The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Automated Gateware Discovery Using Open Firmware

Shanly Rajan

Supervisor: Prof. M.R. Inggs

Co-supervisor: Dr M. Welz

University of Cape Town

Declaration

I understand the meaning of plagiarism and declare that all work in the dissertation, save for that which is properly acknowledged, is my own. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author . . . . Signed by candidate

Cape Town

South Africa

May 12, 2013

University of Cape Town

Abstract

This dissertation describes the design and implementation of a mechanism that automates gateware[1] device detection for reconfigurable hardware. The research facilitates the process of identifying and operating on gateware images by extending the existing infrastructure of probing devices in traditional software by using the chosen technology.

An automated gateware detection mechanism was devised in an effort to build a software system with the goal to improve performance and reduce software development time spent on operating gateware pieces by reusing existing device drivers in the framework of the chosen technology.

This dissertation first investigates the system design to see how each of the user specifications set for the KAT (Karoo Array Telescope) project in [28] could be achieved in terms of design decisions, toolchain selection and software modifications. The final design satisfies the user specifications by treating the gateware programmed on reconfigurable hardware just like any other physical device attached to the system, extending the device database available to bootloader, mapping kernel device driver to operate on the gateware programmed and allowing the user to run suitable applications based on the personality of the gateware image programmed. The system implementation is then described and issues related to the process of integrating gateware, bootloader and kernel interfaces are discussed. The results of tests conducted on the actual hardware demonstrating the overall concept are presented. Conclusions are drawn based on these results and suggestions for future work and design improvements are recommended.

[1]Gateware is a University of California, Berkeley coined term for design logic that goes into the FPGA

University of Cape Town

To The Lord Almighty,

my family, friends

for all their support

and guidance. If not for all of you,

I would not be the man I am today.

University of Cape Town

Acknowledgements

I would like to thank the following parties for their assistance and involvement in the completion of this project:

- My supervisor, Prof Mike Inggs who believed in me as a professional and gave me the freedom and space to research at my own pace. A special thanks for his valuable time for reading through the draft of this dissertation and recommending corrections.
- My cosupervisor, Dr Marc Welz who has been instrumental in the success of this research. His valuable advice, steadfast support and kind words has been of immense value during the tenure of this research. A special thanks for his time for reading through and recommending corrections to the draft of this dissertation.
- Dr Alan Langman for providing the initial support and advice for the research.
- The SKA / KAT bursary programme for providing funding for this research.
- The entire DBE Team which i regard and respect as a family, ably led by Mr Francois Kapp. A special thanks to Mr Andrew Martens and Mr David George for assisting me with inputs for the design stages.
- My family and friends for their love and encouragement throughout the project.
- My wife, Bifin Shanly, who has been very instrumental in providing a peaceful family atmosphere so that i could focus on my research.

University of Cape Town

Contents

Declaration i

Abstract ii

Acknowledgements iv

Nomenclature xiii

1 Introduction 1

1.1 Background of Investigation 1

1.2 Project Motivation 2

1.3 Project Scope 2

1.4 Objectives 3

1.5 Dissertation Overview 3

2 Background 9

2.1 Device detection in conventional computer platforms 10

2.1.1 PCI (Peripheral Component Interconnect) 10

2.1.1.1 Brief history 10

2.1.1.2 Overview 10

2.1.1.3 Device Detection 12

2.1.2 ACPI (Advanced Configuration and Power Interface) 12

University of Cape Town

2.1.2.1 Brief history 12
2.1.2.2 Overview 13
2.1.2.3 Device Detection 14
2.1.3 OF (Open Firmware) 15
2.1.3.1 Brief history 15
2.1.3.2 Overview 15
2.1.3.3 Device Detection 17
2.1.4 Summary 18
2.2 Device detection in FPGA-based software systems 20
2.2.1 Related Studies 20
2.2.2 Linux 21
2.2.2.1 Background 21
2.2.2.2 Overview 22
2.2.2.3 Features 23
2.2.2.4 Device Detection in Linux 24
2.2.3 BORPH 24
2.2.3.1 Background 24
2.2.3.2 Overview 24
2.2.3.3 Features 26
2.2.3.4 Device Detection in BORPH 27
2.2.4 Summary 27
2.3 Chapter Summary 29

University of Cape Town

3 Design **31**

3.1 Design Constraints 32

3.2 Design Choices 32

3.3 Design Methodology 35

3.4 System Architecture 36

3.5 System Design 38

3.5.1 System Overview 38

3.5.1.1 System Connection Diagram 38

3.5.1.2 AMCC PowerPC Architecture 39

3.5.2 FPGA Programming Design 41

3.5.2.1 From U-Boot 44

3.5.2.2 From linux 45

3.5.3 Gateware Design 46

3.5.3.1 Gateware Toolflow 46

3.5.3.2 Gateware Support 47

3.5.4 Device Detection Design 52

3.5.4.1 From U-Boot 53

3.5.4.2 From Linux 55

3.5.5 Design Optimisations 58

3.5.5.1 Memory mapped device driver 58

3.6 Chapter Summary 59

4 Implementation and Results **61**

4.1 Embedded Development Setup 62

4.2 High level implementation block diagram 64

4.3 Gateware implementation 65

University of Cape Town

4.3.1 Serial loopback implementation 66

4.3.2 Simple data capture implementation 67

4.4 Device description 69

4.5 Device discovery and enumeration 71

4.6 Device operation and control 75

4.6.1 Device Drivers 75

4.6.2 Userspace Applications 79

4.7 Chapter Summary 81

5 Conclusions and Future Directions **83**

5.1 Summary 83

5.2 Conclusions 83

5.3 Dissemination Strategy 84

5.4 Recommendations for future work 84

A Source Code **86**

B ROACH Board Connections **87**

C DTC Compiler Usage **89**

Bibliography **90**

University of Cape Town

List of Figures

1.1 Device detection overview 5

2.1 Layout of a typical PCI system [23] 10

2.2 PCI configuration header [23] 11

2.3 ACPI system architecture 13

2.4 ACPI device detection 14

2.5 Open Firmware internal and external interfaces 16

2.6 Open Firmware probe and detection 17

2.7 Recent operating systems for FPGAs [32] 20

2.8 Linux kernel overview [23] 22

2.9 High level overview of GNU / Linux [27] 23

2.10 High level block diagram of BORPH [16] 25

2.11 BORPH [5] 26

2.12 Results of test conducted between BORPH and Linux device driver with memory mapped support 28

3.1 Linux and BORPH operating system flow diagram 33

3.2 Design steps to reach the final system 35

3.3 Automated gateware discovery system architecture 37

3.4 System connectivity diagram 38

3.5 PPC440EPx functional block diagram [4] 40

University of Cape Town

3.6 Slave configuration modes [44] 41

3.7 SelectMAP Configuration Flow Chart 42

3.8 FPGA-PowerPC SelectMAP programming interface 43

3.9 Programming FPGA from U-Boot using SelectMAP interface 44

3.10 Character device driver for FPGA programming 45

3.11 MSSGE toolflow diagram for CASPER hardware 46

3.12 UARTLite serial OPB core connectivity in EDK 48

3.13 Graphical representation of sound card design using iADC in Simulink . . 50

3.14 Graphical representation of PPC440EPx device tree 52

3.15 Graphical representation of ROACH device tree 54

3.16 Device tree usage and detection in Linux 56

3.17 Memory mapping FPGA 58

4.1 Cross-development setup for ROACH 62

4.2 U-Boot port changes for ROACH2 63

4.3 High level implementation block diagram of automated gateware discovery mechanism using *OF* 64

4.4 Gateware connections to PowerPC, buses and FPGA 65

4.5 Serial design running on PowerPC and FPGA 66

4.6 PowerPC-FPGA interface for sound capture design. 67

4.7 Lab setup of the experiment 68

4.8 Test setup for the experiment 68

4.9 ROACH2 device tree snapshot 69

4.10 ROACH and ROACH2 Device Tree Entries 70

4.11 device tree compiler usage 71

4.12 U-Boot on ROACH2 72

University of Cape Town

4.13 FDT blob listing in U-Boot 73

4.14 Adding a device dynamically on FDT blob from U-Boot 74

4.15 Creating required character device files 75

4.16 UARTlite device driver log from Linux 76

4.17 Audio file operations structure 77

4.18 Aumix operation device driver log 78

4.19 Strace sample output 79

C.1 dtc compiler usage 89

University of Cape Town

List of Tables

2.1 Comparison of various hardware device detection mechanisms 19

2.2 Short history of Linux 22

2.3 Reconfigurable software comparison 29

3.1 Supported slave modes in ROACH 41

3.2 SelectMAP signal functional description 42

3.3 UARTLite register & address Map [42] 49

3.4 iADC sound card register & address map [1] 51

3.5 Device tree compiler formats 54

3.6 Open Firmware platform identification functions 56

4.1 Audio device driver events and associated functions 77

4.2 Summary table of the *OF* implementation done for automating gateware detection 80

University of Cape Town

Nomenclature

ACPI — Advanced Configuration and Power Interface. Open industry standard for device configuration and power management by Operating System.

ADC — Analogue to Digital Converter. Device that converts analog data to digital samples.

AML — ACPI Machine Language. Intepreters convert the plain source format, ASL into a machine recognisable format which is AML.

ASL — ACPI control method Source Language. Programmers use this source format to write definition blocks required for ACPI.

APM — Advanced Power Management. A BIOS based system power manager.

BIOS — Basic Input-Output System. ROM based software for basic configuration of computer.

Bitfile — Compiled design logic used to configure an FPGA.

Blob — Binary form of flattened device tree that gets passed to the kernel. (Also referred to as FDT blob).

Boffile — Compiled design logic with bof header containing register mapping of gateware design used to configure an FPGA and generate /proc entries.

BORPH — Berkeley Operating system for ReProgrammable Hardware. An operating system for FPGA-based reconfigurable computers.

CASPER — Center / Collaboration for Astronomy Signal Processing and Electronics Research. Open source collaborative community with focus on radio astronomical applications.

Device Tree — A representation that describes the systems hardware devices and their interconnections.

DSDT — Differentiated System Description Table. Important table that contains definition blocks required for ACPI.

University of Cape Town

DTB — Device Tree Binary. The compiler converts the source written, DTS to kernel required format namely DTB.

DTS — Device Tree Source. The textual format which programmers use to describe a device.

EDK — Embedded Development Kit. Compiles the generated low level code into a bitstream that runs on the targeted FPGA.

FDT — Flattened device tree. Device tree format that Linux kernel recognises.

FPGA — Field Programmable Gate Array. A semiconductor device capable of synthesizing complex logic designs.

FOCA — FPGA Operating system for Circuit Autoconfiguration. Hardware operating system specified with a set of VHDL processes for reconfigurable computers.

FreeBSD — Free Berkeley Software Distribution. Another free Unix-like operating system initiative.

Gateware — Digital design logic implemented on FPGA.

GATOS — Gate Array Terminal Operating Systems. Interactive windowing operating system with a graphical user interface.

GNU — GNUs Not Unix. A project founded by Richard Stallman with aim to provide free and quality software.

IADC — ADC board used on BEE2s and IBOBs, reconfigurable hardware boards built at University of California, Berkeley..

KATADC — ADC board custom design and built by KAT.

Matlab — A scriptable back-end for Simulink written in the Matlab language.

MSSGE — Matlab/Simulink/System Generator/EDK. Toolflow which stitches together several design and implementation environments for generating FPGA based designs.

OF — Open Firmware. Hardware independent boot code.

OSPM — Operating System directed Power Management.

PCI — Peripheral Component Interconnect. Standard for describing hardware connections in a system in a structured way.

PowerPC — Performance Optimized With Enhanced RISC Processor Chip. Popular RISC architecture in embedded systems and high performance processors.

University of Cape Town

REConfigME — Another operating system for reconfigurable computing.

RISC — Reduced Instruction Set Computing.

RCs — Reconfigurable Computers.

ROACH — Reconfigurable Open Architecture Computing Hardware. FPGA based computing board.

ROACH2 — Next generation ROACH hardware board built by KAT.

Simulink — Graphical programming tool that has both schematic capture tool and a design simulation environment for system models targeted for CASPER FPGA boards.

System Generator — Translates Simulink schematics into low level hardware code (either VHDL or Verilog) during design compilation. It also enables design simulation from within the Simulink environment.

SLOC — Source Lines Of Code. Number of lines in the kernel source code.

U-Boot — Universal bootloader for a number of different computer architectures.

University of Cape Town

Chapter 1

Introduction

This dissertation presents the design of *automated gateware*[1] *discovery mechanism* useful for radio astronomy applications using generic reconfigurable computing hardware and toolflows. In this introduction, we provide a brief background and motivation for this topic, list objectives of the dissertation, and outline the contents of the dissertation.

1.1 Background of Investigation

FPGAs (Field Programmable Gate Arrays) are semiconductor devices containing programmable logic components and programmable interconnects that can be reconfigured many times with different functions (gateware images). At the most basic level they are arrays of logic gates whose functionality can be changed in software. A change in the functionality requires the user to essentially perform three sequential steps: change logic function, re-compile and program it onto the FPGA. Over the years there has been an increased use of hybrid FPGA-CPU architectures to speed up computationally intensive algorithms and problems—this constitutes the broad area called *"reconfigurable computing"*. The use of FPGAs in conjunction with microprocessors allows reconfigurable computers to offer much of the flexibility of a general-purpose computing architecture, but at the same time provide many of the performance benefits of having an algorithm implemented in a hard-wired chip. A recent example of hardware platforms that utilise the hybrid Xilinx FPGA-PowerPC architecture are the ROACH[2] (Reconfigurable Open Architecture Computing Hardware) series of hardware boards. They have been developed by the KAT (Karoo Array Telescope) project in conjunction with the Berkeley CASPER (Center for Astronomy Signal Processing and Electronics Research) group and NRAO

[1] Gateware is a University of California, Berkeley coined term for design logic that goes into the FPGA

[2] ROACH is a South African collaborative project in conjunction with University of California research group at Berkeley: CASPER (Center for Astronomy Signal Processing and Electronics Research) and NRAO (National Research Astronomy Observatory) group.

University of Cape Town

(National Radio Astronomy Observatory). Commonly used FPGA-based reconfigurable computers are managed by off-the-shelf operating systems like Linux. In a DSP pipelined approach, gateware design implementations like correlators and spectrometers (instruments) needs the right software (device drivers) to detect the function implemented and operate on it. For each gateware design implementation writing a piece of software for that particular piece of instrument turns out to be an arduous task.

1.2 Project Motivation

Reconfigurable hardware components which can be treated as peripherals to a conventional processor present certain challenges to the host operating system. In particular, it is a complex task for the operating system to auto-detect such devices and load appropriate drivers for operating the device. This is because reconfigurable hardware can be programmed with different gateware. Each gateware can implement substantially different functions thereby increasing the complexity of software support required by the Operating System. Therefore an alternate software approach that interacts with the gateware programmed on the FPGA is needed in order to auto-detect gateware designs and load the right software for operating the device.

University of Cape Town

1.3 Project Scope

With the increasing presence of hybrid FPGA-CPU architectures in the realm of reconfigurable computing, the goal of effectively and efficiently integrating traditional software with FPGA-based reconfigurable computers regains significance. It would be useful if automated gateware discovery mechanisms were able to facilitate the process of identifying and operating on gateware images by extending the existing infrastructure of probing devices in traditional software in a suitable manner. The mechanism devised in Chapter 3 would form part of the infrastructure required for probing gateware images. This mechanism would contribute to an alternate approach for identifying FPGA based instruments by making necessary modifications to the conventional hardware platform database.

Projects relying heavily on reconfigurable computing resources may find this mechanism useful in saving time and effort spend on implementing new ways to interact with FPGA designs. This project implementation facilitates the loading of the right software for corresponding gateware designs. This work is developed for the MeerKAT[3] project and can add value and support to the KAT DBE (Digital Back End), by reducing the effort and time spent in writing device driver for each piece of gateware generated. The hope is that

[3] a precursor instrument for the SKA (Square Kilometer Array)

the project may be useful to other radio astronomy projects like SKA and the infrastructure developed shall be sufficiently generic to be of use in other fields.

1.4 Objectives

The objective of this dissertation is to design and develop a mechanism which automates gateware detection for reconfigurable hardware designs and simplify the task of writing software for each of the gateware image programmed.

The requirements review was conducted upon the user requirements [28] generated for the KAT project by Dr Marc Welz and were refined and finalized by further discussions with Prof. Inggs and Dr Alan Langman.

The summarised user requirements are:

- Developing a mechanism where reconfigurable hardware can register its functions with conventional platform hardware databases.
- Integrating reconfigurable devices into platform system initialisation or hotpluggable frameworks.
- Extending the infrastructure for probing hardware devices to include suitably designed gateware images.

University of Cape Town

1.5 Dissertation Overview

This section briefly provides an overview of the dissertation on a chapter by chapter basis.

Chapter 2 provides an overview of the various mechanisms that exist to describe, detect and connect physical components of a system. The chapter is divided into two main sections: one section that reviews the device detection mechanisms used by conventional computer platforms and other section that reviews configuration mechanisms developed for FPGA based systems. A brief history, overview and device detection method of the corresponding mechanisms are explored. The associated terminologies are also introduced in the process. Each section is summarised with a table that compares the various device detection and configuration mechanisms reviewed. Based on the inputs from the concluded table, we choose a device detection mechanism from the table that meets the user requirements [28] outlined in Chapter 1. The reasons to choose *OF*[4](Open Firmware)

[4]Formerly endorsed IEEE standard that defines the interface of computer firmware system.

for accomplishing the goal of automating gateware detection on FPGA based reconfigurable hardware platforms are mentioned.

Chapter 3 describes a design for building a gateware detection system to use *OF*. Based on the *OF* concepts supplemented in Chapter 2, we will design the various stages that constitute an automated gateware discovery mechanism for Reprogrammable Hardware.

The chapter commences by stating design goal of the project and how to meet it using *OF*. The design choices made for the implementation of the system illustrated in Figure 3.3 are discussed in detail by drawing comparisons between a current approach used for software-FPGA interaction which is BORPH (Berkeley Operating System for ReProgrammable Hardware) and the proposed software-FPGA approach which uses linux with OF support. The benefits of such an approach undertaken is mentioned. Finer objectives are extracted keeping in mind the proposed software-FPGA approach considered and adapting the general objectives listed in *Section* 1.4 to technical objectives.

We move onto the design constraints that are imposed while implementing the project for MeerKAT DBE mainly regarding the choice of hardware platform and the choice of software used to control and command the FPGA designs implemented on the hardware platform. A system architecture diagram elaborating the design concept is illustrated in Figure 3.3. The various design stages of the diagram are explained below, from generating gateware to programming the FPGA to device detection at various levels of the software.

- **Gateware Design:** The CASPER[5] approach uses the MSSGE[6] toolchain consisting of Matlab, Simulink, System Generator and EDK to generate *gateware* for the FPGA. The various compilation stages of the toolflow are depicted in Figure 3.11. In simple terms, an application model file described in a graphical programming language called Simulink is passed through the MSSGE toolchain to generate bitstream, termed as *gateware* that gets programmed on the FPGA. FPGA changes its functionality as different pieces of gateware are programmed.

- **FPGA Programming Design:** PowerPC is the CPU of the ROACH hardware platform. It interacts with the FPGA using the *selectmap* interface. A brief introduction about this interface on ROACH and ROACH2 is provided using Figure 3.8. The FPGA can be programmed using *selectmap* interface at different software levels. At the most basic level, FPGA programming can be achieved by using JTAG tools. At the bootloader and kernel level we look into how selectmap interface can be used to program the FPGA.

[5]CASPER website:https://casper.berkeley.edu/

[6]Toolflow which stitches together several design and implementation environments for generating gateware designs

University of Cape Town

- **Device Detection Design:** This is the core design stage where we integrate Open Firmware components and concepts into our system architectural design. The gateware programmed on the FPGA is the device that gets represented in the form a *device tree* at the bootloader level. From the bootloader, the device tree gets converted into a kernel recognised format called the FDT (Flattened Device Tree) or *blob*.

The *OF* supported kernel unflattens the *blob* and uses *OF* platform initialisation routines and methods to probe and identify the devices represented in the *device tree.* The kernel loads the matching device drivers to operate on the device identified.

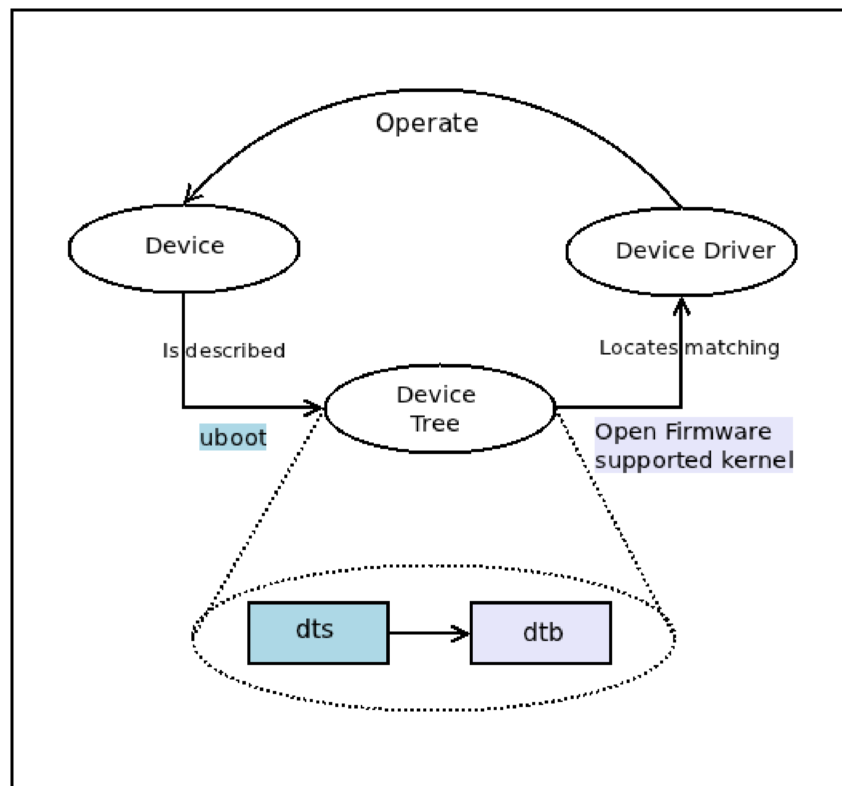


Figure 1.1: Device detection overview

Chapter 4 focuses on the implementation phase of the project, associated test setup and results obtained based on the design elaborated in Chapter 3. Some actual FPGA application designs are run to illustrate the various concepts of the design. The results of the tests performed are discussed to validate categories of gateware detection, loading correct device drivers to operate on it and extending *OF* infrastructure available for probing hardware devices. The testing methods are chosen considering the finer objectives enumerated in *Section* 3.2.

We start by preparing the design tools and environment needed for implementation of the project. The available MSSGE toolflow and U-boot[7] [7] bootloader were examined.

