir stets zur Seite stand. Dr. Venelin Angelov, Dirk Hutter sowie Stefan Kirsch danke ich für ihre Hilfsbereitschaft, besonders in den letzten Wochen.
Mein herzlichster Dank gilt Christina Schneller, die während dieser Zeit besonders für mich da war und mir ihre volle Unterstützung bot. Ebenso danke ich meinen Eltern Heinrich und Waltraud Gerlach, die mir dieses Studium ermöglicht haben.

...as my guitar lies bleeding in my arms!
My Guitar Lies Bleeding In My Arms (Bon Jovi, 1995, "These Days")

Erklärung zur selbständigen Verfassung

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, im September 2008.

Thomas Gerlach