[7]Das U-boot is an open source, multi-platform bootloader used in embedded devices

Linux kernel with PowerPC and *OF* support was ported to ROACH, the hardware platform. The CPU, peripherals, buses and interconnects of ROACH are described in the form of a device tree and passed to the kernel for probing and bringing the board up and running with the appropriate device drivers. Kernel device drivers were studied in detail with sufficient examples as preparatory work in order to write custom device drivers for operating a device, in this case, gateware on FPGA.

With the first example, a serial loopback test, we attempt to establish the concept that gateware detection pieces programmed on the FPGA can be treated just like any other physical peripheral and further it can be operated on by an existing device driver that gets loaded by the kernel at run-time. For this purpose the chosen gateware implementation was a Xilinx UARTLite OPB serial core. The generated gateware is programmed. An entry is made in the device tree source describing the UARTLite serial gateware at the right node. The compiled blob is probed by the kernel at run-time and the corresponding UARTLite serial driver is loaded to operate on the piece of gateware logic residing in the FPGA, now being treated as a physical serial device. The interrupt routines were diabled for simplicity in design and for focusing on concept demonstration. The results were inline with the expectation that device trees can be extended to describe a piece of gateware just like any other physical peripheral.

With the simple data capture test example, we attempt to establish the concept that *OF* probe and detect infrastructure can be utilised to load custom device drivers for operating radio astronomy instruments programmed on the FPGA. Further we are aiming to use existing applications to display the data captured, thereby needing not to write special application software for streaming the captured data. In a typical digital signal processing chain, data is captured by ADC and channelled to storage location in FPGA and processing of data happens by reading out this data in chunks over the network. In this implementation, iADC[8] captures data and stores it in BRAM (Block RAM) in FPGA. We are aiming to use existing audio applications to adjust the gain of the ADC and play the data using the audio application interface. The custom built audio device driver for operating the data available in BRAM was written with the intention to integrate the new driver into the *OF* infrastructure and test the probing and detection capabilities of the *OF* framework upon recognising a new piece of similar gateware.

At this point of the project the following can be demonstrated:

- Gateware can be treated like any other physical peripheral residing in the hardware plaform.
- Gateware can be described in the device tree.

[8]ADC board used on BEE2s and IBOBs, reconfigurable hardware boards built at University of California, Berkeley

University of Cape Town

- The detection and loading of exisitng device drivers by the OS to operate on the corresponding gateware pieces is possible and demonstrated.
- The *OF* infrastructure initialisation and probing capabilites can be modified to detect and load custom device drivers for operating corresponding gateware.

Chapter 5 concludes the dissertation and provides insights into future research and areas of development. We outline the benefits and limitations of the proposed design using *OF* as technology framework for automating gateware detection. The conclusions are drawn from the test results obtained in Chapter 4.

University of Cape Town

Chapter 2

Background

This chapter reviews some of the common standards that can be used to describe and connect physical peripherals in a system and how device detection[1] is made possible through these standards. A system is constituted of several peripherals that are electrically connected in a logical manner to behave the way it should be. The operating system gathers information about resident physical devices made available through these standards. The chapter reviews these standards in two ways: From a technology point of view and from an operating system point of view.

The first section reviews device detection in conventional computer platforms. PCI, ACPI and OF are industry standards that have been widely used to make device autodetection possible in computing platforms. A brief history, overview and device detection mechanism of each standard is explored and summarised. A table is drawn comparing the parameters that are key to device detection for the reviewed standards.

Software support for reconfigurable computer that facilitates the end user to interact with hardware designs has been a research area for a long time. The second section reviews reconfigurable software that were developed for FPGA based systems. Some of the available software designed for FPGA based systems are listed in Fig 2.7. One of the commonly used operating system for FPGAs in the radio astronomy field, BORPH is discussed. Brief history of BORPH, overview and features are discussed. We elaborate the same for another widely used and familiar operating system, Linux.

The reasons to choose OF and linux for meeting the goal of automating gateware detection on FPGA based reconfigurable hardware platforms is concluded from the review done.

[1]Device detection is the process of identifying physical devices attached to a system and configuring it inorder to communicate with the device

University of Cape Town

2.1 Device detection in conventional computer platforms

2.1.1 PCI (Peripheral Component Interconnect)

2.1.1.1 Brief history

Work started on PCI in 1990s by Intel, with the intention of providing a high performance interconnect that allows faster transfer of data between computer and peripherals. It was introduced as a replacement to the ISA (Industry Specific Architecture)[2] standard. PCI came to existence in 1992 and it modelled as a standard that connected devices in a structured and logical way. It simplified the process of adding and removing peripherals to a system by supporting auto detection of interface boards and devices. Further it was designed to be as platform independent as possible and is used extensively on IA-32, Alpha, PowerPC, SPARC64, and IA-64 systems.

2.1.1.2 Overview

PCI is a mechanism designed to interconnect peripherals on a motherboard in a structured and controlled way. The PCI address space is partitioned into three namely: PCI IO, PCI Memory and PCI Configuration address space. The PCI IO space and memory space are 4GB in size. PCI IO and PCI Memory spaces are used by the devices to map their internal registers into these spaces allocated to them for further device specific operations. A 256 byte configuration data structure namely the PCI configuration register which aids the BIOS initialisation routines to device detection is the core of the PCI specification. It contains fields that are relevant for detection, status enquiry and control of a PCI device. The PCI initialisation routines access the devices configuration space, initialises the configuration registers and provide access for the device to IO and memory spaces.

University of Cape Town

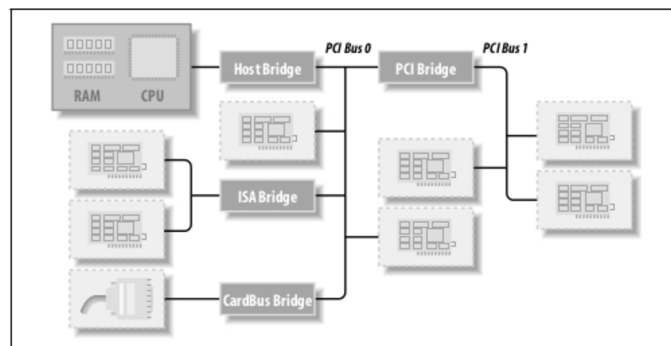


Figure 2.1: Layout of a typical PCI system [23]

[2]Computer bus standard for IBM PCs

Every PCI slot has its PCI configuration header in memory space in an offset thats related to the position of slot in the board. In other words, for each function contained within the PCI device, there is a dedicated configuration address space. The configuration register is 256 bytes long or 16 dwords long. Of this space, PCI configuration header occupies 16 bytes and the remaining 240 bytes is inhabited by the device specific configuration fields. There are three header formats that are currently defined in the PCI specification:

- Header Type Zero: Defined for all PCI devices excluding the bridges.
- Header Type One: Defined for PCI-to-PCI bridges.
- Header Type Two: Defined for CardBus bridges as illustrated in Fig 2.1.

PCI configuration header represented in Fig 2.2 consists of the following important fields:

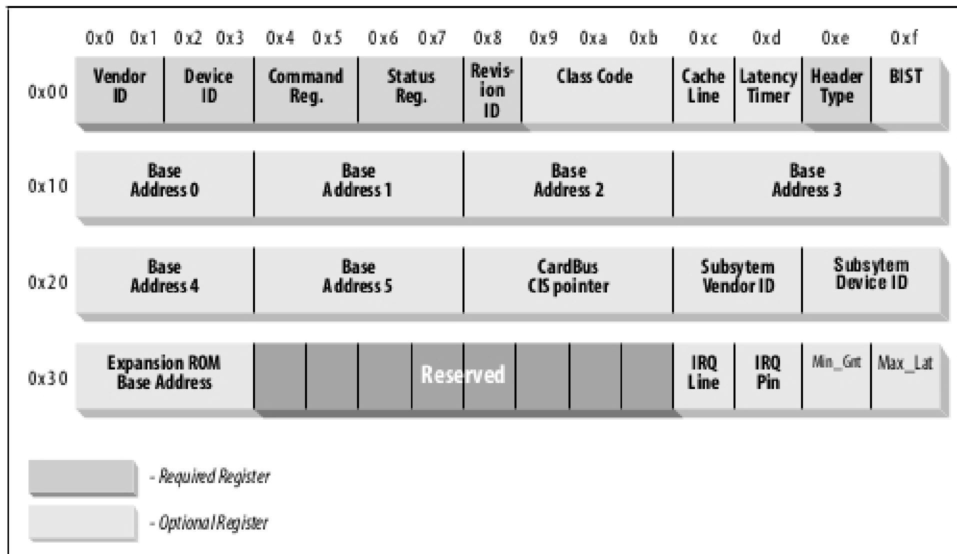


Figure 2.2: PCI configuration header [23]

- Vendor identification: Unique number that identifies the originator of PCI device. eg: Intel has vendor ID defined as 0x8086, Acer has a vendor ID of 0x0402.
- Device identification: Unique number that identifies the device. eg: A device ID of 0x9665 refers to Acer crystal eye webcam if its vendor ID is 0x04012.
- Status: Status of the device with each bit having special meaning as per specification.
- Command: The system controls the device by writing to this field, thereby allowing PCI IO and memory access to the corresponding PCI device.

- Class Code: Identifies the type of the device. eg: Class code of a SCSI device is 0x0100 and a IDE device is 0x0101
- BAR (Base Address Register): From these registers, the PCI device get to know how much I/O and memory space it has been allocated to operate.

2.1.1.3 Device Detection

PCI systems probes and detects devices and system resources (memory space, I/O space, etc..) attached to the system by inspecting the PCI configuration address space. The BIOS is stored with system specific hardware mechanism that accesses the PCI configuration address space. The address space contains PCI configuration headers of the various PCI devices attached to the system. Each PCI device is identified by a bus, device and function. Every PCI device allocates a data structure called configuration header in the address space. Each PCI device can have up to a maximum of 8 functions, so allows for multi-function devices. The PCI header has fields that classifies it as a bus or device. Only PCI configuration code reads and writes the PCI configuration addresses. The device drivers should only read PCI I/O and PCI memory addresses. The PCI configuration code attempts to examine all possible PCI configuration header for a given PCI bus and know which devices are attached to the bus. Once the PCI device is detected, the device specific I/O and memory space is allocated to the device by the configuration software upon reading the configuration registers BAR.

University of Cape Town

2.1.2 ACPI (Advanced Configuration and Power Interface)

2.1.2.1 Brief history

Untill the 90s, there was limited support for power management in personal computers. In 1991, Intel and Microsoft codeveloped APM (Advanced Power Management), a BIOS based system power manager. Even though it provided CPU and device power management there was lack of communication between OS and APM BIOS with the latter being required to maintain complex state machines. This lack of cooperation between system components and lack of participation by addon components in power management led to more technology industries to get involved to find a solution. It was ACPI (Advanced Configuration and Power Interface), an interface specification to hardware, software and firmware. ACPI is a collection of power management BIOS code that provides advanced power management and better communication between interfaces. ACPI addressed some of the issues of APM schemes by making BIOS a layer responsible to pass information about hardware control mechanisms to OS and allowing power management policy decisions to be implemented and controlled by OS and driver layer .

2.1.2.2 Overview

ACPI is an interface specification thats responsible for configuration and power management of devices and entire systems. It is a key component to OSPM (Operating System directed Power Management)[3] and UEFI (Universal Extensible Firmware)[4] systems.

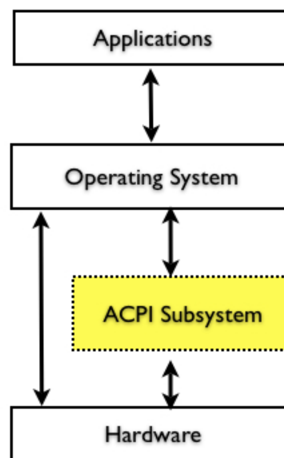


Figure 2.3: ACPI system architecture

Universi

The diagram above illustrates the various components of an ACPI compatible system. ACPI provides more control and flexibility to the Operating System. It enables the OS to manage power activity of devices and respond to events. It provides an abstract interface for configuration of an ACPI system. The OS can now interact with hardware through ACPI routines. The OS plays a pivotal role in characterising an ACPI system.

The ACPI subsystem consists of the following run-time level components:

- ACPI Tables - The core of ACPI subsystem. These tables provide information about the system hardware. Information needed for Plug and Play and Power Management is stored in these tables.

[3] A computer specification defined by the OS for device configuration and power management
[4] Another specification that defines a software interface between OS and platform firmware

- ACPI Registers - These are registers for events, controls, timer, processor control registers and general purpose events. The location of these registers can be obtained from ACPI tables. Most of them are fixed registers available to all systems and other registers can also be assigned by manufacturers.
- ACPI BIOS - System BIOS sets up a linked list of pointers containing addresses to information stored in the ACPI tables.

The system BIOS routines initialises the CPU, memory controller and enables memory and chipset. Following POST, ACPI tables are allocated in the system memory. The system tables are setup in memory as illustrated in the diagram below:

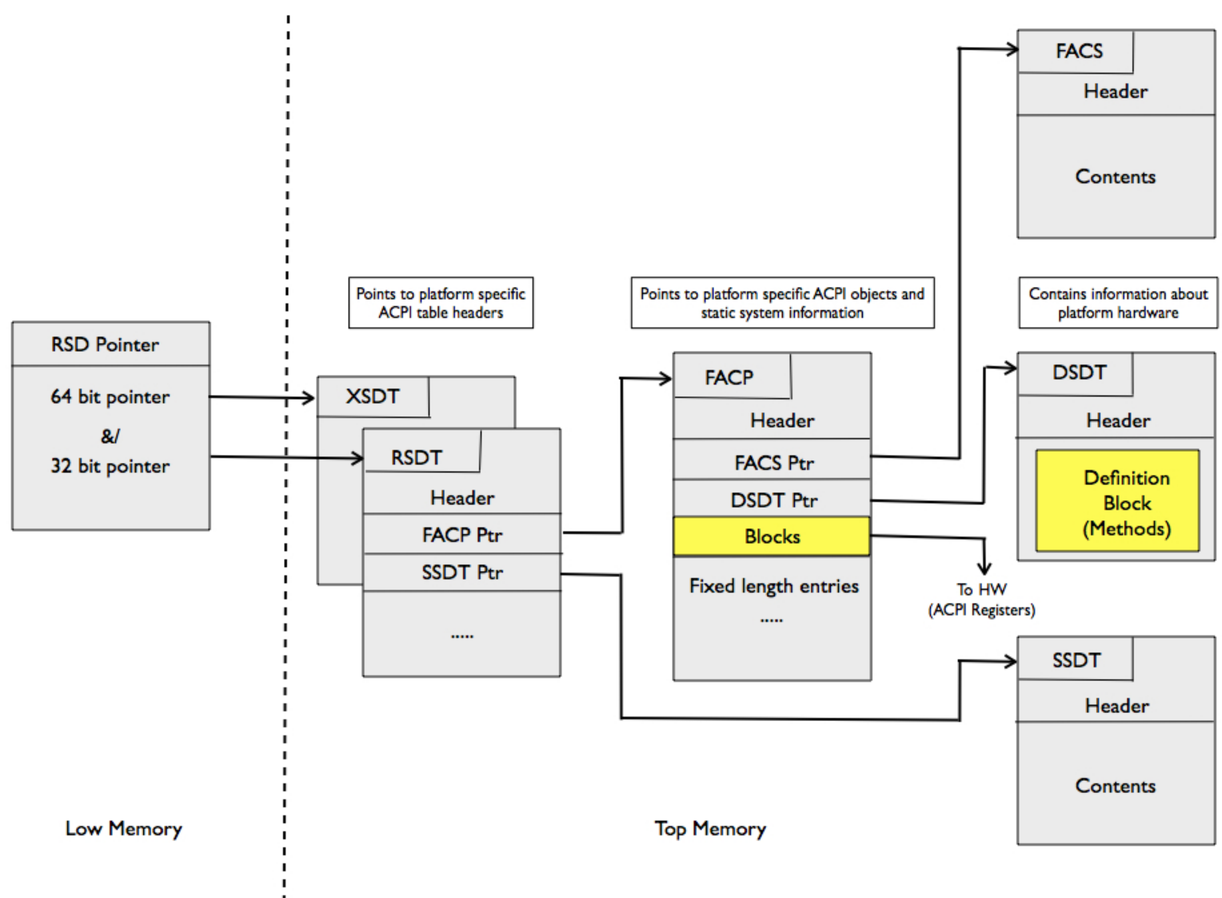


Figure 2.4: ACPI device detection

2.1.2.3 Device Detection

At boot time, ACPI BIOS reports to the OS through a set of tables namely ACPI tables. The starting point is obtained from the RSDP (Root System Description Pointer) structure located in the low memory of the systems memory address space. This structure points to

the RSDT (Root System Description Table) / XSDT (Extended System Description Table) depending on the size of pointer. The table of interest is the next inline which is FACT (Fixed ACPI Description Table) which contains pointers to blocks that hold information about ACPI Registers. It also points to a very important data structure namely DSDT (Differentiated System Description Table) which contains definition blocks, highlighted in yellow in Fig 2.4. Definition blocks holds the implementation details of the hardware platform in the form of data objects arranged in a hierarchical manner. The OS populates ACPI namespace[5] during system boot time with the information from DSDT. Definition blocks are encoded in AML (ACPI Machine Language). Firmware programmers write definition blocks in ASL (ACPI control method Source Language) and operating systems use an AML intepreter to translate it to OS meaningful format. The operating system loads these objects into kernel and uses them along with ACPI device drivers. Figure 2.4 was derived from the description of ACPI system description table architecture explanation in ACPI specification [19].

2.1.3 OF (Open Firmware)

2.1.3.1 Brief history

Nearing the 90s, there were many hardware platforms and boot ROM configurations that existed. Hence maintenance and support work needed for all the different platforms was also more. To prevent such a scenario, Sun introduced a novel approach called Open Boot. The aim was to have one boot ROM that runs independent of the underlying hardware. Later in 1991, Open Boot codenamed into Open Firmware with efforts joined by Apple and IBM. Open Firmware official standard, IEEE 1275 was published in 1994 and later got withdrawn in 1999 as an IEEE standard because of procedural reasons . The *OF* working group is still active and support for different CPU architectures are available.

University of Cape Town

2.1.3.2 Overview

OF is a hardware independent *boot firmware*[6] that loads the operating system. *OF* was developed using FORTH[7] programming language.

OF is a bootcode that is similar to the BIOS of an x86 PC. The main functions of *OF* [20] are as follows:

- Provide early initalisation code to configure the system

[5] A large data structure constructed from named data objects present in DSDT

[6] Boot firmware is ROM based software that acts a bridge between power-on of the system and loading of the operating system

[7] Forth is a stack based programming language that is commonly used in microprocessors.

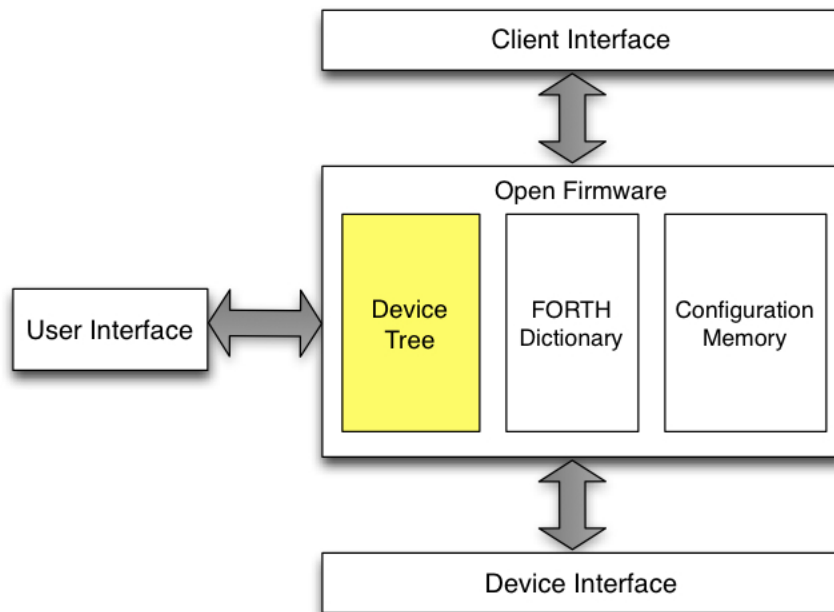


Figure 2.5: Open Firmware internal and external interfaces

- Determine the physical configuration of the system and builds a device tree structure
- Load the operating system and enumerates devices by looking through the device tree data structure.

OF has three external interfaces namely:

- User Interface - Outer interpreter that *OF* uses to provide a shell capability to the user. This is used for administration and debugging
- Device Interface - Inner interpreter that *OF* uses to provide plug and play capability by reading configuration information and processor independent limited features device driver residing in ROM.
- Client Interface - Well defined operating system interface that provides services for OS and loaders.

The three *OF* internal data structures are a device tree (hierarchical representation of devices in the form of nodes, properties and methods), FORTH dictionary (a lookup table which contains Forth words) and configuration memory (storage and maintenance of user choices, setting of environment variables).

OF provides a standard that is machine independent and instruction set independent. This provides the advantage of needing only a single boot driver that will work across a wide range of hardware, thus aiding in the autoconfiguration across different platforms. The underlying technology in *OF* is Forth microkernel which provides the features to develop

University of Cape Town

drivers and interfaces imparting plugin and descriptive capabilities. The boot drivers are written in *FCode*, compiled form of Forth program. *OF* has an interactive Forth language interpreter called *Tokenizer* which is responsible for compiling Forth programs into *FCode*, a byte based machine independent code. During the booting process, *OF* dynamically loads and executes the *FCode* drivers and methods. The boot-time driver for each device is provided in the form of *FCode* programs when executed initiates the build of a device tree data structure. As the probing progresses, the device tree structure grows in width and depth. The device tree gets populated with nodes that represents the device itself, properties and methods that describes the device in detail. The nodes with children are usually buses and without children are usually individual devices. Each node is distinguished by name and unit address. *OF* provides plug and play capabilities by providing boot drivers on demand from its integrated set of drivers or after executing the *FCode* programs on external device plugged in.

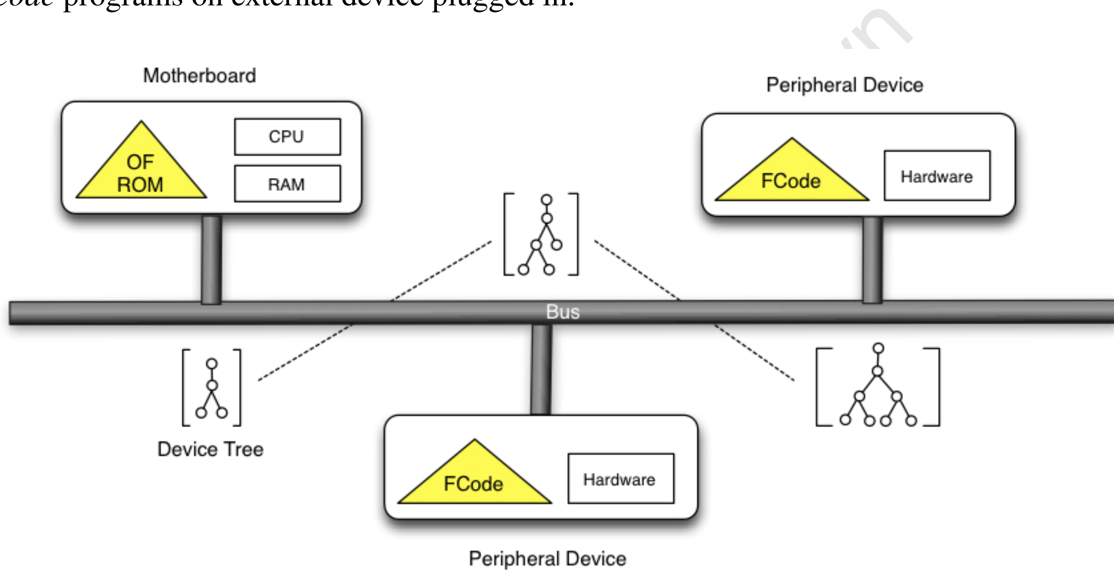


Figure 2.6: Open Firmware probe and detection

2.1.3.3 Device Detection

During boot time, *OF* starts the probing process of identifying the underlying hardware. The *OF* ROM contains *FCode* interpreter which aids in converting Forth programs to *FCode*. It reads and intepretes the ROM resident *FCode* and creates nodes for each device detected, fills properties and methods which gets populated in the form of a hierarchical data structure namely *device tree*. The *FCode* program creates device driver methods in RAM which are useful for initialising the corresponding device. When *OF* passes control to the OS, it uses the client interface to access OF services. The OS accesses the device tree constructed for auto-configuration and initialisation.

The hardware information gathered is of use to the hardware designer or software developer as bugs can be eliminated at boot level before it even gets passed to the OS using *OF*

user interface capabilities. U-boot and Linux does not really do Forth that much but this could be future work.

2.1.4 Summary

Some of the available conventional computer platform technologies were reviewed conceptually and a table 2.1 summing up / comparing the different technologies is drawn. The various parameters were selected keeping in mind to choose an appropriate technology or extend concepts from the technology for our design and implementation. The focus of each review was how device detection happens in different hardware technologies for computer platforms.

Most x86 based machines are based on PCI. As explained in section 2.1.1.3, PCI configures and makes device detection possible in a straightforward way. The kernel scans the PCI bus and determines what devices are attached along with their respective addresses. For this PCI relies on configuration headers to be made available for each device attached to the bus. The devices that are not enumerable (devices not attached to bus) are made available at known locations. This information is hard-coded for the kernel to look at. In short, PCI builds information about the system with data available from PCI configuration registers.

ACPI makes BIOS a responsible layer for passing underlying hardware information and coordinating with the ACPI tables that the OS builds on. It relies on AML methods to be written by developers that gets included as definition blocks. This makes it more sophisticated for device description. ACPI assembles system information with data from ACPI tables, mainly definition blocks in the DSDT table.

OF provides firmware support and makes device discovery simpler. It gathers information about the system by assembling the device tree. Some buses / interconnects that are not self-enumerable also get represented in the device tree format. The information isnt hardcoded, it is represented in this format and gets passed onto the kernel. The kernel uses the information supplemented for initialisation, device detection and binding appropriate driver to device.

University of Cape Town

Table 2.1: Comparison of various hardware device detection mechanisms

PARAMETER	PCI	ACPI	OF
Hardware Description	PCI Configuration register	DSDT	FDT
Compiler	OS specific device driver	ACPI Driver / AML Intepreter	DTC
OS kernel format	PCI Bus Driver	AML	DTB
Source Format	Register set defined by PCI specification	ASL	DTS
Hierarchichal Representation	PCI Configuration Register	ACPI Namespace	Device Tree
Platform Dependency	No	No	No
Device Detection	PCI Configuration Register	ACPI Tables	Device Tree
Complexity	Medium	High	Medium
Run Time Kernel Support	Yes	Yes	Yes
Standard	PCI	ACPI	IEEE 1275-1994
Supported	Yes	Yes	Yes
Year Introduced	1992	1996	1994

The acronyms DSDT, FDT, AML, ASL, DTB, DTS are expanded in nomenclature chapter
† – Standard : official name of the published standard that exist; Supported : The technology mentioned is actively supported or not
Complexity : understanding the technology and implementing it; Source Format : Representing the devices in a format used by the developer
Run Time Kernel Support : Dynamic (plug and play) device detection from the kernel

University of Cape Town

2.2 Device detection in FPGA-based software systems

2.2.1 Related Studies

There have been many advancements in FPGA-based software computing in the past decade and still continuing. Computationally intensive and demanding areas like speech recognition, bioinformatics, gene sequencing and radio astronomy have found this platform viable to launch its many applications.

The use of high speed computing fabrics like FPGAs in conjunction with microprocessors for solving computationally intensive algorithms and problems constitute the broad area of *"reconfigurable computing"*. The concept of reconfigurable existed in 1960's and gained importance with Gerald Estrins [11] paper proposing to build a general purpose, high performance computing device. Since 1990s, the area of reconfigurable computing became an active field of study. Some of the research projects like ReConfigMe, GATOS, FOCA [9, 8, 41, 25, 29] outlined in Fig 2.7 approaches the task of designing operating systems for FPGA-based reconfigurable computers. Most of them dedicate their efforts to solving the problem of dynamic FPGA resource allocation, partitioning and memory management of FPGA resources or virtualization between software and hardware tasks on FPGA-based systems.

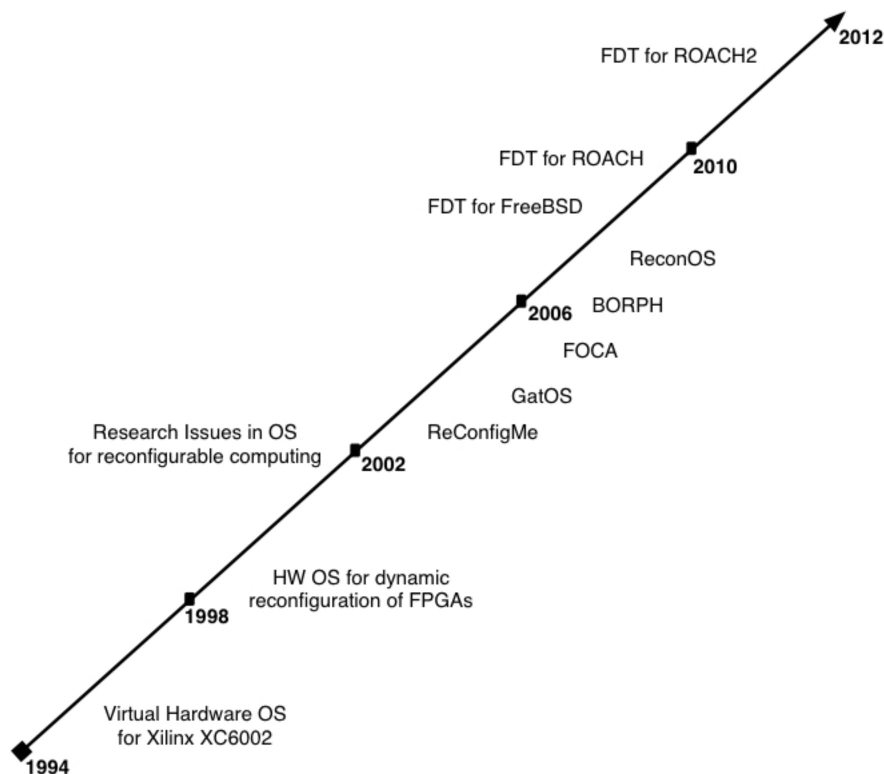


Figure 2.7: Recent operating systems for FPGAs [32]

In addition there are off-the-shelf operating systems such as Linux and VxWorks that run on commercial FPGA based reconfigurable computers use FPGAs as software accelerators. BORPH, introduced in 2007, brought a novel concept of running a *"hardware process"* on FPGA similar to a software process on a CPU and extended UNIX process semantics to hardware process also. Inorder to achieve this, BORPH extends a standard linux kernel to include support for FPGA resources.

Another mechanism that aided operating systems like Linux and FreeBSD for device detection was the use of *"device trees"* to describe underlying hardware and components. The concept of device trees is inherited from Open Firmware IEEE 1275 standard and is used to describe devices whether they are self-enumerable or not. This provided OS with the ability to use device trees for describing underlying hardware in a platform-independent manner and aided OS to extract information and use them during load / run-time.

2.2.2 Linux

2.2.2.1 Background

Linus Torvalds, a university student in Finland, wrote a terminal emulator for 386 processors in 1991. This work got extended into a monolithic[8] kernel with a range of features derived from the Minix[9] community of users. The initial capabilities of the kernel were limited as it was architecture dependent and had limited utilities in bash and gcc. Subsequent years saw the kernel merging with the GNU[10] project. Since that, GNU/Linux has emerged as an operating system that is non-commercial, platform-independent, non-proprietary and open source.

Table 2.2 below lists the Linux kernel release year, version released, a few important features related to the release referred to as milestones and SLOC[11] (Source Lines Of Code).

University of Cape Town

[8]Operating system architectures where the entire operating system works in kernel space
[9]Unix-like operating system founded by Andrew S Tanenbaum which uses a microkernel
[10]A project founded by Richard Stallman with aim to provide free and quality software
[11]SLOC refers to the number of lines in the kernel source code

Table 2.2: Short history of Linux

Year†	Version†	SLOC†	Milestone†
1991	0.01	10,239	Linus Torvalds contributes Linux kernel
1994	1.0.0	176,250	First production release
1994	1.2.0	310,950	Linux ported to non-intel processors
1999	2.2.0	1,800,847	Multiprocessor support, networking
2001	2.4.0	3,377,902	USB support, ISA plug and play
2003	2.6.0	5,929,913	Improved scheduler and VM subsystem
2011	3.0	14,619,185	Fast boot, storage improvements
2012	3.2	14,998,651	Ongoing improvements

† –
Year : year which the Linux kernel got released
Version : released kernel version
SLOC : Source Lines of kernel Code
Milestone : Important contributions for the kernel

Town

2.2.2.2 Overview

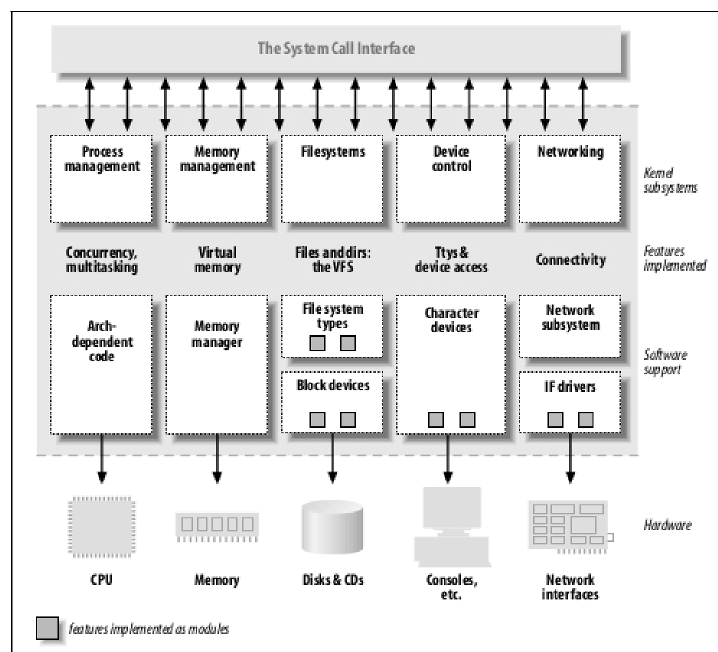


Figure 2.8: Linux kernel overview [23]

The Linux kernel is the core component of the operating system organised into subsystems and layers. The kernel manages several concurrent processes that performs different tasks.

As depicted in Figure 2.8, kernel task management are distinct namely:

- Process Management - Kernel manages the creation and destruction of processes and threads. The time-sharing and scheduling of CPU between processes is managed through efficient algorithms. The communication between the various processes through signals and pipes is also handled by the kernel.

- Memory Management - The kernel provides means and mechanisms to manage the memory available to processes. It is in charge of creating virtual address space when available memory starts to exhaust.

- Filesystems - Almost everything in Unix is treated as a file. The kernel is in charge of creating and presenting a structured filesystem to the user inorder to talk to the hardware underneath meaningfully.

- Device Control - Physical devices get managed and operated from the kernel through device drivers.

- Networking - The OS is in charge of collecting and distributing network packets across the system.

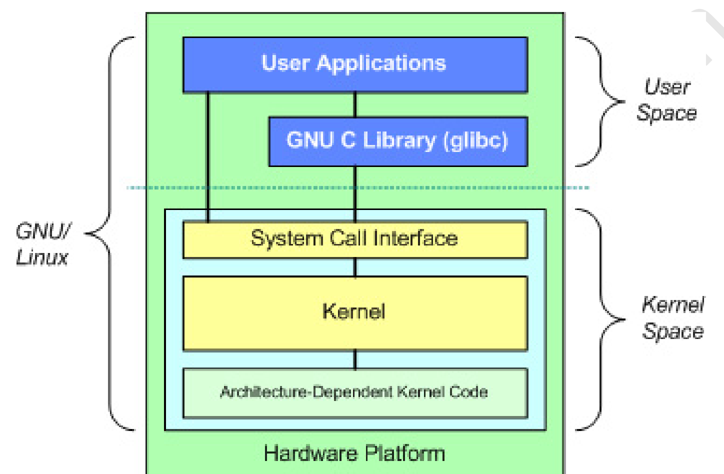


Figure 2.9: High level overview of GNU / Linux [27]

The architectural decomposition of a Linux kernel as illustrated in Fig 2.9 reveals a layered architecture. At the top there is the user level and bottom is kernel level. Each user instance has its own virtual address space and kernel has a single address space. All applications are executed in the userspace. It transitions between the two levels through a system call interface layer. The GNU C library provides support for the applications run on the userland. The core of the Linux is the kernel code which is architecture independent. This code is common to all the processors and architectures supported by Linux. The architecture-dependent code part is specific to a particular architecuture and processor. Linux kernel follows a monolithic operating system architecture where all the basic services are managed from the kernel mode.

2.2.2.3 Features

- Portability - Linux can be ported on a number of different architectures and processors. The architecture independent code makes it possible for the ease of portability.

- Efficiency - Linux dominance in the IT market has been mainly due to its performance and efficiency, thanks to the cleary defined and documented architecture and free access to source code which aids developers from around the world to tweak and consistently improve performance and efficiency.
- Dynamically loadable modules - Linux is a modular operating system. The components can be added and removed from the kernel during run-time. If a particular device requires a module, it can be loaded while the kernel is running. This feature is of immense use during early stages of device driver development.
- Growing database of device drivers - The device drivers gets added to the Linux database with minimal bugs due to the number of people involved. The device drivers pertaining to common devices and specific fields are increasingly growing with Linux being a popular choice of OS among the developers community.

2.2.2.4 Device Detection in Linux

Linux has a unified device driver model which breaks the system into buses, devices and classes. Device driver abstracts the hardware from the user by handling all the low level communication details. It acts an intermediate layer between the user application and the device, thereby restricting access to the user programs to readily access the important kernel data structures and hardware itself. This adds a layer of protection and isolates the user from damaging the system. The device is matched with the appropriate device by the kernel and bound together.

University of Cape Town

2.2.3 BORPH

2.2.3.1 Background

BORPH (Berkeley Operating System for Reprogrammable Hardware) is an operating system framework developed by Hayden So for FPGA based reconfigurable computers. The operating system was developed as part of his PhD thesis in 2007 at University of California, Berkeley. Several papers have been published on BORPH from improving usability [17], to file system access [37] and providing runtime filesystem support for FPGAs [38, 36].

2.2.3.2 Overview

BORPH is an operating system that extends a standard linux kernel[12] to include support for FPGA resources in a RC. It treats FPGA resources as computational resources, the

[12]Core component of operating system that acts as a bridge between hardware and applications

same way conventional OS treats the processor. This means that just like a software application running on the processor is managed by the kernel, the same applies to a hardware design running on a FPGA. Gateware images programmed onto the FPGA are registered as a hardware process[13] which interacts with the kernel to appear like any software process. A message passing system is implemented just like the system call interface for a hardware process to communicate when it crosses the user-kernel boundary space.

The BORPH process of running a design on FPGA is outlined as follows:

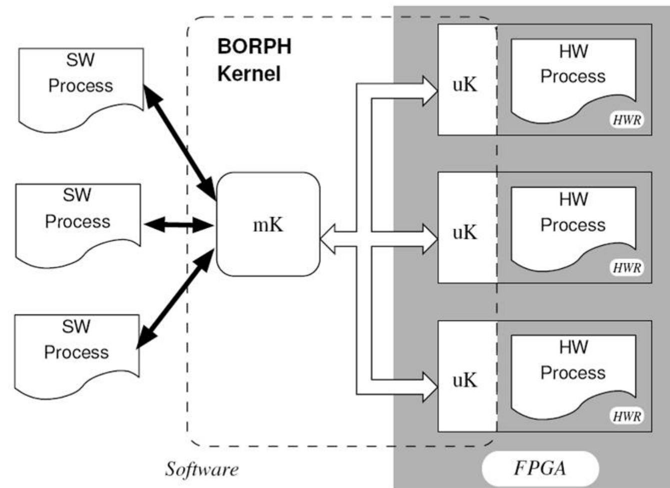


Figure 2.10: High level block diagram of BORPH [16]

As illustrated in Fig 2.10, BORPH kernel can be decomposed into two logical components, the mK(Main kernel) and uK(Microkernel). The mK is the main controlling kernel. It is a slightly modified version of Linux running on a powerpc. The specific diagram referenced has a modified version of Linux 2.4.30 kernel running on PowerPC 405 core on a BEE2 board. A standard Debian powerpc root file system provides the familiar Linux applications to the user. The uK is in charge of managing the reconfigurable hardware regions (HWR) on the FPGA and all low level management of hardware process. BORPH has been ported to a number of hardware devices since BEE2 like ROACH, NetFPGA[14] etc. Device specific code is selected by kernel config option and compiled into the kernel thereby eliminating the need for device drivers. Each time a new device needs to be added, only the device specific code needs to be implemented thereby making portability easy in BORPH.

[13]A hardware process is term coined for a running design on FPGA.

[14]NetFPGA is a FPGA-based open source project that enables rapid prototyping of networking devices

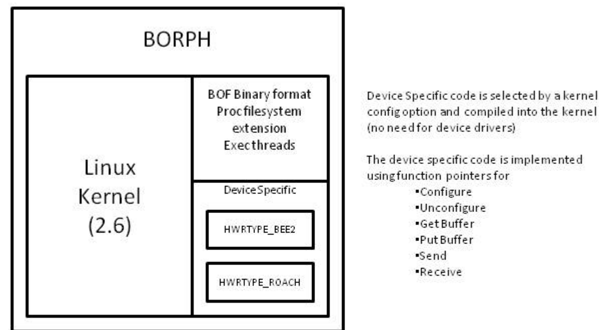


Figure 2.11: BORPH [5]

2.2.3.3 Features

- High level design flow

  Hardware designs are compiled using a combination of high level tools namely Simulink, Xilinx System Generator, libraries and wrappers. The wrapper is extended to include support to generate BORPH Object File (BOF)[15] instead of the default .bit file as final output.

- Ease of use

  The primary goal of BORPH is to increase the usability of reconfigurable computer by extending operating system support to run hardware designs. The users are allowed to run their hardware designs on the FPGA just like a software executable. The BOF files can be executed by users during BORPH run-time.

- UNIX semantics

  The UNIX semantics are applied to the hardware process thereby able to use UNIX file stream and pipe concepts. These are particularly useful for DSP applications.

- Access to reconfigurable resources

  BORPH exports hardware-mapped registers in FPGA to userspace as files. The /proc directory of Linux gets extended to include information about hardware processes.

[15] BOF file encapsultes FPGA configuration information and high level information about the design running on FPGA like locations and names of user defined registers, memory blocks, FIFOs

University of Cape Town

2.2.3.4 Device Detection in BORPH

The gateware programmed on FPGA has configuration information about the design as well as information about register names and locations, memory blocks, FIFOs etc. These hardware mapped software registers get exported to userspace as entries in /proc directory. BORPH provides an IOREG interface underneath /proc directory in the kernel where all the hardware process specific information in the form of software registers is stored. The user reads and writes to these registers to run the design on the FPGA.

2.2.4 Summary

We reviewed a few of the commonly used FPGA software systems. The characteristics are chosen keeping in mind the requirements for an automated gateware detection mechanism to be developed. For a larger project like MeerKAT, where high speed data is handled, the capabilities of an OS to perform is very important. CASPER had early adopted BORPH on its reconfigurable hardware platforms BEE2, ROACH. Even though the ease of usability of OS increased, it was noticed that the modified Linux kernel, BORPH was utilising lots of system calls[16] for performing a read / write operation. This is not considered optimal.

A solution for enhancing performance would be conforming to the Linux device driver framework where the complex abstraction is moved away from the kernel to the userspace. A memory mapped device driver for FPGA access, mentioned in Section 3.5.5 was developed as a result.

An experiment was conducted to measure performance between BORPH and Linux with conventional device driver for FPGA access. One of the gateware designs developed for MeerKAT was run on FPGA. A simple test of writing, reading and verifying the result of scratchpad register[17] was run. The results of this experiment was presented [33, 35] at CASPER workshops[18]. The experiment was further extended to include more cases as listed below.

The six cases used in the experiment as plotted in Figure 2.12 are:

1) An application accessing scratchpad register half a million times using a memory mapped device driver in Linux kernel

2) An application accessing scratchpad register half a million times using a memory mapped device driver along with msync method in Linux kernel

[16]system calls are function invocations made from user space applications inorder to request some service from the operating system

[17]one of the registers in the FPGA design that can be read and written

[18]Annual workshop on radio astronomy centered digital signal processing hosted by the CASPER collaborators

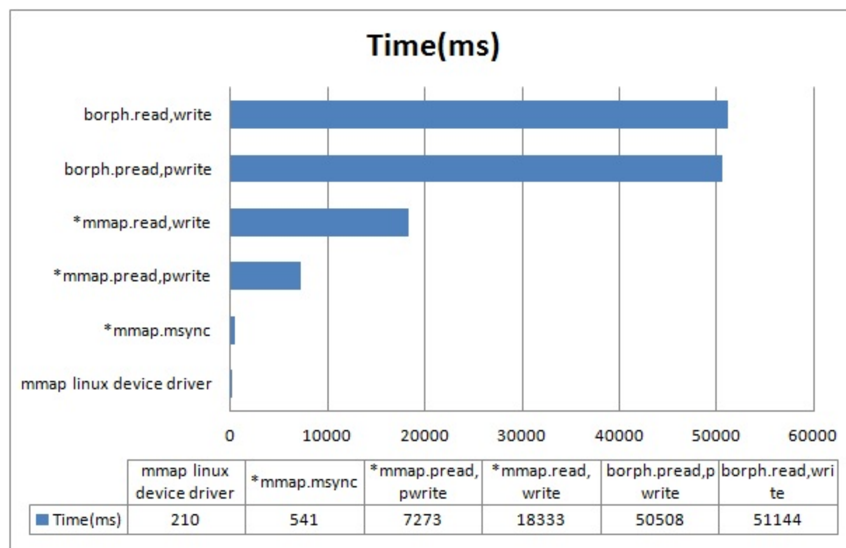
University of Cape Town

3) An application accessing scratchpad register half a million times using pread and pwrite with a memory mapped device driver in Linux kernel

4) An application accessing scratchpad register half a million times using seek, read and write with a memory mapped device driver in Linux kernel

5) An application accessing scratchpad register half a million times using pread and pwrite in a BORPH kernel

6) An application accessing scratchpad register half a million times using seek, read and write in a BORPH kernel



* mmap in the figure above refers to the memory mapped linux device driver

Figure 2.12: Results of test conducted between BORPH and Linux device driver with memory mapped support

The results plotted as a bar graph in Figure 2.12 shows that an application running in Linux using memory mapped device driver for FPGA access has a performance improvement by a factor of 100 over an application accessing FPGA in a BORPH supported kernel. The improvement in performance between read / write and pread / pwrite can be attributed directly to the reduction in system calls. More details about the working of memory mapped driver can be found in Section 3.5.5.

After conducting the test above and drawing from the experience of using systems enabled with BORPH and Linux with device driver support, Table 2.3 was tabulated. The table can be used as a reference for users to choose between BORPH and Linux with conventional device driver support for FPGA based systems. Device detection happens in BORPH through IOREG interface where all the device registers are exported while the linux kernel uses the device driver model where a driver is used to match the device and bind them. FPGA designs that make frequent access to FPGA memory resources will find the device driver model with Linux support useful while non-performance critical designs can find BORPH useful with its intuitive user interface.

Table 2.3: Reconfigurable software comparison

Characteristic	***Linux with conventional device driver***	***BORPH***
Range of compatible hardware	Very wide	Moderate
Performance†	Very high	Moderate
Ease of use†	Moderate	High
Developer Complexity	Moderate	Moderate
Developer Support	Minimal	Moderate
Source Code	Available	Available
Device Driver	Needed	Not needed
Supported Architectures	alpha, arm, blackfin, x86 microblaze, powerpc, mips	powerpc, x86, arm

† – Performance : Measured in terms of cost of accessing FPGA registers
– Ease of use : Accessibility of FPGA registers

2.3 Chapter Summary

University of Cape Town

This chapter discusses the various device detection mechanisms available for discovering devices from a technology and operating system point of view. We explored mechanisms that would aid in automating device detection with software support available.

In Section 2.1, we reviewed on some of the available hardware technologies in place namely PCI, ACPI and OF for conventional computer platforms. We draw comparisons between the different hardware device detection mechanisms in Table 2.1. From the hardware detection mechanisms summary in section 2.1.4, we see that *OF* provides a mechanism for describing a device in a format that is easy to understand and record namely the *device trees*. Once the device is described, we want a mechanism to operate the device.

Hence we explored in Section 2.2, some of the software detection mechanisms for FPGA, namely BORPH and Linux, commonly used in the radio astronomy field. We prefer to use Linux because we want to bind the device described in device tree to a device driver. The advantage of using device drivers, existing or custom-written for operating a device are many. It reduces the effort spend on writing software for a particular device. One can benefit from the growing database of device drivers that are supported. The performance benefits of a Linux system conforming to the unified device driver framework can be tapped into.

Chapter 3

Design

Chapter 3 introduces various components and their design or extension that led to the development of a mechanism for automated gateware detection. The implementation section, Chapter 4 will elaborate on the components that were assembled or modified to develop an automated gateware detection system. During the span of this project, two generations of digital signal processing boards were released, namely ROACH and ROACH2[1]. At the outset of this project, ROACH2 [10] development work is still in progress. This provided the opportunity of testing the design and implementation on two hardware generation DSP boards used for the KAT project.

This chapter begins by elaborating the design choices made after reviewing the user specifications [28] in detail. The hardware, software and toolflow constraints imposed on the project are determined. The design methodology steps are discussed which helped in extracting the finer objectives from user requirements. A system architecture diagram is drawn to visualise the concept of automated gateware detection using *OF* (Open Firmware). The system design presents each stage of *OF* extended design and its operations. Each operation presented in Figure 3.3 is addressed individually and discussed in detail in Section 3.5.

The design goal of *OF* enabled operating system is to extend the *OF* infrastructure available for probing hardware devices to include suitably designed gateware images programmed on FPGA based RCs (Reconfigurable Computers). In particular, an *OF* driven approach meets this goal by extending device tree support to FPGA designs which is presented in Section 3.5.4.

University of Cape Town

[1]ROACH2 is a high performance reconfigurable hardware board and is the successor of ROACH board. It was developed by the MeerKAT project as part of the CASPER collaboration. https://casper.berkeley.edu/wiki/ROACH2

3.1 Design Constraints

The MeerKAT DBE[2] (Digital Back End) team is one of the active contributors in the CASPER community. The DBE approach is mainly CASPER centric which consists of open source hardware, software, toolflow and libraries. The design of extending the *OF* infrastructure to probe hardware devices utilises meerKAT DBE hardware, gateware and software infrastructure. Hence the hardware platform chosen is the widely used ROACH board which has a Virtex 5 FPGA and an AMCC PowerPC and next generation ROACH2 boards which has a Virtex 6 FPGA and the same PowerPC.

Gateware or FPGA designs are generated with MSSGE toolflow as illustrated in Figure 3.11. U-boot, the bootloader, used for bringing the ROACH board up initially and for later booting the kernel has device tree and *OF* support which eliminates the need for searching other bootloaders for development. The software development work involved with U-Boot and Linux kernel was in C programming language. These constraints had to be taken into consideration and the design goal had to be aligned with these constraints in order to meet the objectives of the project in section 1.4.

3.2 Design Choices

The MeerKAT project utilises ROACH boards for its computationally intensive digital signal processing operations. The collaborative effort between MeerKAT and CASPER group intends to streamline and reduce the current radio astronomy instrumentation design flow through an open source approach. The ROACH boards being deployed for MeerKAT are configured with gateware designs generated from the CASPER toolflow, MSSGE[3] (Matlab / Simulink / System Generator / EDK).

Currently BORPH (Berkeley Operating System for ReProgrammable Hardware) is used as an operating system extension for the ROACH platform. BORPH is an operating system designed for reconfigurable computers and more background can be read in section 2.2.3. It extends a standard Linux kernel to include support for FPGAs in reconfigurable computers by adding a hardware abstraction layer. Any gateware image programmed is registered as a hardware process which interacts with the kernel just like any other software process. BORPH can be ported to support different hardware platforms some of which are BEE2 (Berkeley Emulation Engine Version 2), ROACH and ROACH2 and different software architectures like arm[4], ppc[5] and powerpc.

[2] Subsystem responsible for signal processing

[3] Toolflow developed at BWRC(Berkeley Wirless Research Center) which stitches together several design and implementation environments for generating gateware designs

[4] 32-bit RISC(Reduced Instruction Set Computing) architecture developed by ARM

[5] PowerPC and ppc(no longer supported) are 64 bit RISC architectures,backward compatible to 32-bit RISC architecture also, developed by IBM

University of Cape Town

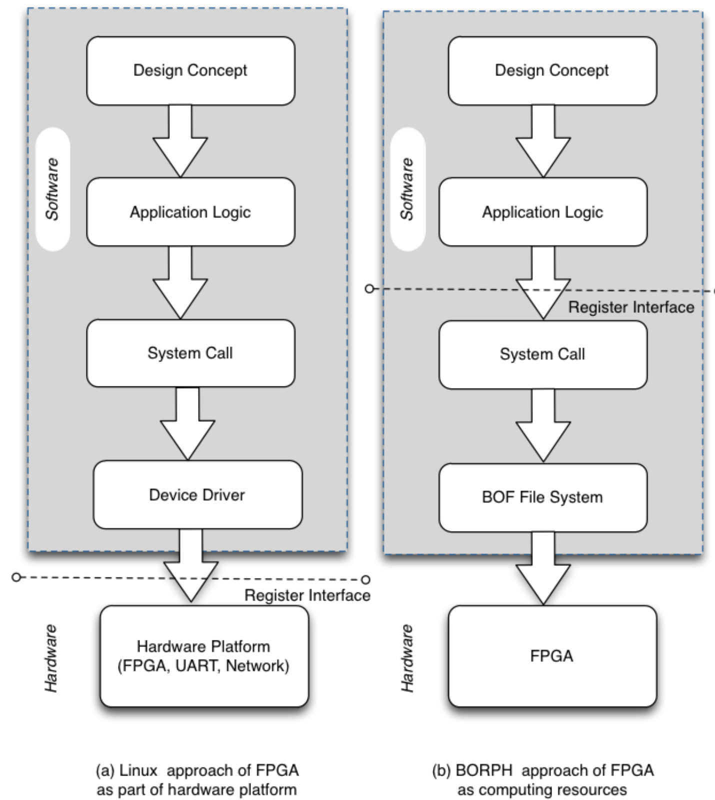


Figure 3.1: Linux and BORPH operating system flow diagram

The design flow diagram, Figure 3.1 illustrates the difference in approaches between traditional operating systems like Linux and extended operating systems like BORPH. The implementation of BORPH does not focus on providing device drivers for the gateware programmed, instead it exposes the hardware-mapped registers to user-space. It is left to the user to either work on the registers directly or write logic that interacts with the registers exported to user space. While the existing BORPH approach provides a neat usable interface, it requires system calls for each read / write operation.

Some radio astronomical designs make frequent access to FPGA memory resources implying lots of register reads and write operations. This can compromise the performance of a DSP design that involves performance critical I/O between PowerPC and FPGA. So this trade off between performance and usability was taken into consideration for the choice of an alternative software approach that supports gateware detection and loads device driver to operate on it, thereby presenting a higher level interface by conforming to a conventional device driver model approach.

The alternate approach to software-FPGA interaction which supports device drivers for automated gateware discovery is by utilising the pluggable mechanism by which peripherals gets identified in the operating system. In a typical hardware plug-in scenario, the corresponding device driver gets loaded by the host operating system to operate the peripheral attached. The observation that gateware implementations on FPGA can be treated as attachable / detachable peripherals suggests that the same scheme of loading device

University of Cape

drivers can be extended for detecting gateware images programmed on FPGAs.

This leads us to adoption of *OF* technology for the design of automated gateware detection mechanism. Physical devices are made available to *OF* through a form of device tree[6]. U-Boot, the bootloader for the powerpc architecture in ROACH has built-in *OF* and device tree support. FPGA gets treated as a physical peripheral in this design and makes itself available to the bootloader and then to the *OF* enabled operating system thereby increasing the chances of the right software being loaded for the corresponding gateware image.

The user requirements listed in Section 1.4 of Chapter 1 provide a starting point towards initiating the project but doesnt not identify specific project requirements. One of the design methodology steps as mentioned in Figure 3.2 is to conduct requirements review and list finer objectives. The following are the finer objectives of the project:

- Investigate *OF* infrastructure with a view to extending it to include suitably designed gateware images and provide an architectural design.
- Describe gateware in device tree and verify detection and loading of appropriate device driver by the OS (Operating System).
- Understand *OF* platform bus initialisation and probing capabilities to ensure correct loading of modified device drivers.
- Generate gateware designs that can be applied to radio astronomical instruments.
- Write device drivers to operate new instruments described and detected from the device tree.
- Implement the software and utilities required to demonstrate the concept.

University of Cape Town

[6] A representation that describes the systems hardware devices and their interconnections

3.3 Design Methodology

The following figure illustrates the steps taken to design an automated gateware detection system using *OF*. Its a systematic approach that was undertaken starting from project outline and literature review, to integrated system testing and conclusions. The first step in the system design process was to establish the user requirements for the project. The process involved correspondences and meetings with engineers at KAT. From these discussions, the initial user requirements as listed in Section 1.4 were developed. The second step of the design methodology was to conduct relevant background research and literature review on the topics of device detection in hardware and FPGA based reconfigurable systems. The literature review conducted in Chapter 2 aided in developing a solid understanding of the various device detection methods for FPGA based systems.

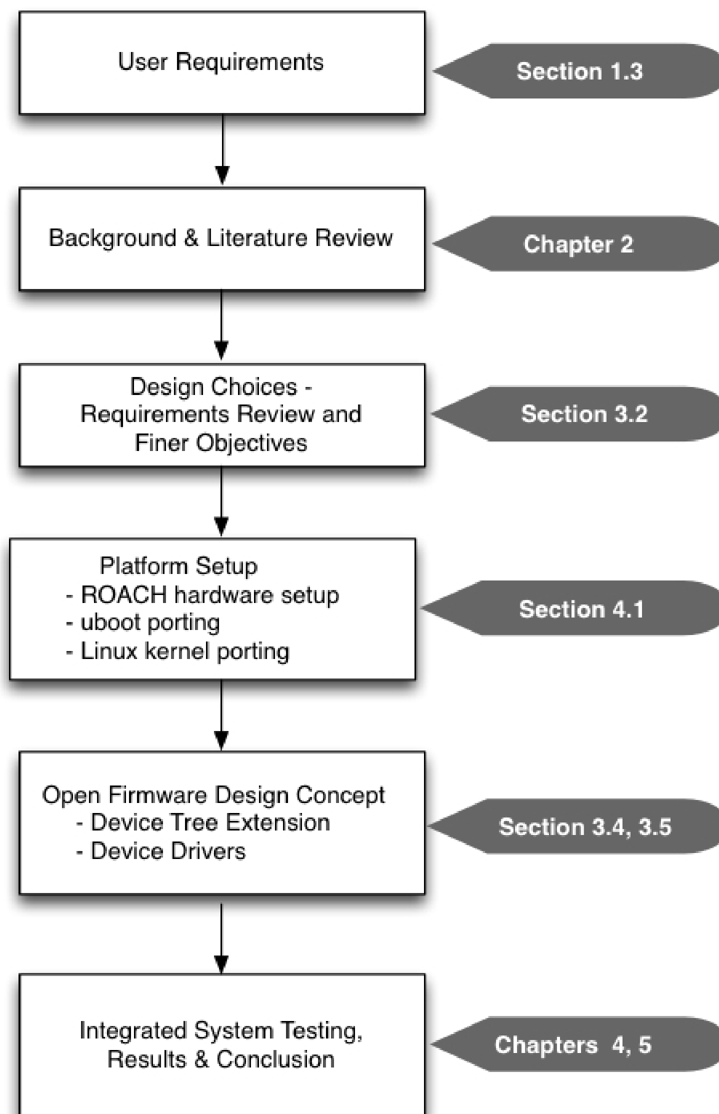


Figure 3.2: Design steps to reach the final system

The third step, requirements review process in Section 3.2 helped in defining the finer objectives or final goals of the project. The finer objectives were derived after obtaining a better understanding of the system and choosing *OF* as the technology to automate gateware discovery. As part of the fourth design methodology step, the required platform setup for applying the research was setup with efforts spent in porting the bootloader and kernel to the ROACH platform. The *OF* technology was applied to the platform in the fifth step in Section 3.5.4, thereby allowing us to utilise the features provided by *OF*, notably the device tree extension and device detection. In the final step, the final system was integrated by extending support in *OF* for including suitably designed gateware images for radio astronomy. The test and results conducted for developing an automated gateware detection mechanism using *OF* are elaborated in detail in Chapter 4. Conclusions and recommendations for future work were derived based on the outcomes and results of the project. As highlighted in grey extension boxes in Figure 3.2, Chapter 4 and 5 lists the final step undertaken in the project methodology.

3.4 System Architecture

The overall architecture of the Automated Gateware Discovery System is shown in Figure 3.3. *OF* technology provides the feature of device trees which we apply and extend to represent our devices. The PowerPC microprocessor configures the FPGA on the ROACH with user gateware designs and provides it with a functionality that can be described in a textual format called DTS (Device Tree Source) files. U-Boot, the bootloader for the PowerPC system probes the supplemented file and augments it to the list of physical devices attached to the PowerPC which gets represented as a device tree. This augmentation of functionality programmed onto the FPGA extends the database of devices available to the bootloader to operate and pass onto the kernel thereby providing run-time configuration support.

When the Linux kernel gets loaded, it is provided with a Flattened Device Tree (FDT), which contains information on the hardware devices that it needs to operate. The actual probing and detection of devices within the *OF* framework starts and the corresponding device drivers are fetched to operate on the gateware designs implemented on the FPGA. The user application can communicate with the designs now treated as another physical device through device nodes. The individual blocks that constitute the system architecture are explained in Section 3.5.

University of Cape Town

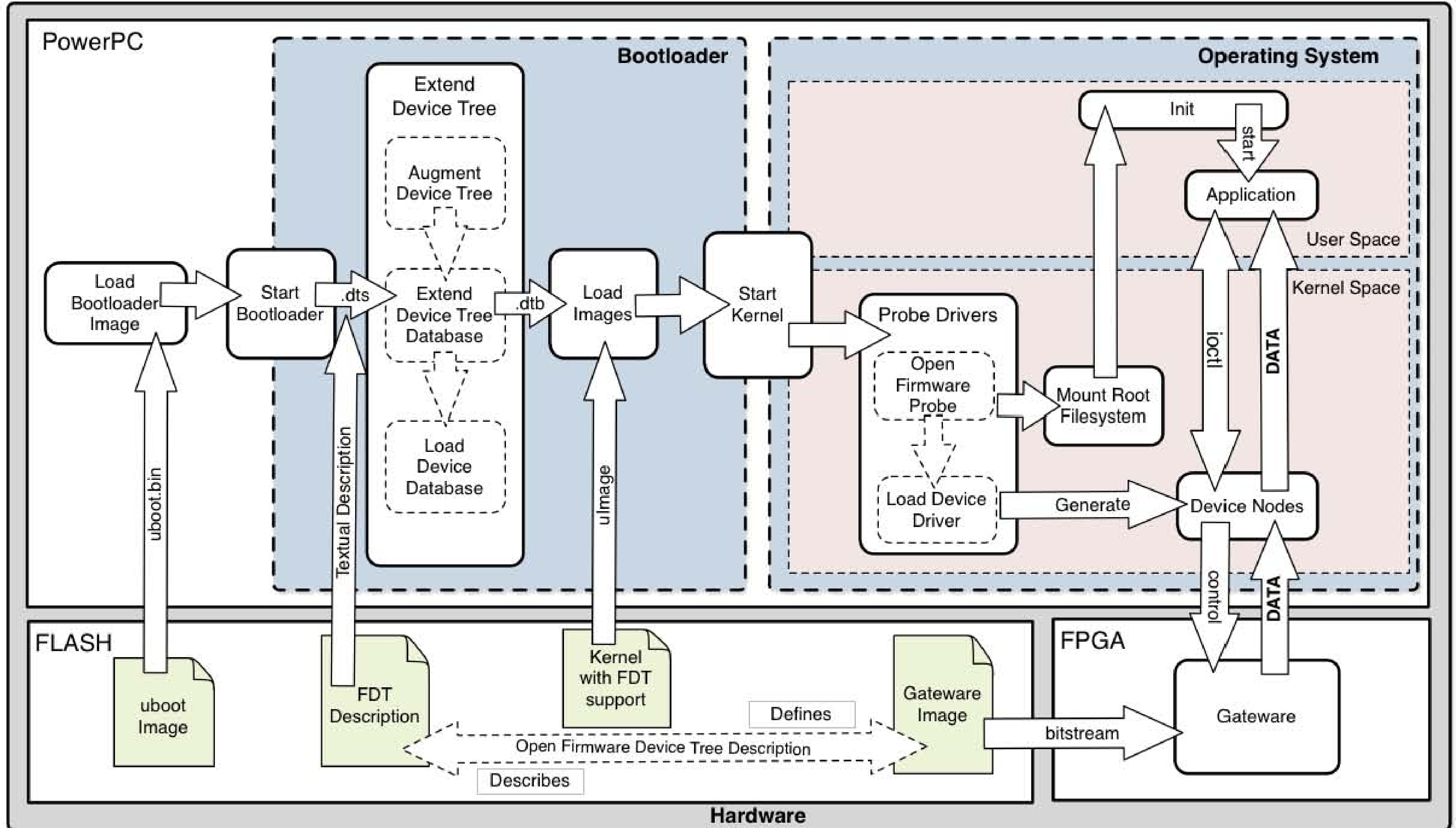


Figure 3.3: Automated gateware discovery system architecture

3.5 System Design

3.5.1 System Overview

At a high level, the system being designed consists of the following hardware devices, a ROACH board with ADC attached to it and a control computer. The connection between various components of the system are discussed below.

3.5.1.1 System Connection Diagram

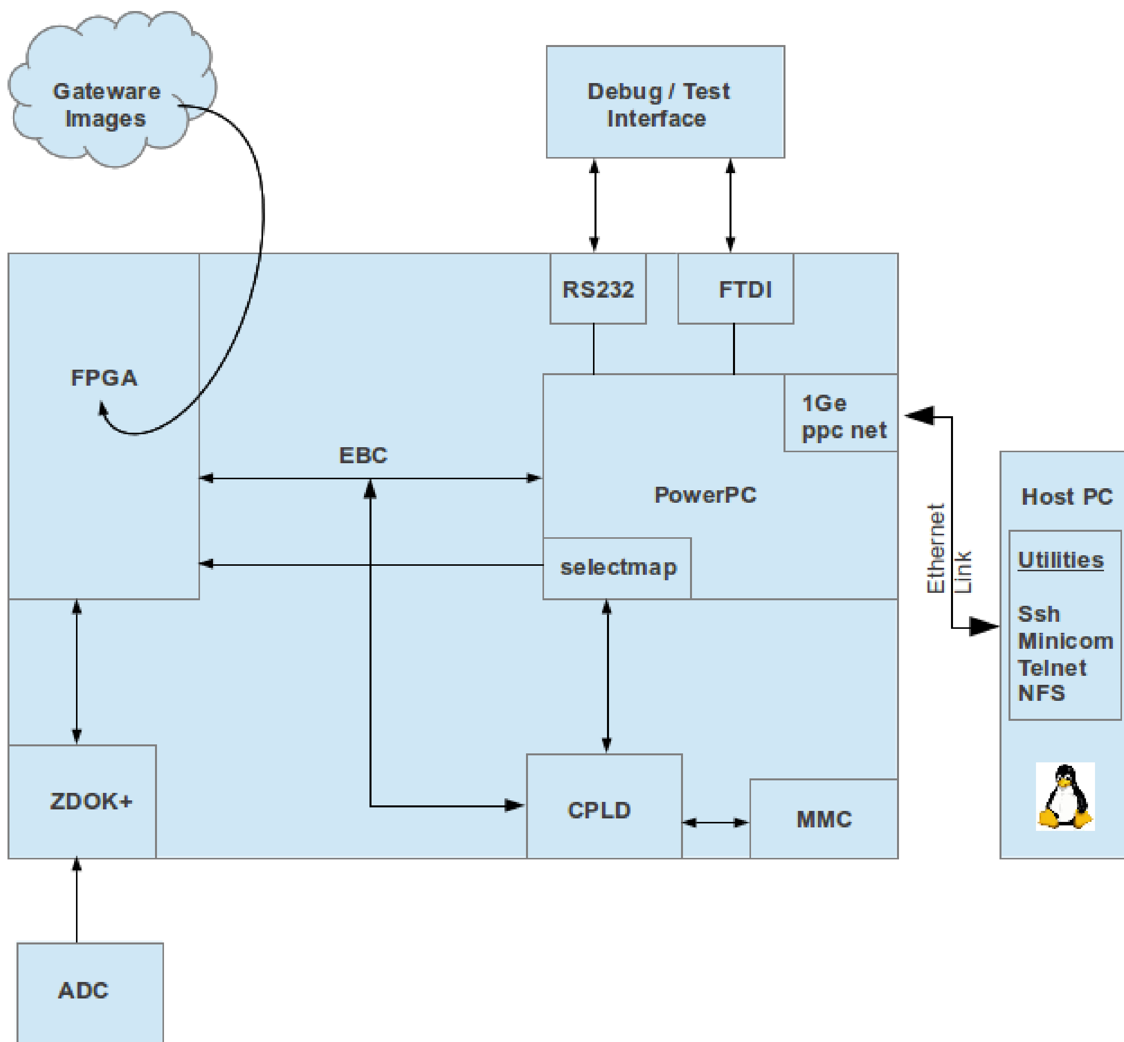


Figure 3.4: System connectivity diagram

Figure 3.4 shows the various connection interfaces within the ROACH boards and how they communicate each other. The PowerPC processor and Virtex series FPGAs are the major components of the board that is used for high speed signal processing and control. The PowerPC is in charge of loading the bootloader and ultimately the operating system. The PowerPC can be communicated from the control or host PC through serial

data connection, telnet connection or NFS (Network File System). The communication link between the PowerPC and the FPGA is through the EBC.

3.5.1.2 AMCC PowerPC Architecture

The processor on ROACH boards is the AMCC Power Architecture 440EPx Embedded Processor. The 440EPx is a high performance, low-power SOC (system on a chip) using IBM CoreConnect bus architecture utilised in many high end embedded applications. The following are some of the main features[7] of 440EPx which are of interest to this design:

- High performance 32-bit RISC processor
- EPB (External peripheral bus), 32-bit data bus
- Dual 10/100Mbps Ethernet ports
  - Useful for debugging, controlling and communicating with multiple ROACH boards over the network.
- On-chip SRAM
- DDR (Double Data Rate) SDRAM (Synchronous DRAM) controller
  - Memory controller for bringing the board up and running
- Four user-configurable serial ports
  - serial UARTs are useful for debugging the programs running on powerpc. Serial interactive programs like minicom and hyperterminal use serial UARTs for communication between host and PowerPC.
- Programmable Interrupt Controllers
- GPIO (General Purpose Input Output) interface
  - GPIO interface is used by select map logic and some of the input device drivers.
- Bootloader NOR flash on EPB
  - Flash is useful for storing bootloader, kernel, environment variables and root file system.

[7]There are several other important features, but a listing of features that are of interest to this design is only mentioned

University of Cape Town

- External MMC (MultiMediaCard) flash
  - More capacity for storing user preferred root file system.
- JTAG interface for debugging
  - Basic debugging interactive software tool

Figure 3.5: PPC440EPx functional block diagram [4]

The ROACH boards are architected around this 32-bit RISC processor. Figure 3.5 is a functional block diagram of AMCC PPC440EPx processor. The high end digital signal processing board gets its full functionality by integrating this microprocessor coupled with the latest Xilinx Virtex FPGA, memory chips and physical I/O hardware with minimal interface circuitry.

3.5.2 FPGA Programming Design

The Virtex series FPGAs on ROACHs are configured by loading bitstream into FPGAs internal memory. On ROACH boards, the FPGAs are configured by an external source, the PowerPC microprocessor. FPGAs can be reprogrammed / reconfigured an unlimited number of times. A bitstream is loaded through special configuration pins that provides an interface for different slave modes[8] shown in Table 3.1.

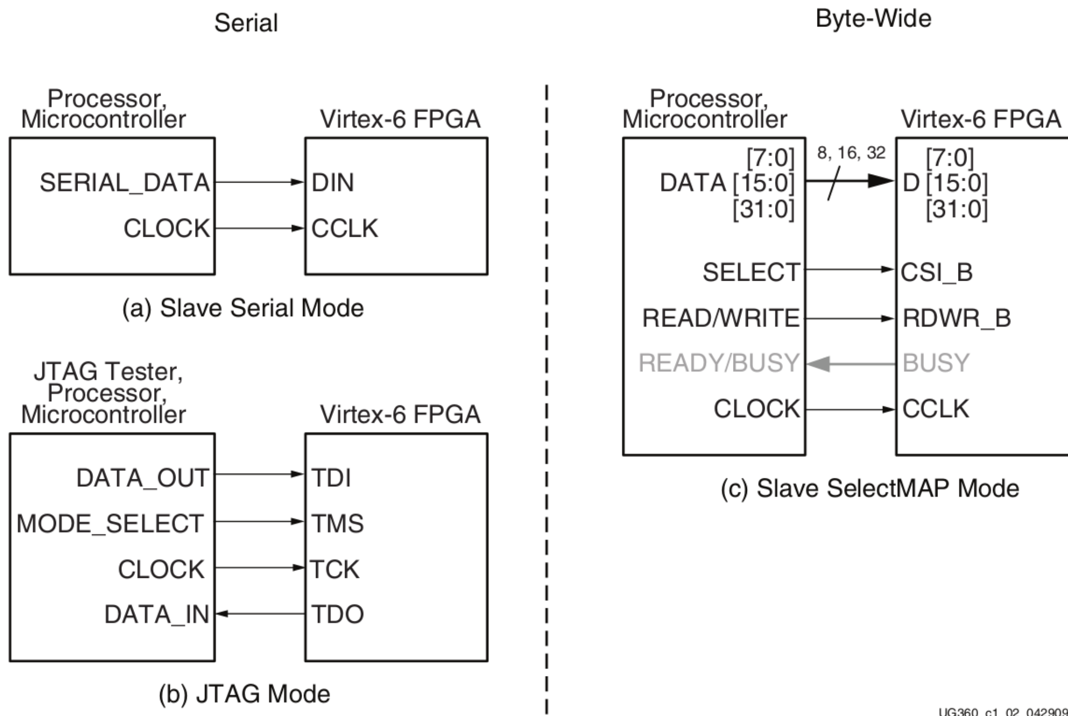


Figure 3.6: Slave configuration modes [44]

Although JTAG configuration mode is a simple serial configuration mode, the SelectMAP configuration mode is used because a) it provides a byte-wide peripheral interface for configuration b) avoids an externally plugged JTAG cable for configuration.

Table 3.1: Supported slave modes in ROACH

Configuration Mode	**Bus Width**
JTAG	1
Slave SelectMAP	8, 16, 32

[8]Externally controlled loading FPGA configuration modes

Table 3.2: SelectMAP signal functional description

Signal Name	Functional Description
cclk	A clock used to load the bitstream data into the fpga
prog_n	A signal which you set to zero to begin the configuration process
init_n	A signal asserted when the fpga is ready to accept configuration data
rdwr_n	A signal used to indicate reading/writing data into the fpga
data[31:0]	The configuration data that you clock into the fpga
cs_n	The data is only accepted into the fpga when this signal is low

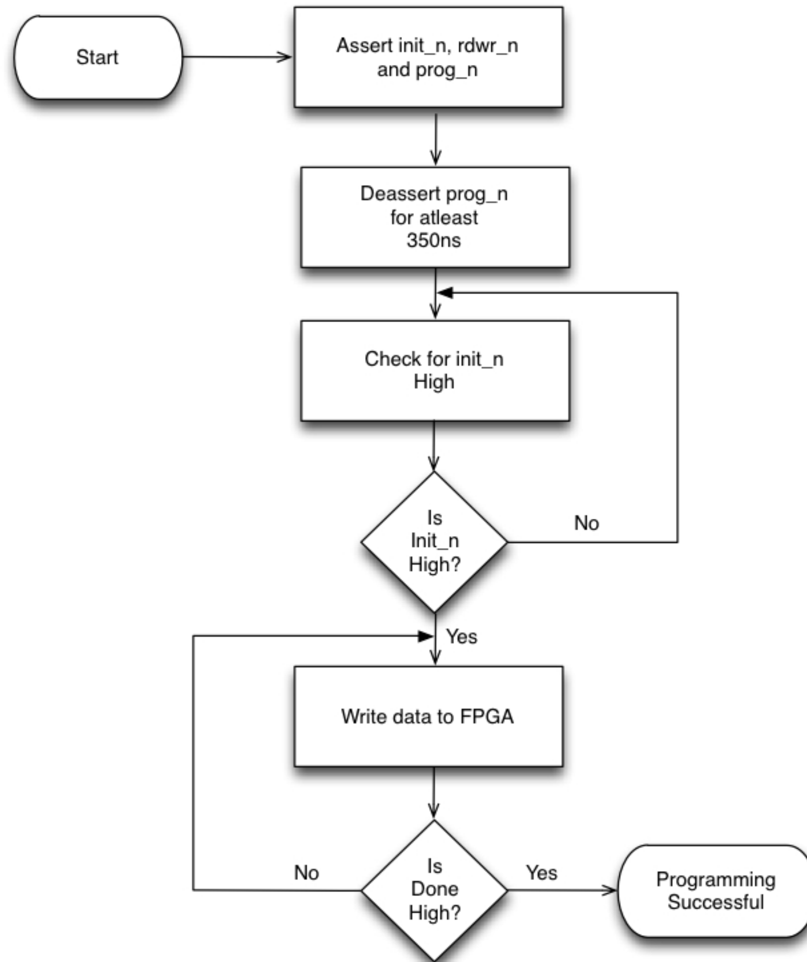


Figure 3.7: SelectMAP Configuration Flow Chart

There are several signals that constitute the SelectMAP interface which is listed in Table 3.2. On ROACH-1, all the signals excluding the data are driven from the CPLD. Inorder to set them we needed to write into the CPLD memory using the address and data lines. On ROACH-2, we do not use the CPLD interface for programming at all, as we use the available PowerPC GPIOs. The SelectMAP configuration remains the same, it strobes GPIO pins instead of address and data lines.

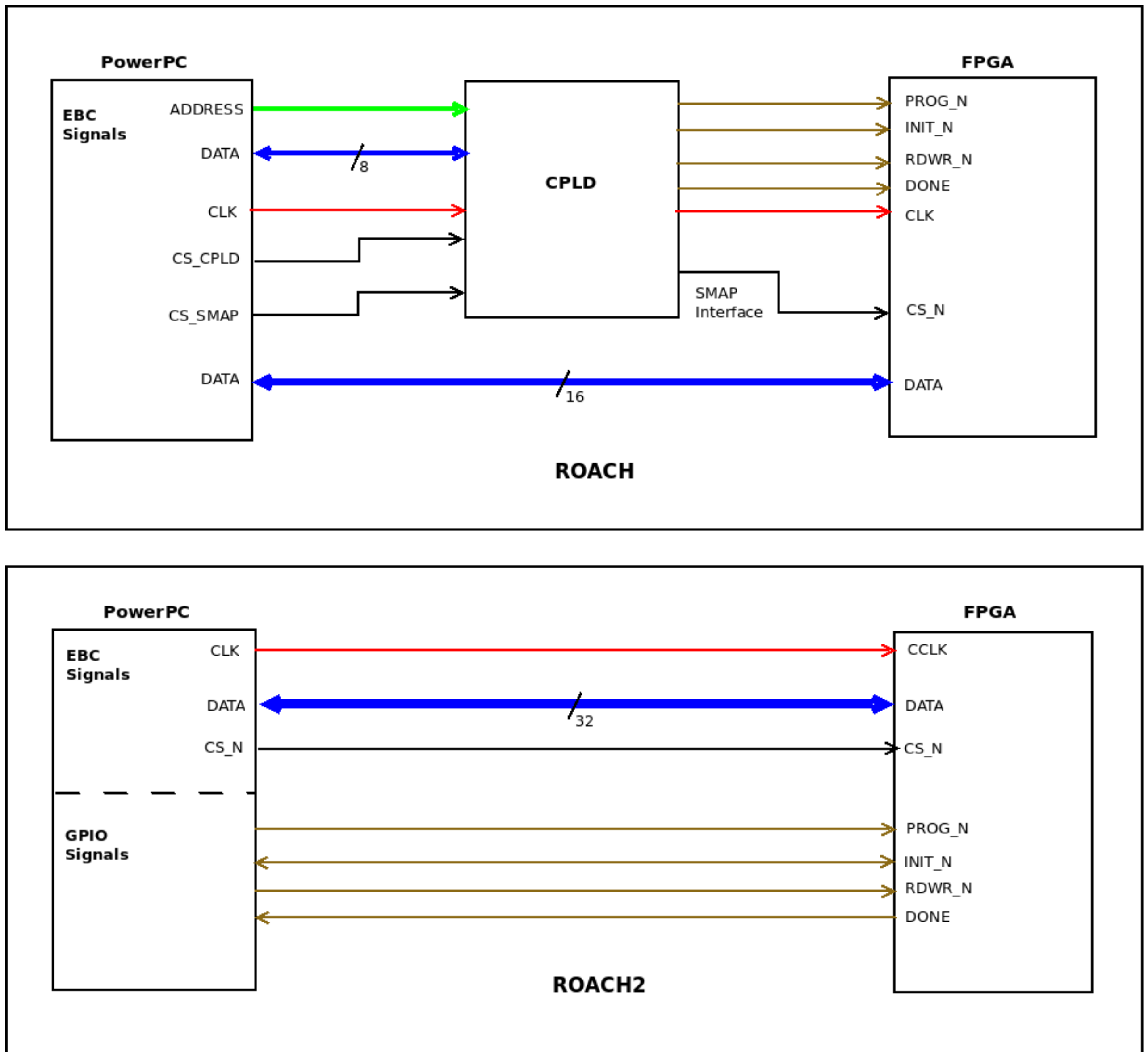


Figure 3.8: FPGA-PowerPC SelectMAP programming interface

The digital back end uses ROACH boards extensively, configured as spectrometers and correlators for KAT-7 operations in Karoo. The ROACH2 boards will be used for the advanced prototype, MeerKAT. These boards are configured by programming the FPGA and imparting them a personality as an instrument like a correlator or spectrometer. At KAT, there are two ways by which FPGAs are configured using the SelectMAP interface:

- From bootloader, U-Boot
- From kernel, Linux

3.5.2.1 From U-Boot

U-Boot[9] is a bootloader that supports multiple architectures and processors. Its primary aim is to perform basic processor and platform initialisation before handing over full control to the OS.

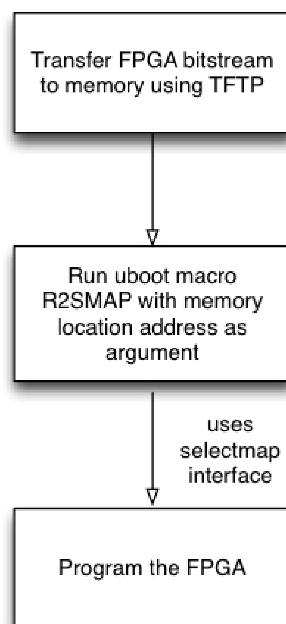


Figure 3.9: Programming FPGA from U-Boot using SelectMAP interface

Figure 3.9 illustrates the steps taken to program FPGA using the SelectMAP programming interface. U-Boot allows the user to define macros or configurable commands. One such configurable command is CONFIG_CMD_R2SMAP which uses the SelectMAP routines to configure and access the FPGA. The macros are written using C code that follows the SelectMAP configuration steps as outlined in Figure 3.7.

[9] Official name is Das U-Boot, maintained by Wolfgang Denx, hosted at www.denx.de/wiki/U-Boot

3.5.2.2 From linux

Figure 3.10 represents a generic low-level programming interface that can be used to program FPGA once the operating system, Linux takes control from U-Boot. The device driver follows the SelectMAP configuration flowchart depicted in Figure 3.7 with code containing initialisation and configuration routines.

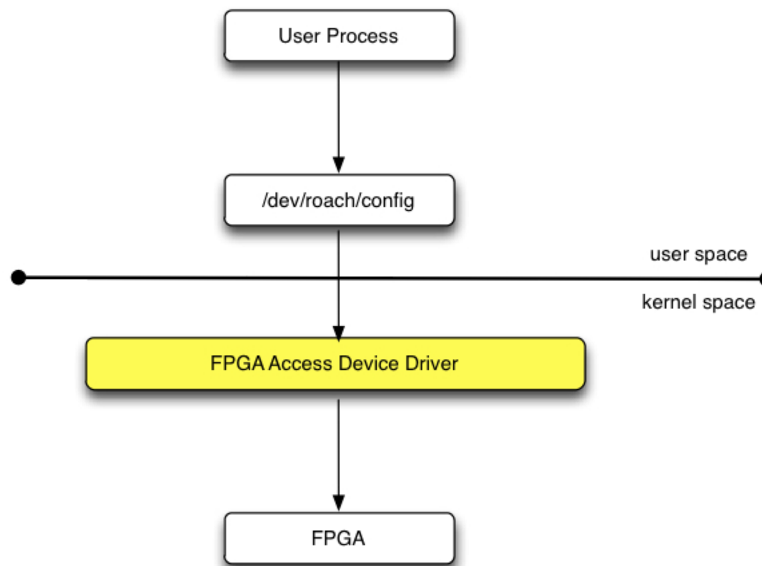


Figure 3.10: Character device driver for FPGA programming

University of

Linux is modelled on UNIX where most devices appear as files. The device driver then provides an interface layer between application and the device. The device driver model as depicted in Figure 3.10 is a character device driver[10] that handles data as serial streams of sequential data. This is best suited for the FPGA SelectMAP configuration and programming.

A character device driver with FPGA configuration capabilities takes away the complexity from the user programming the FPGA. Device drivers appear as device nodes in "/dev" directory to provide access to the FPGA. When accessing the FPGA, the kernel loads the correct device driver by mapping the major number and minor number assigned to devices. In the box below, "/dev/roach/config" device node has a major number of 252 and a minor number of 0. The major number links the device node to the device driver required for operating the device. "/dev/roach/config" node provides FPGA configuration capability.

```
crw-r--r-- 1 root root 252, 0 May 20 /dev/roach/config
```

[10] A character device driver is a type of device driver that transfers device data directly as a stream of bytes with the user process

3.5.3 Gateware Design

3.5.3.1 Gateware Toolflow

Designs that run on the FPGA are generated using the MSSGE toolflow. The CASPER approach, hardware and toolflow is described in the Masters dissertation of Peter McMahon [31].

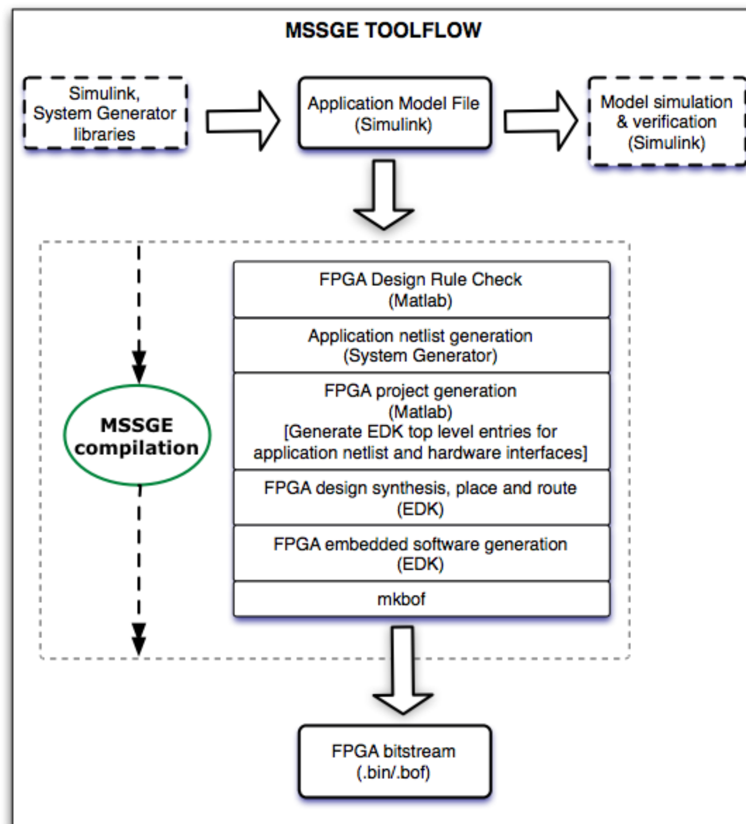


Figure 3.11: MSSGE toolflow diagram for CASPER hardware

The CASPER hardware (ibob, BEEs, ROACHes) boards are supported using MSSGE toolflow based on MATLAB's graphical modeling tool Simulink, Xilinx's System Generator and EDK. The application model file is developed in Simulink by pulling in libraries from both Simulink and System Generator. The model which represents a FPGA design is simulated and verified in Simulink.

The next stage is to compile the model file into a bitstream that can be programmed onto the FPGA. The various steps happening in MSSGE compilation stages are illustrated in Figure 3.11. EDK pulls together the bus infrastructure, pin layout and other components including the application model file under one project. Xilinx ISE tools are used to build the designs with PowerPC support under the EDK project.

3.5.3.2 Gateware Support

- ***UART Lite gateware design***

This section illustrates the design of support gateware generated using MSSGE toolflow to run on the FPGA and aids in developing the system of gateware detection using *OF* technology. Suitably designed gateware pieces are chosen to demonstrate the proof of concept that gateware programmed onto FPGA can be treated just like physical peripheral and further that it can be operated by loading an existing device driver.

University of Cape Town

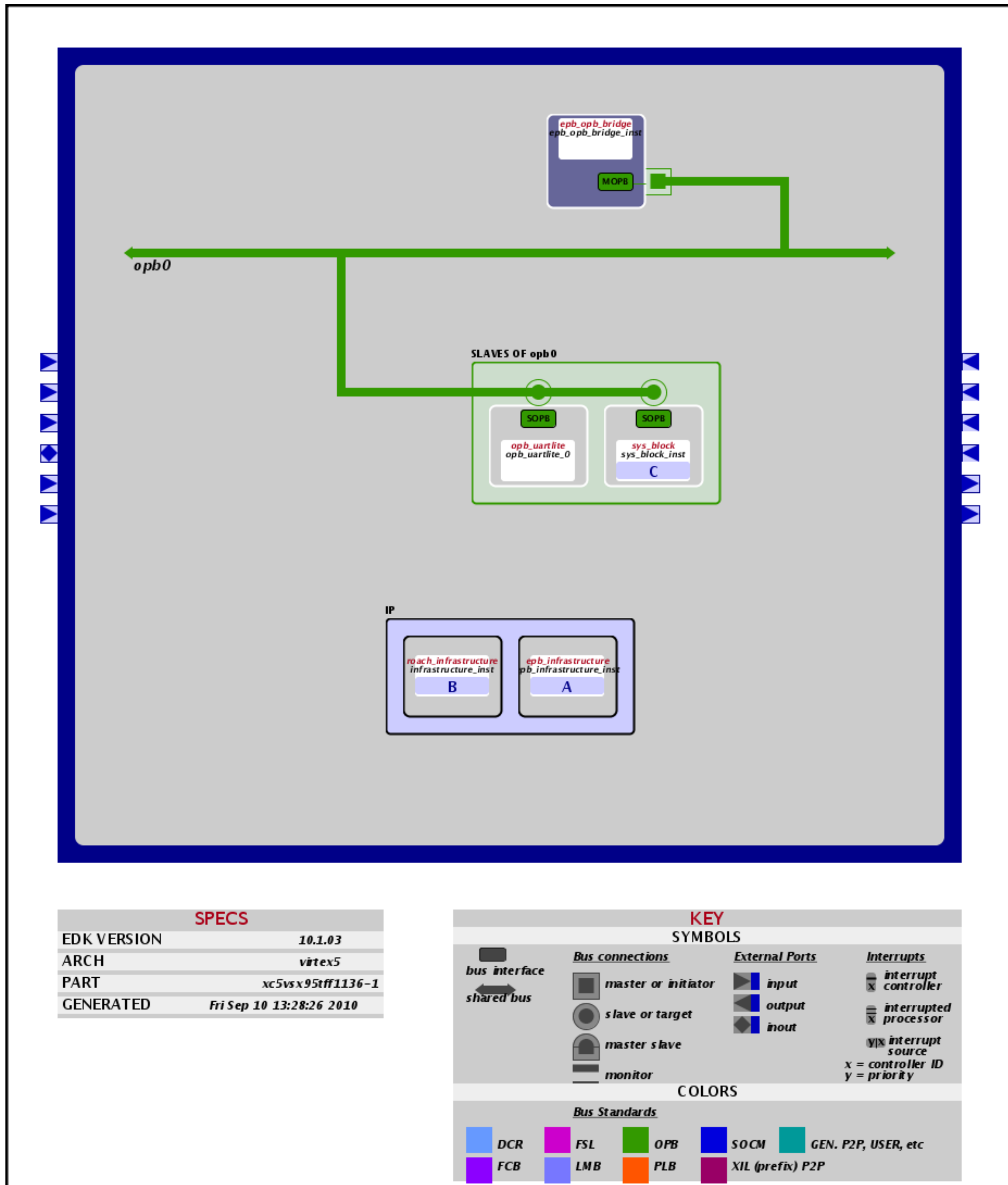


Figure 3.12: UARTLite serial OPB core connectivity in EDK

The FPGA design chosen for this purpose is a Xilinx UARTLite OPB serial core. UART Lite is a module that attaches to the OPB (On-Chip Peripheral Bus) as seen in Figure 3.12. EDK hooks together the UART Lite module pulled from library and the OPB-EPB bus infrastructure, sets pin connections where the loopback mode is configured (rx is tied to tx). ISE compiles the design and generates the bitstream with a serial loopback capability or UART personality.

The following features of the UART core were parameterized after pulling into EDK:

- Support for 8-bit bus interface
- 16-character transmit and receive FIFO
- Number of data bits set to 8
- No parity
- Baud rate set to 115200
- Interrupt routines disabled

Register access is byte wise and the data is organised in big-endian format in the registers. The status register contains errors of any during transmit and receive operation, and registers status of receive and transmit FIFO, if interrupts are enabled which in our case is disabled. The control register is used to set or clear FIFOs and enable interrupts. The address map corresponding to the registers is given in Table 3.3.

University of Cape Town

Table 3.3: UARTLite register & address Map [42]

Register Name	**Register Address**	**Description**
Receive FIFO	UART_BASE_ADDRESS + 0	Read from Receive FIFO
Transmit FIFO	UART_BASE_ADDRESS + 4	Write to transmit FIFO
Status	UART_BASE_ADDRESS + 8	Read from status regster
Control	UART_BASE_ADDRESS + 12	Write to control register

- ***ADC as audio device gateware design***

This section illustrates a FPGA design that uses a yellow block (ADC), pulled from Simulink library and connected to snapblocks[11] to capture data from the ADC. This design was developed inorder to prove the concept that radio astronomy instruments programmed on the FPGA can be represented in device tree and custom device drivers can be written to operate it. Userspace applications can then be used to display the data captured thereby eliminating the need to write custom userspace applications, hence saving time and effort.

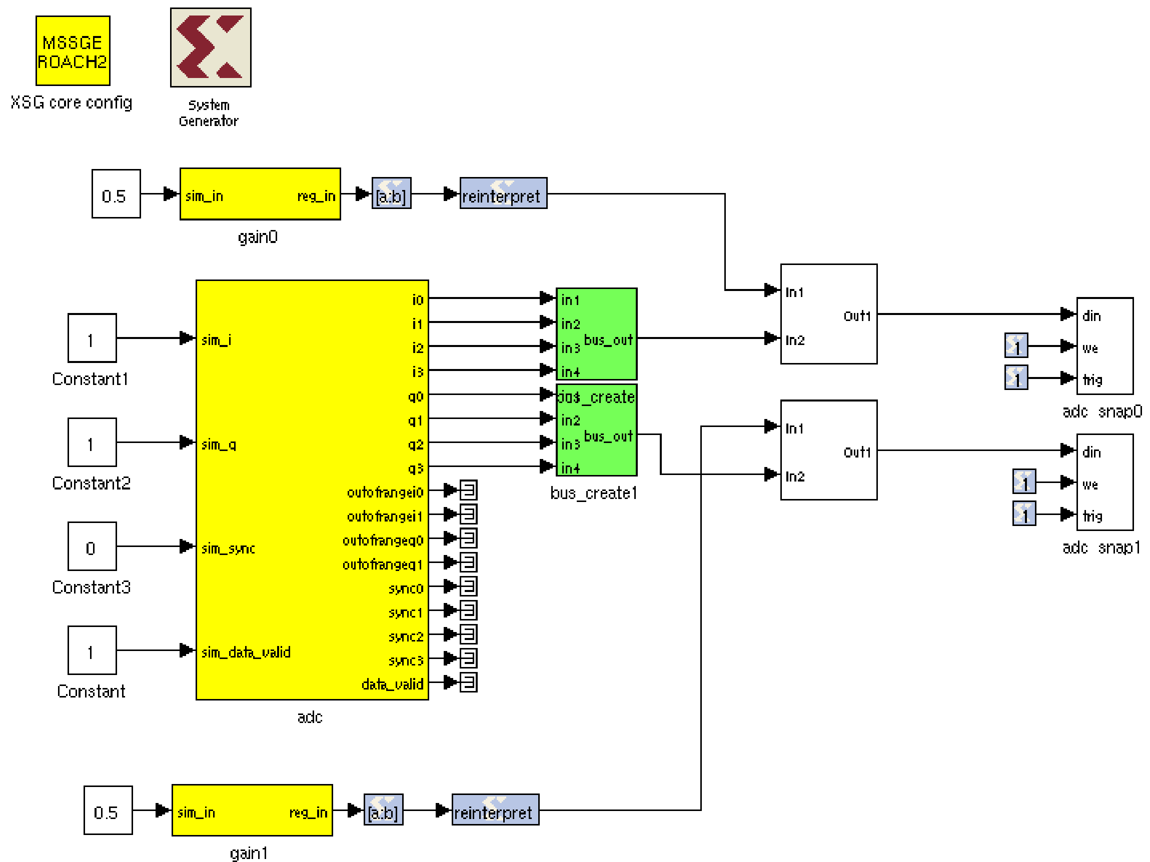


Figure 3.13: Graphical representation of sound card design using iADC in Simulink

[11]Snap blocks are generic blocks that are used throughout the digital signal processing chain to capture snapshots(in time) of data

Table 3.4: iADC sound card register & address map [1]

Register Name	Register Address	Description
adc_snap*_ctrl	0x01001000	Used to control the operation of snap block
adc_snap*_status	0x01001100	Indicates snap block status
gain*	0x01003200	14 bit signed values written to change gain
adc_snap*_bram	0x01000000	Data buffer containing captured data to be read

* represents the actual block instance number. For eg: In the above example, 0 indicates registers relating to input I and 1 indicates registers relating to input Q.

University of Cape Town

3.5.4 Device Detection Design

Device tree is the mechanism by which we pass information about an underlying platform (ROACH) to the bootloader and kernel. It is the fundamental *OF* data structure that imparts capabilities of device database extension. Device trees are also referred to as DTB (device tree blob) or FDT (flat device tree). *OF* translates the physical layout of devices including buses on a system to a tree of devices consisting of nodes. If we translate PPC440EPx functional diagram shown in Figure 3.5 to *OF* device tree, we derive the following graphical representation. Each device gets represented as a node in the tree structure. The root node consists of children including devices, buses and packages [12].

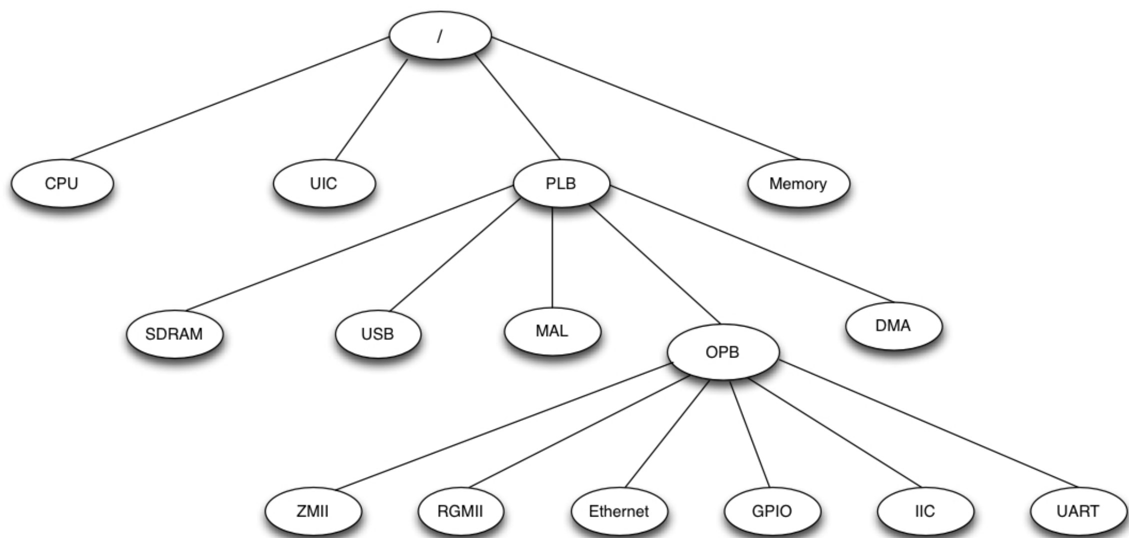


Figure 3.14: Graphical representation of PPC440EPx device tree

The nodes that are arranged hierarchically provide structure to the system. In the above figure, the PLB-OPB infrastructure hosts a number of devices that gets represented as nodes of a tree. New devices are added to this nodal structure depending on its place on the bus infrastructure thereby extending the tree of devices available to the system.

After the ppc and powerpc merge, it was made mandatory to pass underlying hardware information in the form of device tree. There are two levels through which the device trees represented in this design can be passed after it has been compiled into a FDT blob:

- From bootloader, U-Boot
- From kernel, Linux

[12]Package is referred to the set of methods, properties and data that is presented through a device node

3.5.4.1 From U-Boot

U-Boot maintains a database of hardware components on the ROACH platform in the form of device trees. The device-tree layout for U-Boot is derived from the definition of Open Firmware IEEE 1275 - 1994 device-tree. The device trees are represented in a textual format that lists the devices in the form of a tree with nodes and leaves. The nodes represent the devices and leaves the children. These textual representation is usually stored in DTS (Device Tree Source) files. There is a DTS source file requirement for the board that needs to be registered.

U-Boot passes this information to the kernel in the form of DTB (Device Tree Blob). This is the default mechanism by which device information is passed to the kernel at boot time and kernel can then initalize and manage those devices. Prior to this requirement of device tree for new board, U-Boot used to pass the underlying hardware information to the kernel as a board specific structure, but some of its details was hardcoded in the bootloader and syncing low-level information passed between bootloader and kernel was challenging. This also had the disadvantage of not being portable and parameterization was diffcult.

U-Boot uses device trees in ROACH platform for three major purposes:

1. Maintain and extend database of hardware devices
2. Runtime configuration
3. Pass, verify and modify low-level information to the kernel

- ***Maintain and extend database of hardware devices***

University of Cape Town

The bus communication between the PowerPC and the FPGA is controlled by an EBC (External Bus Controller) in the ROACH hardware platform. The EPB provides a direct attachment for most SRAM, flash memory, peripheral devices like FPGA and CPLD. The FPGA is seen as a half-word device (16-bit) attached to the EPB (External Peripheral Bus). Up to six peripheral devices can be attached to the peripheral bus.

This project emphasises on describing gateware pieces programmed on the FPGA and thereby the database of devices resident in a platform. As depicted in Figure 3.15, the EBC bus infrastructure shaded in blue is added to the device tree, and the yellow shaded blocks are the gateware implementations that we describe in device tree format. This diagram extends the device dictionary available to U-Boot and Linux for probing, detection and operation of devices through loading of appropriate device drivers.

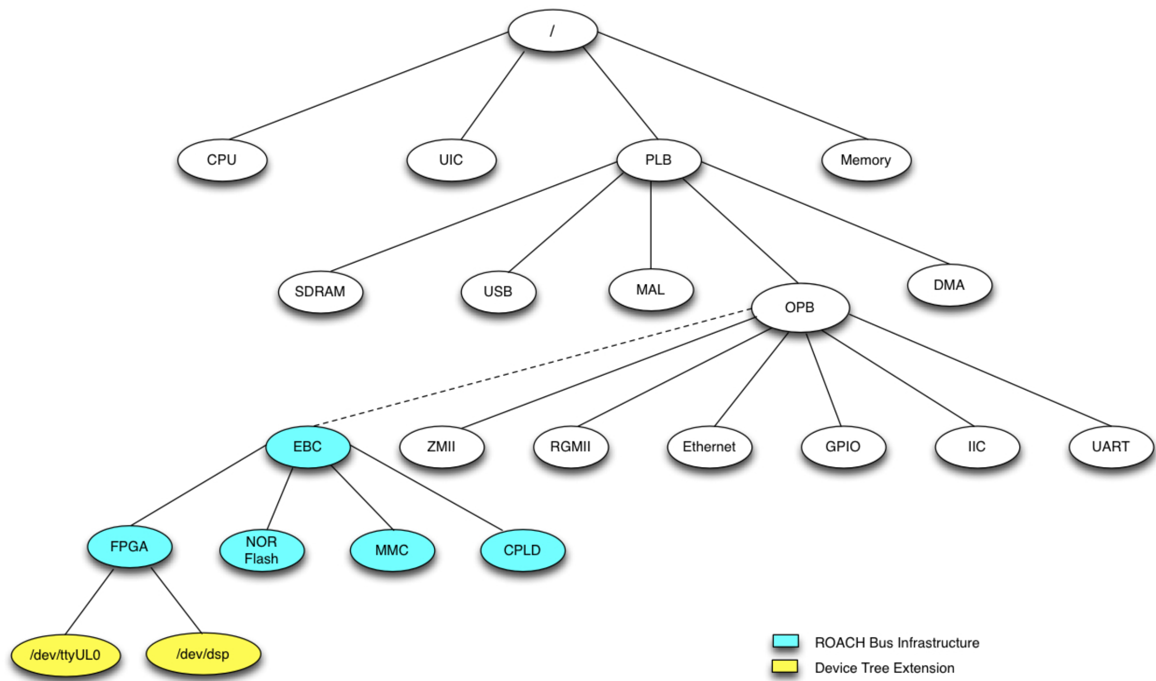


Figure 3.15: Graphical representation of ROACH device tree

- ***Runtime configuration***

U-Boot provides runtime configuration through flattened device tree (FDT). The flattened device tree can be represented in a simple text file capturing the hierarchial layout of the system. There is support infrastructure in the form of a compiler and library that enables FDT support in U-Boot.

The DTC[13] (Device Tree Compiler) compiles the textual representation to binary blob called FDT which is used by U-Boot to provide run-time support.

Compiler for device tree that accepts a device-tree in one of the given formats as indicated in Table 3.5 and outputs a compact device tree blob representation required for the kernel. In addition to converting from one format to another, it also performs sanity checks on the device tree.

Table 3.5: Device tree compiler formats

Input Formats	**Output Formats**
dtb	dts
dts	dtb
fs	asm

[13] git://jdl.com/software/dtc.git

Syntax of dtc tool is:

```
dtc [-I <input-format>] [-O <output-format>]
    [-o output-filename] [-V output_version] input_filename
```

- ***Pass, verify and modify low-level information to kernel***

U-Boot provides comprehensive set of commands for handling FDT blob. The FDT gets loaded into memory and from this location the blob can be listed and modified. The complete tree consisting of nodes can be inspected. The properties can be modified using the available set of commands. New nodes can be added, existing nodes can be removed.

This flexibility of adding new nodes and modifying existing nodes form part of our automated gateware detection mechanism where gateware pieces can be described in the form of nodes with properties and methods added that gets appended to the existing device tree and passed onto the kernel for control and operation.

3.5.4.2 From Linux

The Linux kernel has been supporting device trees for a long time, with OF being increasingly used in PowerPC platforms. *OF* provides this mechanism to the kernel to discover and register devices dynamically, thereby eliminating the need to hard code details of the underlying devices.

As the merger between ppc and powerpc happened, the kernel maintainers made it mandatory to have DT support. The equivalent DT representation in kernel called FDT was created which got passed to the kernel as a blob. Linux with OF support code probes for the devices and loads the device drivers to operate the device identified.

Linux OF functions for device detection

The *OF* supported Linux kernel makes use of device trees extensively through a 4-step process as illustrated in Figure 3.16.

1. Unflatten device tree
2. Platform identification
3. Runtime configuration
4. Device discovery and initialisation

University of Cape Town

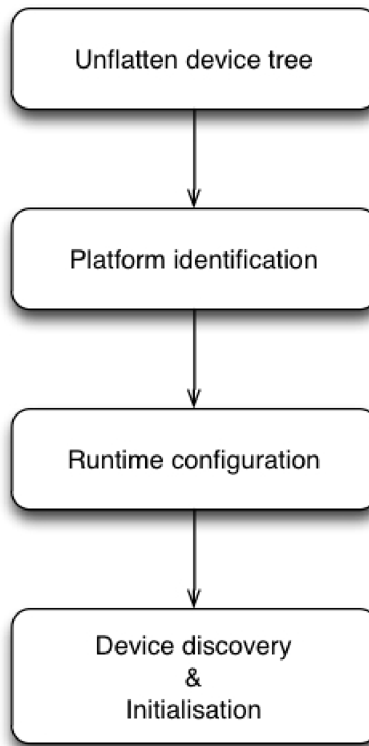


Figure 3.16: Device tree usage and detection in Linux

University of Cape Town

- **Unflatten device tree**

  The kernel entry point and pointer to device tree blob in memory is defined.

- **Platform identification**

  Kernel runs early boot code to initialise CPU and memory. It then tries to identify the platform it is running on from the device tree which gets unflattened. Machine specific setup hooks are used for the purpose of platform identification during the early boot process.

Table 3.6: Open Firmware platform identification functions

Function Call	Specific hooks	Description
probe()	init_early()	Machine specific setup
	init_irq()	Set up interrupt handling
	init_machine()	Platform data into Linux device model

Table 3.6 above lists the specific probe hooks that are involved in the platform identification process. The kernel calls *probe_machine()*, which looksup the *machine_desc* table, calling *probe()* hook for the each one. In this stage, probe() mainly checks the root node of the device tree for the *compatible* property to make a decision on whether platform support code for ROACH board is present.

- **Runtime configuration**

  Device trees can serve as communication medium between bootloader and kernel. The kernel receives runtime and configuration data like the kernel paramaters string. The configuration data is contained in the */chosen* node. Using this node, the bootargs property which contains the kernel arguments required for booting Linux is passed.

- **Device discovery and initialisation**

  The kernel obtains information about the peripheral attached to the system through device trees. The OF bus infrastructure provides a mechanism to register devices to the device model. *OF* platform populate function walks through the device tree and registers devices from nodes. Device drivers required to operate the devices in turn register a *struct of_platform_driver*. The glueing or matching of devices to device trees is handled by the *OF* platform infrastructure. The matching between devices and device drivers in done on the basis of *name, device_type* and *compatible* properties values. Thus *OF* platform code determines what devices are present by iterating through the device tree generating devices from nodes and registers it with the kernel.

Device Driver Model

Device drivers will match *compatible* property values with device nodes in a device tree. The driver will determine how to configure the device based on matching description in a device tree. The device drivers which can be built as loadable kernel modules handle the rest of configuration and device trees are only a mechanism to communicate about a device being present.

Important members of *struct platform_driver* are given below:

```
platform_driver
|– probe
|– remove
|– driver
```

Probe and remove driver attributes are responsible for the identification and removal of devices. The driver attribute contains *name*, the *OF* match table consisting of the *type* and *compatible* properties based on which the device tree node is bound with the corresponding device.

The *file operations* data structure available for the driver gets populated with code which imparts the functionality to the device. Each function call is a regular system call to the kernel requesting for an operation to happen. The operations that cannot be expressed by regular system calls can be made available through ioctl (input/output control).

University of Cape Town

This OF design concept of extending device trees for including and describing suitable gateware designs and using device drivers in Linux for operation and control forms one of the main design methodology steps in Figure 3.2.

3.5.5 Design Optimisations

3.5.5.1 Memory mapped device driver

Another by-product of this research is to interface PowerPC processor to FPGAs on ROACH boards using a Linux memory mapped device driver. Instead of making a series of system calls that involve file I/O, we memory map the FPGA to the user process address space in the external processor. In Linux, most devices appear as normal files, in this case FPGA can also be treated as a file. Memory mapping forms an association between the FPGA and the user processor memory. The process can read and write the file contents with ordinary memory access.

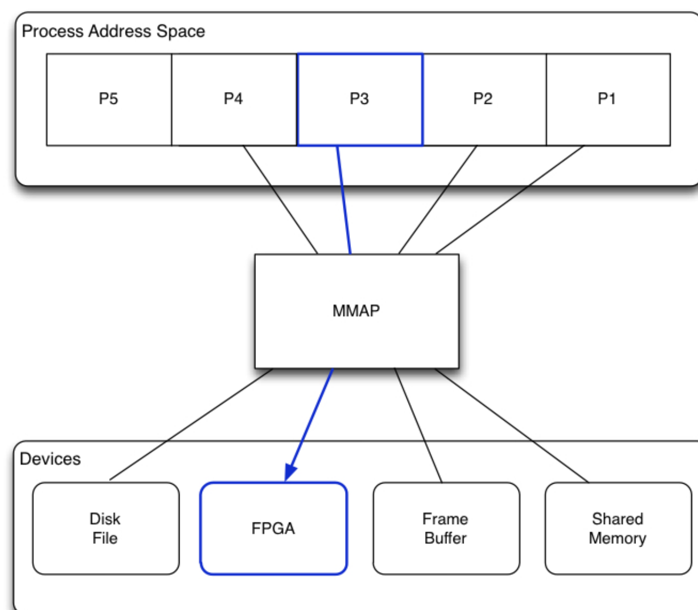


Figure 3.17: Memory mapping FPGA

The contribution of memory-mapped approach is two-fold:

1. The overhead of a system call performing I/O operations is eliminated which results in faster I/O. There are only two system calls involved. mmap and munmap. The rest is pointer level access of memory.

2. Unnecessary memory copies are not kept in the kernel. The memory region mapped is the kernels page cache, hence no need of extra copies.

3.6 Chapter Summary

This chapter provided a description of the various design and development stages towards achieving an automated gateware discovery mechanism using *OF*. The mechanism provides an alternate approach to software-FPGA interaction. It uses *OF* device trees to describe hardware information. Gateware pieces programmed onto the FPGA can be treated as pluggable peripherals and can be described in device trees just like any other hardware peripheral. The scheme of loading device drivers, existing or custom-built, is used to operate the device recognised.

The design constraints imposed on the project in the form of toolflow, hardware and software choices was discussed. The research was done on the ROACH hardware with CASPER toolflow and KAT software choices influencing the design. The bootloader U-Boot and OS, Linux supports *OF* device trees as a mechanism to pass hardware information for detection and usage.

The functional blocks in flash, powerpc and fpga are identified and the system architecture diagram was derived. The system design was composed into three stages.

First, the MSSGE toolflow used to generate gateware and the process used for gateware generation was discussed. The design information of supporting gateware required for the implementation of the project was elaborated.

Second, FPGA programming was achieved using SelectMAP interface. The interface was highlighted in relation with ROACH and ROACH2. The configuration of these boards from U-Boot and Linux using SelectMAP interface were discussed.

Third, device detection mechanism by which platform information gets passed to the bootloader and kernel was presented. The fundamental *OF* data structure, device trees was used for this purpose. The graphical illustration of how gateware nodes fit into the *OF* description of ROACH was derived. The device detection process from U-Boot and Linux were elaborated in detail.

These three stages will be integrated together in Chapter 4 using *OF* for providing an automated gateware discovery mechanism. A byproduct of this research in building a memory mapped device driver for FPGA access was also presented.

University of Cape Town

Chapter 4

Implementation and Results

This chapter elaborates on how the design components introduced in Chapter 3 gets assembled together or modified to develop an automated gateware detection system using OF. The chapter starts by introducing the development setup, software and tools used. Porting of U-Boot, the bootloader and Linux kernel with PowerPC and OF support implemented on the hardware platform, ROACH is discussed. As mentioned in design chapter, the implementation was done on two different generations of ROACH boards, ROACH and ROACH2.

The chapter expands on the high level implementation block diagram, Figure 4.3 derived from the system architecture diagram in Figure 3.3. The implementation blocks are labelled with subsequent section numbers in the high level implementation block diagram and each block will be elaborated further with examples. The gateware implementation section contains block diagrams of the implementations that reveal the internal working of the respective designs. The objective and associated test setup of the experiment is presented. We explore OF inherited device trees and ways to decribe the device in the next section namely device description. In other words, the gateware programmed on the FPGA gets associated with a personality which we describe in a device tree and pass it to U-Boot and Linux. In the device detection and enumeration section, we see how devices get passed in OF defined FDT format, how it gets enumerated and how dynamic changes can be made to the device tree. The last section emphasises the usage and development of existing or custom-built device drivers for operation and control. Userspace applications that were used to interact with the devices are listed and explained also.

The implementation blocks assembled in sequence gives rise to an automated gateware detection system using OF that can be useful for detecting FPGA designs with a personality that can be described.

University of Cape Town

4.1 Embedded Development Setup

Figure 4.1 shows the layout of the cross-development environment. This is one of the steps in the design process as illustrated in Figure 3.2.

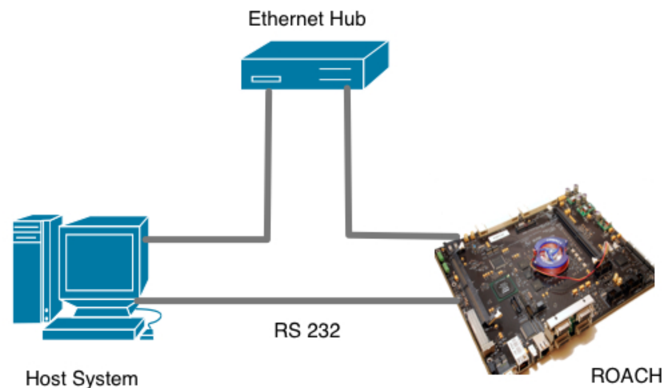


Figure 4.1: Cross-development setup for ROACH

The cross-compiler chosen after taking into consideration the design requirements and constraints is a Denx ELDK[1] (Embedded Linux Development Kit). ELDK is installed on a host[2] machine running a Debian distribution on x86_64 system. Since the target[3] hardware, ROACH has a PowerPC processor, the target architecture powerpc is selected. There is a serial terminal on the host connected to the RS232 serial port, with one or more telnet or SSH sessions to the target hardware platform.

Porting

The porting of software tools, U-Boot and Linux to ROACH platform was done by the DBE team of MeerKAT project in South Africa, mainly consisting of contributions from Marc Welz and David George. The porting was based on a similar board reference available for the PowerPC architecture namely Sequoia. Some software modifications, boot-level drivers for both U-Boot and Linux were added during the development of this project for better understanding of the software toolflow. U-Boot port for the ROACHs mainly consists of setting up the hardware and memory address maps correctly inorder to provide a basic IO system for booting Linux. Figure 4.2 shows the files that are changed or added while porting U-Boot to ROACH2. In the figure besides including board support files, additional run-time basic functionality drivers for debugging is also provided in the form of built-in commands (for eg: cmd_r2sensors.c, cmd_r2gpio.c) as mentioned in subsection 3.5.2.1.

[1] ELDK is an open-source cross-development toolchain that includes GNU cross development tools such as compilers, prebuilt target tools and libraries that provide functionality to the target system

[2] Development workstation that consists of all required tools and utilities

[3] Embedded hardware platform, in this case ROACH

University of Cape Town

```
arch/powerpc/cpu/ppc4xx/start.S       |    4 -
board/kat/roach2/Makefile             |   54 ---
board/kat/roach2/bit.c                |  535 ---
board/kat/roach2/chip_config.c        |  122 ---
board/kat/roach2/cmd_r2bit.c          |  673 ---
board/kat/roach2/cmd_r2debug.c        |   93 ---
board/kat/roach2/cmd_r2gpio.c         |  116 ---
board/kat/roach2/cmd_r2sensors.c      |  291 ---
board/kat/roach2/cmd_r2smap.c         |  161 ---
board/kat/roach2/cmd_roach2.c         |  322 ---
board/kat/roach2/cmd_roach2.h         |   47 ---
board/kat/roach2/config.mk            |   42 ---
board/kat/roach2/include/cpld.h       |   38 --
board/kat/roach2/include/fpga.h       |   74 ---
board/kat/roach2/include/gpio.h       |   35 --
board/kat/roach2/include/sensors.h    |   39 ---
board/kat/roach2/include/smap.h       |   27 --
board/kat/roach2/init.S               |   79 ---
board/kat/roach2/roach2.c             |  331 ---
board/kat/roach2/roach2_bsp.h         |   63 ---
board/kat/roach2/sdram.c              |  122 ---
board/kat/roach2/sensors.c            |  298 ---
board/kat/roach2/sensors.h            |   42 ---
boards.cfg                            |    1 -
config.mk                             |    2 +-
include/configs/roach2.h              |  524 ---
```

Figure 4.2: U-Boot port changes for ROACH2

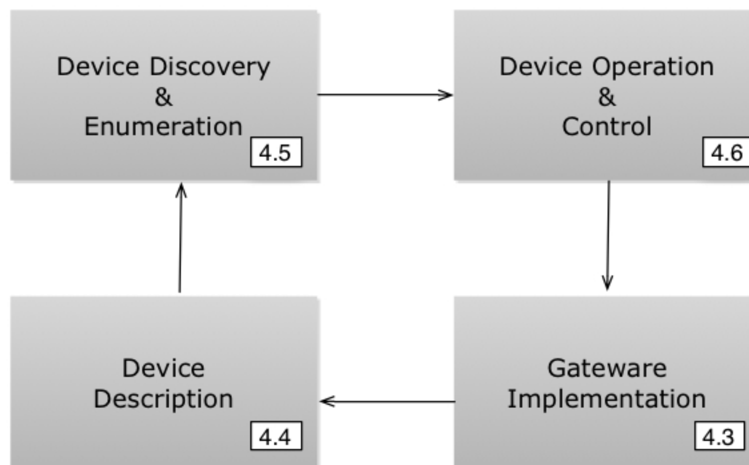
Porting of Linux to ROACH platforms is also based on the Sequoia board package. The kernel can be customised for the ROACH platform using "*make menuconfig*". Based on the configuration file generated, the kernel image gets built with the required configuration parameters for devices supported. The device tree source file for ROACH, DTS is also based on Sequoia board and the necessary changes are also made for the ROACH platforms. DTS is explained in detail in Section 4.4. It serves as a system information file that Linux uses when it boots. Apart from changes to the kernel configuration, a memory mapped device driver for FPGA access was developed and added for the ROACH2 platform. This eliminated the need to have BORPH, as the driver provided FPGA configuration and accessing capabilities at a higher speed.

For the operating system to be complete it needs a root file system inorder to provide the system with capabilities to interact with the world. A Debian root file system is provided which is in charge of providing the initial mount point for Linux as well as all initialisation and startup scripts.

University of Cape Town

4.2 High level implementation block diagram

The automated gateware detection mechanism using *OF* can be split into four distinctive blocks as shown in Figure 4.3. The high level implementation block diagram for the detection system is derived from the system architecture diagram, Figure 3.3. The implementation focuses on stitching together these four distinctive blocks to provide an infrastructure for describing gateware images with a personality, detecting the described device with methods supplemented from *OF* technology and to operate the device using standard Linux device drivers when needed. The hope is that this implementation shall be sufficiently generic to be of use in radio astronomy and other fields.



† – Each block is marked with section numbers that explains implementation.

Figure 4.3: High level implementation block diagram of automated gateware discovery mechanism using *OF*

University

The first block discusses on gateware designs implemented on the FPGA. The gateware designs generated using MSSGE toolflow gets programmed onto the FPGA. The implemented designs are simplified designs generated to serve as proof of concept. The iadc data capture gateware design is an exisiting KAT design compiled for this project by Andrew Martens[4] using MSSGE toolflow . Complicated designs are to be generated and tested in future which is outside the scope of this project. The second block inherits the idea of device trees from *OF* and extends device tree to desribe the gateware designs programmed on the FPGA. We look into how devices (gateware images with a personality) get described in the device tree. The third block implemented uses the device description information to extend the database of devices available to the bootloader and operating system. The extension / modification of the functionality of devices at the bootloader level using FDT commands is discussed. The listing, searching and modification of *OF* supported FDT is explored. The last block is in charge of binding device drivers to the device and operating the device. The device drivers are developed when required. The

[4]Digital Engineer working at SKA/KAT

aim is to use exisiting device drivers wherever possible thereby eliminating the need to write software each time for a piece of gateware image is programmed.

4.3 Gateware implementation

Figure 4.4 shows the different connections that exist between PowerPC, FPGA and interconnect buses and how the gateware design fits into the EPB-OPB bus infrastructure. EDK software is responsible for attaching the bus infrastructure to the gateware design and setting up the pin connections. The existing modules are pulled from EDK library like the OPB UARTLite module or the designed modules like BRAM blocks are setup, both attached to the OPB-EPB bus infrastructure.

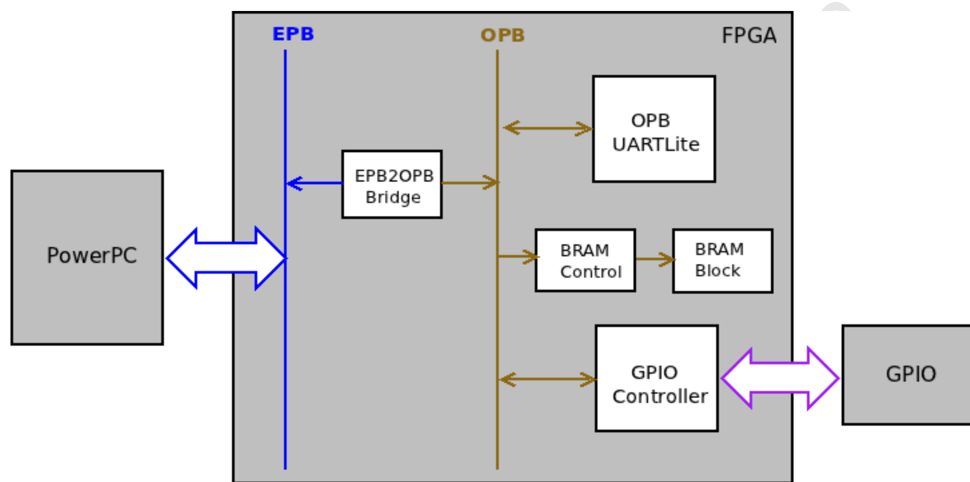


Figure 4.4: Gateware connections to PowerPC, buses and FPGA

Programming gateware on FPGA

The SelectMAP interface used for programming gateware on the FPGA is discussed in detail in the FPGA programming design subsection in 3.5.2.

- From U-Boot

  We transfer gateware design file to U-Boot memory and use user-defined U-Boot command *r2smap*[5] to program the FPGA.

```
dhcp; tftp 100000 roach2_iadc_gain.bin; r2smap 100000
```

- From Linux

  The device driver developed maps and reserves FPGA region for IO transations. If the FPGA is programmed from U-Boot, we do not need to reprogram the FPGA as

[5]Thanks to the work done by David George in developing this user-defined U-Boot command

the configuration is stored until the power is completely switched off or a hard reset is issued on the board.

The memory mapped device driver developed as a replacement for BORPH has FPGA configuration capabilities as well as read / write capabilities. More details on the driver is explained in design subsections 3.5.2.2 and 3.5.5.1. An example usage of programming the FPGA from Linux is piping the file to the device node associated with the FPGA.

```
cat roach2_iadc_gain.bin > /dev/roach/config
```

4.3.1 Serial loopback implementation

The serial loopback gateware implementation was done and tested on the ROACH platform. The FPGA design chosen was a Xilinx UARTLite OPB serial core module pulled from EDK library. Referring to the design of UARTLite core in sub-section 3.5.3.2 and table 3.3, the four registers (Control, Status, TX and RX) are placed at user-specified FPGA location which the driver and application accesses. The design is implemented such that TX and RX are tied to create a loopback mode and the test involves transmitting a character and receiving the same. The control and status registers serve for enabling interrupts / FIFOs and capture errors in the process. The FPGA design implemented can be tested from U-Boot to ensure that the design works as it is supposed to and also from Linux where proper testing is done. The implementation imparts a UART personality to the gateware design implying we can treat the design on FPGA to be a serial device to operate on.

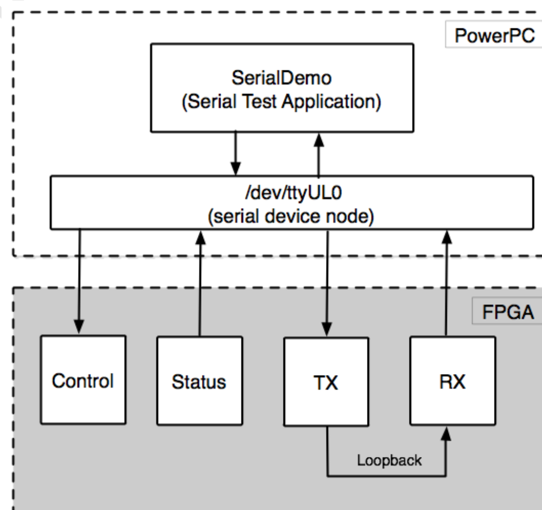


Figure 4.5: Serial design running on PowerPC and FPGA

University of Cape Town

4.3.2 Simple data capture implementation

Introduction

The data capture gateware implementation was done and tested on the ROACH2 platform.

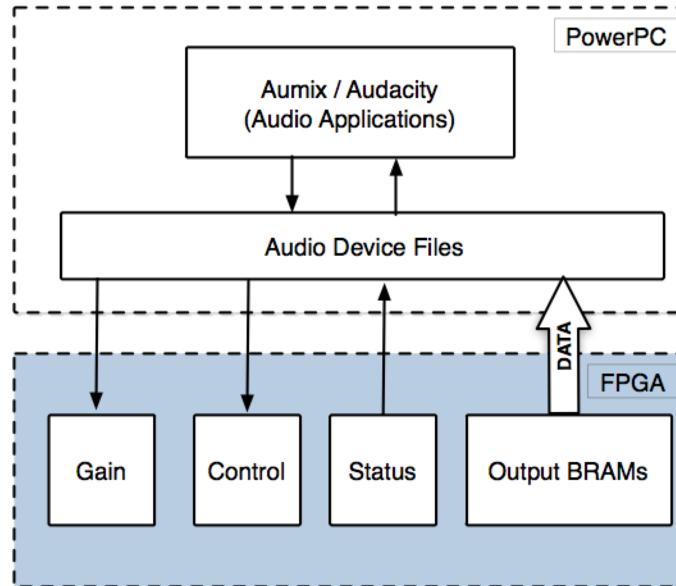


Figure 4.6: PowerPC-FPGA interface for sound capture design.

The data capture implementation emulates an audio device running on the FPGA. The iADC attached to the ROACH2 board acts as a soundcard capturing samples in this case noise and multiplying it with the gain we set before outputting the data to the snapblocks. The embedded PowerPC runs the user audio application that sets the gain and reads back the data for visualisation through the device nodes.

The data capture implementation consists of the following registers:

- gain - values written to change the gain of the audio device
- control - values written to control the operation of snapblock
- status - register that indicates status of the snapblock operation
- snapblocks - data buffer that contains the captured audio data

Objectives of measurement

The objective of this experiment is to program the FPGA with the gateware imparting it an audio device personality. The audio device captures data and stores it into BRAMs in FPGA.

Test setup

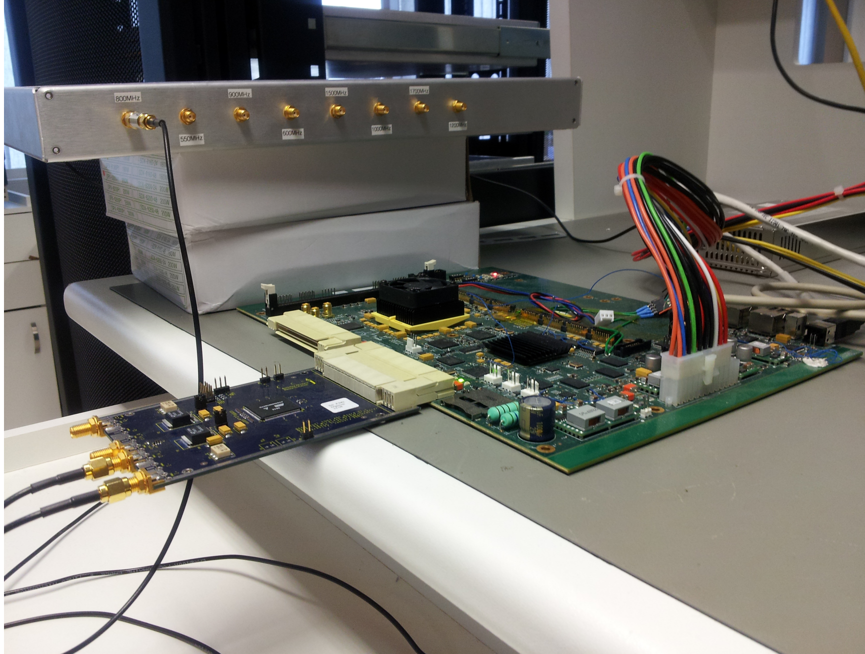


Figure 4.7: Lab setup of the experiment

The mentioned test were performed at the KAT DBE lab. The test setup for this implementation as depicted in Figure 4.7 consists of ROACH2 board, iADC board, a clock source, a host PC capable of cross-compiling applications to run on PowerPC. The iADC is connected to the ROACH2 board. The clock source is set to generate a sampling clock of frequency, 800MHz for the iADC. The input to the iADC is left free to capture noise indicating a form of signal.

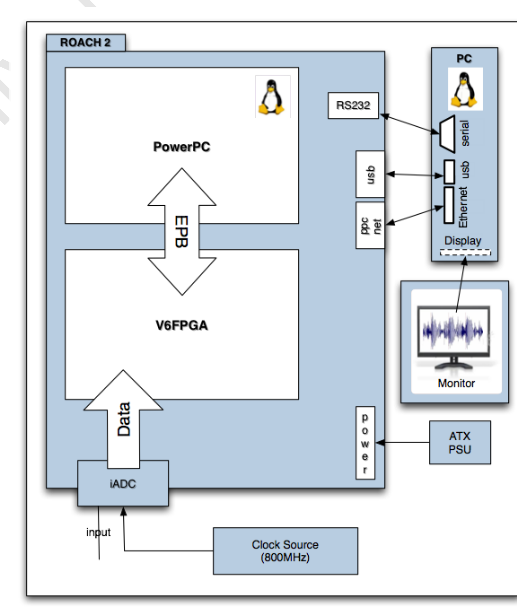


Figure 4.8: Test setup for the experiment

University of Cape Town

4.4 Device description

```
/ {
  #address-cells = <2>;
  #size-cells = <1>;
  model = "kat,roach2";
  compatible = "kat,roach2";
  dcr-parent = <&{/cpus/cpu@0}>;

  aliases {
    ethernet0 = &EMAC0;
    serial0 = &UART0;
    serial1 = &UART1;
  };

  cpus {
    #address-cells = <1>;
    #size-cells = <0>;

    cpu@0 {
      device_type = "cpu";
      model = "PowerPC,440EPx";
      reg = <0x00000000>;
      clock-frequency = <0>; /* Filled in by zImage */
      timebase-frequency = <0>; /* Filled in by zImage */
      i-cache-line-size = <32>;
      d-cache-line-size = <32>;
      i-cache-size = <32768>;
      d-cache-size = <32768>;
      dcr-controller;
      dcr-access-method = "native";
    };
  };

  memory {
    device_type = "memory";
    reg = <0x00000000 0x00000000 0x00000000>; /* Filled in by zImage */
  };
```

Figure 4.9: ROACH2 device tree snapshot

The figure above is a small snapshot of the ROACH2 device tree ported using Sequoia as platform reference. The complete listing of device tree for ROACH2 can be found in accompanying source code attachment in Appendix A. The diagram is used here for introducing some of the device tree concepts before we describe our gateware implementations using these device tree concepts. The device tree, an *OF* contribution, is a simple tree structure consisting of nodes and properties with nodes directly corresponding to devices. The respective platform device trees are stored in a .dts format under arch/powerpc/boot/dts directory. A device tree is assembled with information about the system to be used. For this implementation, to name a few we know that the ROACH2 platform consists of one 32-bit PowerPC CPU, processor local bus, interrupt controllers, four user-configurable serial ports, NOR flash and gpio controllers. We start by building a skeleton of the device tree with a unique platform name that consists of manufacturer and model name. In Figure 4.9, we see the *compatible* property has a value *"kat,roach2"*. This provides a unique identification to the device tree. Linux uses *compatible* value information to choose the right platform to run on.

The *OF* device tree conventions and ways to describe a device using it is included in A for further reference. Fundamentally every device on the ROACH platform is represented

by a device tree node which can have child nodes. Populating these nodes with information available in the form of *OF* properties is the next step. The device node hierarchy represents a parent-child relationship and with more device nodes being added the tree expands. A graphical derivation of ROACH device tree with devices listed as nodes is illustrated in Figure 3.15. Devices that are addressable uses *OF* properties *reg, #address-cells* and *#size-cells* to encode address information into the device tree. For eg in the cpus node in Figure 4.9, the *#address-cells* has a value of 1 and a value of 0, indicating that each address cell is 1 cell wide (32bits). The child nodes inherit values from the parent device node. Hence the child *cpu@0* inherits a 32bit reg value indicating a 32bit address with no size field. By convention each addressable device gets a *reg* which gets represented as tuples in the form = < [address1 length1] [address2 length2] ...>. In the above example, the reg gets a value of < [address1] >.

The *compatible* property of the root node is important as it aids the kernel to load the matching platform support code. In Figure 4.9, a *compatible* value of *"kat,roach2"* helps the Linux kernel to choose ROACH2 platform support code. The *chosen* node, the last node in the device tree, is a special node that doesnt represent any device. It stores environment information like the boot arguments or contains information to choose the default input / output devices. In our implementation, the *bootargs* environment variable value gets built in statically with information from DTS and gets passed to the kernel during boot-time. It can also be changed dynamically using U-Boot FDT commands explained in next section.

of Cape Town

```
/* UARTLite gateware description in ROACH DTS */
SERIAL_DEV: serial@d0010000{
        device_type = "serial";
        compatible = "xlnx,opb-uartlite-1.00.b";
        reg = <0xd0010000 10000>;
        current-speed = <115200>;     /* standard serial device prop */
        clock-frequency = <66666666>;
        xlnx,data-bits = <8>;
        xlnx,odd-parity = <0>;
        xlnx,use-parity = <0>;
        xlnx,family = "virtex5";
};

/* Audio device gateware description in ROACH2 DTS */
SOUND_DEV: fpga@0xd0000000{
       device_type = "audio";
       compatible = "kat,roach2-fpga";
       fpga_type = "virtex6";
       reg = <0xd0000000 0x4000000>;
  };
```

Figure 4.10: ROACH and ROACH2 Device Tree Entries

As mentioned above, data can be supplied to Linux in the form of *OF* device tree which gives the flexibility and convenience to decribe a device. Gateware implementations programmed on the FPGA are described in a similar manner with nodes and properties. The device tree entries for the respective gateware implementations is listed below. The UARTLite implementation description is similar to serial UART descriptions available where the speed, no of data bits, parity fields are mentioned. The main difference is the serial

device location where in this case the device with a serial personality is programmed on the FPGA. Additional properties can be mentioned like the *xlnx,family* which is useful for distinguishing between different FPGA families.

4.5 Device discovery and enumeration

As mentioned in design subsection 3.5.4.1, the device tree compiler, *dtc* is used to compile the assembled ROACH2 device tree source. The dtc compiler usage is obtained by issuing *"dtc help"* U-Boot built-in command. The *"dtc help"* command output is attached for reference in Appendix C. The device tree source gets built into the Linux kernel depending on the image format we choose. If we choose cuImage kernel image, the *make* process compiles roach2.dts file to FDT blob, roach2.dtb and embeds it to the kernel image. The user does not need to provide a roach2 FDT blob in this case. If we choose uImage kernel image, then the "*make uImage*" process leaves the dts source file and expects the user to provide a roach2.dtb file explicity from U-Boot.

```
dtc -I dts -O dtb -R 8 -p 0x4000 -o roach2.dtb roach2.dts
```

Figure 4.11: device tree compiler usage

The above code generates a roach2.dtb FDT blob. It takes a roach2.dts input file, makes space for the blob additions with the *"R"* and *"p"* options and outputs a roach2.dtb file for passing from U-Boot to Linux.

Minicom, a serial interactive communication program is used to log into the ROACH2 serial session. Figure 4.12 illustrates the bootloader, U-Boot running on ROACH2 board. It outputs the minimum useful information like the powerpc architecture, board name, serial number, mac address, available RAM and flash space. As mentioned in FPGA programming design subsection 3.5.2.1, we use supported built-in U-Boot commands. In order to harness the fdt capabilities supported by U-Boot, we issue *"help fdt"* to see a list of available fdt commands. As seen in Figure 4.12, the supported fdt commands are displayed.

University of Cape Town

```
Welcome to minicom 2.6.1

OPTIONS: I18n
Compiled on Feb 11 2012, 18:12:55.
Port /dev/ttyUSB2

Press CTRL-A Z for help on special keys


=> reset


U-Boot 2011.06-rc2-00000-gd422dc0-dirty (Nov 08 2012 - 16:04:14)

CPU:   AMCC PowerPC 440EPx Rev. A at 533.333 MHz (PLB=133 OPB=66 EBC=66)
       No Security/Kasumi support
       Bootstrap Option C - Boot ROM Location EBC (16 bits)
       32 kB I-Cache 32 kB D-Cache
Board: ROACH2
I2C:   ready
DRAM:  512 MiB
Flash: 128 MiB
In:    serial
Out:   serial
Err:   serial
CPLD:  2.1
USB:   Host(int phy)
SN:    ROACH2.2 batch=D#5#1 software fixups match
MAC:   02:44:01:02:05:01
DTT:   1 is 27 C
DTT:   2 is 25 C
Net:   ppc_4xx_eth0
Sensors Config
type run netboot to boot via dhcp+tftp+nfs
type run soloboot to run from flash independent of network

Hit any key to stop autoboot:  0
=> help fdt
fdt - flattened device tree utility commands

Usage:
fdt addr   <addr> [<length>]        - Set the fdt location to <addr>
fdt boardsetup                      - Do board-specific set up
fdt move   <fdt> <newaddr> <length> - Copy the fdt to <addr> and make it active
fdt resize                          - Resize fdt to size + padding to 4k addr
fdt print  <path> [<prop>]          - Recursive print starting at <path>
fdt list   <path> [<prop>]          - Print one level starting at <path>
fdt set    <path> <prop> [<val>]    - Set <property> [to <val>]
fdt mknode <path> <node>            - Create a new node after <path>
fdt rm     <path> [<prop>]          - Delete the node or <property>
fdt header                          - Display header info
fdt bootcpu <id>                    - Set boot cpuid
fdt memory <addr> <size>            - Add/Update memory node
fdt rsvmem print                    - Show current mem reserves
fdt rsvmem add <addr> <size>        - Add a mem reserve
fdt rsvmem delete <index>           - Delete a mem reserves
fdt chosen [<start> <end>]          - Add/update the /chosen branch in the tree
                                        <start>/<end> - initrd start/end addr
NOTE: Dereference aliases by omiting the leading '/', e.g. fdt print ethernet0.
=>
```

Figure 4.12: U-Boot on ROACH2

```
=> setenv fdt_addr 0x400000
=> setenv fdt_blob roach2.dtb
=> tftp ${fdt_addr} ${fdt_blob}
ENET Speed is 1000 Mbps - FULL duplex connection (EMAC0)
Using ppc_4xx_eth0 device
TFTP from server 192.168.40.1; our IP address is 192.168.40.57
Filename 'roach2.dtb'.
Load address: 0x400000
Loading: ##
done
Bytes transferred = 23505 (5bd1 hex)
=> fdt addr ${fdt_addr}
=> fdt list /
/ {
        #address-cells = <0x2>;
        #size-cells = <0x1>;
        model = "kat,roach2";
        compatible = "kat,roach2";
        dcr-parent = <0x1>;
        aliases {
        };
        cpus {
        };
        memory {
        };
        interrupt-controller0 {
        };
        interrupt-controller1 {
        };
        interrupt-controller2 {
        };
        sdr {
        };
        cpr {
        };
        plb {
        };
        chosen {
        };
};
=>
```

Figure 4.13: FDT blob listing in U-Boot

University

The above Figure 4.13 demonstrates the usage of the FDT blob generated using DTC compiler. From U-Boot, the environment variables *fdt_addr* and *fdt_blob* are assigned values using *"setenv"* U-Boot built-in command. The U-Boot built-in command *"saveenv"* saves the defined environment variables in flash, thereby not needing to define it everytime we are in U-Boot. A tftp transfer initiates transfer of the FDT file to the memory address specified in *fdt_addr*. The fdt command "*fdt addr*" chooses / selects the FDT blob stored in memory location specified by *fdt_addr* in U-Boot. Failing to issue *"fdt addr"* will result in the remaining fdt commands to display nothing as we havent selected a FDT blob to work on. We can list the device tree entries from a top-most level by issuing the fdt command *"fdt list /"* where *"/"* stands for the root node. As listed in Figure 4.13, it is evident that its a roach2 platform by inspecting the *compatible* property of the root node. *"fdt chosen"* U-Boot command allows the user to pass dynamic information about the environment and peripherals. It forces the bootloader to include the chosen node to the FDT blob with the *bootargs* value set by the user.

```
=> fdt print /plb/opb/serial
serial@ef600300 {
        device_type = "serial";
        compatible = "ns16550";
        reg = <0xef600300 0x8>;
        virtual-reg = <0xef600300>;
        clock-frequency = <0x0>;
        current-speed = <0x1c200>;
        interrupt-parent = <0x2>;
        interrupts = <0x0 0x4>;
};
=> fdt mknode /plb/opb SERIAL_DEV@d0010000
=> fdt set /plb/opb/SERIAL_DEV device_type "serial"
=> fdt set /plb/opb/SERIAL_DEV reg <0xd0010000 0x10000>
=> fdt print /plb/opb/SERIAL_DEV
SERIAL_DEV@d0010000 {
        reg = <0xd0010000 0x10000>;
        device_type = "serial";
};
=> fdt set /plb/opb/SERIAL_DEV compatible "xlnx,opb-uartlite-1.00.b"
=> fdt set /plb/opb/SERIAL_DEV current-speed 115200
=> fdt print /plb/opb/SERIAL_DEV
SERIAL_DEV@d0010000 {
        current-speed = "115200";
        compatible = "xlnx,opb-uartlite-1.00.b";
        reg = <0xd0010000 0x10000>;
        device_type = "serial";
};
=>
```

Figure 4.14: Adding a device dynamically on FDT blob from U-Boot

The extension of device tree source dynamically based on *OF* properties and methods is illustrated in Figure 4.14. In our example we want to enter dynamic information on serial Xilinx UARTlite device "SERIAL_DEV" by parsing through the FDT blob. We can either enter the serial device UARTlite entry into dts source and pass it to the kernel or we can add the serial device entry on the fly by inspecting similar serial device entries. *"fdt print /plb/opb/serial"* lists the properties associated with the device n16550 serial port. We create a new device by issuing *"fdt mknode"*, set properties and values for the node created using *"fdt set"*. We can confirm the node creation and its associated properties by listing the device by issuing fdt print *"/plb/opb/SERIAL_DEV"*.

University of Cape

4.6 Device operation and control

4.6.1 Device Drivers

U-Boot passes the FDT blob along with the kernel image that runs on PowerPC. Linux kernel uses *OF* functions to discover and register devices dynamically with the information from the FDT. Most of the physical devices registered are operated with a device driver. Device driver will match with compatible property and will determine how to configure the device based on the matching description in the device tree. The standard way device drivers bind with a device is through special files called character device files. We use *mknod* linux command to create these special files called device nodes. A listing of available character devices can be found by issuing *"cat /proc/devices"* from the terminal running on roach2. In our examples, we need to build the special device nodes for serial UARTlite example and audio device example using mknod which is shown below:

```
/* UARTLite device file */
mknod /dev/ttyUL0 c 204 187

/* Audio device files */
mknod /dev/mixer  c  14   0
mknod /dev/dsp    c  14   3
```

Figure 4.15: Creating required character device files

In the above listing, *"c"* refers to the character device to be created followed by major and minor numbers respectively. The created files and the kernel driver is linked using this major and minor number.

The implementations we have are of two types:

- One that uses existing device drivers to operate
- One that needs custom-built device driver to operate

Builtin serial UARTLite driver

Figure 4.16 shows the output of the UARTlite device driver with debug messages enabled while the serial device being operated. The output of the driver with and without gateware bearing serial personality is displayed for comparison. The sample output with serial bit file programmed demonstrates the behavior of a serial device with loopback mode enabled and interrupts disabled. The transmitted character "A" is displayed from the receive buffer. The ULITE_STATUS_RXVALID flag becomes active upon receiving the character.

```
/*Sample Output with serial bit file not programmed*/
root@192:/tmp# mknod ttyUL0 c 204 187
root@192:/tmp# echo ab > /tmp/ttyUL0
ulite_startup():Value[03] written to control register MMIO [f106a00c]
ulite_set_termios(), hardcoding baud rate = 115200
ulite_start_tx():About to transmit:Passing status[f106a008] = 0
ulite_transmit():MMIO[f106a008]with values received: status = 0, character to transmit = A[41]
ulite_start_tx():About to transmit:Passing status[f106a008] = 0
ulite_transmit():MMIO[f106a008]with values received: status = 0, character to transmit = A[41]
ulite_start_tx():About to transmit:Passing status[f106a008] = 0
ulite_transmit():MMIO[f106a008]with values received: status = 0, character to transmit = A[41]
ulite_tx_empty():MMIO 0xf106a008, status = 0
ulite_tx_empty():MMIO 0xf106a008, status = 0

/*Sample Output with serial bit file programmed*/
root@192:/tmp# echo ab > /tmp/ttyUL0
ulite_startup():Value[03] written to control register MMIO [f106a00c]
ulite_set_termios(), hardcoding baud rate = 115200
ulite_start_tx():About to transmit:Passing status[f106a008] = 4
ulite_transmit():MMIO[f106a008]with values received: status = 4, character to transmit = A[41]
ulite_start_tx():About to transmit:Passing status[f106a008] = 0
ulite_transmit():MMIO[f106a008]with values received: status = 5, character to transmit = A[41]
ulite_receive():RX_VALID:character received from location[f106a000] is A
ulite_receive():ULITE_STATUS_RXVALID:1
ulite_start_tx():About to transmit:Passing status[f106a008] = 0
ulite_transmit():MMIO[f106a008]with values received: status = 5, character to transmit = A[41]
ulite_receive():RX_VALID:character received from location[f106a000] is A
ulite_receive():ULITE_STATUS_RXVALID:1
ulite_tx_empty():MMIO 0xf106a008, status = 0
ulite_tx_empty():MMIO 0xf106a008, status = 0
```

Figure 4.16: UARTlite device driver log from Linux

University of Cape

Custom built audio driver

A gateware with an audio device personality is programmed on the FPGA. The context of programming audio device drivers in Linux applies now. The iADC acts as a capture device and we need a device file namely *"/dev/dsp"* to listen to the audio, in radio astronomy terms, signal or noise. We want to tune an audio device with gain setting, in radio astronomy terms we want to amplify or reduce the signal strength by setting the gain value. The main function of a mixer in audio devices is to set gain level. This also requires an audio device file, *"/dev/mixer"*. The Linux open source community has many device drivers that one can use as template to get started and therefore writing device drivers isnt too difficult. Reading the *"/dev/dsp"* device file activates the A/D converter for signal capturing. Figure 4.8 shows iADC connected to ROACH2 board. Analog data is converted to digital samples and stored into BRAMs. When a sound application program like aumix tries to use the mixer device, the data stored in BRAMs is read into the application programs data buffer for display.

If a device has to perform more functions apart from data transfer, Linux provides a method namely ioctl. The various other audio settings that dont fit into read / write system calls can be adjusted using ioctls. All ioctl calls to *"/dev/dsp"* are names prefixed with SOUND_PCM and all the mixer ioctl commands are prefixed with SOUND_MIXER. Gain level can be set using MIXER_WRITE macro. The macro MIXER_READ gives the current level setting of the device. The device driver source code for this detected audio device is available in the accompanying DVD and uses the MIXER_READ and

MIXER_WRITE macros for reading / setting the gain levels.

Table 4.1: Audio device driver events and associated functions

Events	User functions	Kernel functions	Brief explanation
Load module	insmod	bram_init()	Allocating and initialising each device Setting up these character devices Mapping and reserving FPGA
Open device	application	bram_open()	Opening the audio device file Setting initial iADC default values
Read device	application	bram_read()	Read the captured data from snapblocks
Write device	application	bram_write()	Not applicable as audio capture only
Other	application	bram_ioctl()	General audio settings;Setting gain
Close device	application	bram_release()	Releasing the device file
Remove module	rmmod	bram_exit()	Deallocating memory regions Releasing character device files

The kernel functions assembled together in Table 4.1 provides a complete audio device driver for controlling the gateware programmed on the FPGA with an audio device personality. One of the great features of Linux is that it provides capability to load and unload driver from userspace during run-time. This is very useful for testing the driver and its associated functions. This is accomplished using the Linux commands *insmod* and *rmmod*. The equivalent kernel functions for insmod and rmmod command, *bram_init()* and *bram_exit()* can be seen in Table 4.1.

As explained in design chapter 3 in subsection 3.5.4.2, the file operations structure in the device driver gets populated with code that imparts functionality to the device. Figure 4.17 lists the file operations structure used for writing audio device driver to control the gateware programmed with audio device functionality.

```
static struct file_operations bram_fops = {
  .owner          =   THIS_MODULE,
  .read           =   bram_read,
  .write          =   bram_write,
  .unlocked_ioctl =   bram_ioctl,
  .open           =   bram_open,
  .release        =   bram_release,
};
```

Figure 4.17: Audio file operations structure

The aumix operation device driver log output in Figure 4.18 demonstrates that as we change the mixer levels, the gain value changes and the data also changes proportionally. The device driver is enabled with maximum debug messages and hence the descriptive output. As seen from the figure, the aumix application sets the gain using its controls and the corresponding value is used as the input gain value. The SOUND_MIXER_WRITE_IGAIN macro sets this input gain value. The initial input gain value can be obtained through a device tree entry that can be searched using *OF* node search functions.

```
bram_init:FPGA I/O memory range 0x1d0000000 to 0x1d4000000 allocated (virt:0xf9100000)
bram_init: Gateware control kernel module loaded
bram_open: Opening mixer
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 0
bram_ioctl:CHECK:CHECK Gain is(00000404)
SOUND_MIXER_WRITE_IGAIN: GAIN[1028:404]:unsigned value=1028, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 1028
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 1028
bram_ioctl:CHECK:CHECK Gain is(00000808)
SOUND_MIXER_WRITE_IGAIN: GAIN[2056:808]:unsigned value=2056, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 2056
bram_ioctl:MIXER:cmd[0x40044dff] not matching:ordinal number is 255,dev is 0
bram_ioctl:MIXER:cmd[0x40044d00] not matching:ordinal number is 0,dev is 0
bram_ioctl:MIXER:cmd[0x40044dff] not matching:ordinal number is 255,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 2056
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 2056
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 2056
bram_ioctl:CHECK:CHECK Gain is(00000c0c)
SOUND_MIXER_WRITE_IGAIN: GAIN[3084:c0c]:unsigned value=3084, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 3084
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 3084
bram_ioctl:CHECK:CHECK Gain is(00001010)
SOUND_MIXER_WRITE_IGAIN: GAIN[4112:1010]:unsigned value=4112, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 4112
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 4112
bram_ioctl:CHECK:CHECK Gain is(00001414)
SOUND_MIXER_WRITE_IGAIN: GAIN[5140:1414]:unsigned value=5140, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 5140
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 5140
bram_ioctl:CHECK:CHECK Gain is(00001818)
SOUND_MIXER_WRITE_IGAIN: GAIN[6168:1818]:unsigned value=6168, ADC_GAIN:loc[01003200]
bram_ioctl:MIXER:cmd[0xc0044d0c] not matching:ordinal number is 12,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 6168
bram_ioctl:MIXER:cmd[0x40044dff] not matching:ordinal number is 255,dev is 0
bram_ioctl:MIXER:cmd[0x40044d00] not matching:ordinal number is 0,dev is 0
bram_ioctl:MIXER:cmd[0x40044dff] not matching:ordinal number is 255,dev is 0
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 6168
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 6168
SOUND_MIXER_READ_IGAIN: Value of tmp to be written is 6168
bram_release: Releasing mixer
```

Figure 4.18: Aumix operation device driver log

4.6.2 Userspace Applications

serial app

A serial application we developed demonstrates that a character transmitted is received in loopback mode was developed. It is included in the source code attachment in Appendix A.

aumix

Aumix is an audio application that operates the mixer from the terminal in Linux. This application was used for controlling the gain for operating the gateware with audio device personality thereby eliminating the need to write userspace applications.

strace

strace is a valuable debugging utility that displays all the system calls issued to the kernel and the signals associated with it.

A sample output of strace is displayed below when using aumix application for setting the gain levels:

```
execve("/usr/bin/aumix", ["aumix", "-i55"], [/* 11 vars */]) = 0
brk(0)                                  = 0x1001d000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x48020000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=28417, ...}) = 0
mmap(NULL, 28417, PROT_READ, MAP_PRIVATE, 3, 0) = 0x48023000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/usr/lib/libgpm.so.2", O_RDONLY)  = 3
read(3, "\177ELF\1\2\1\0\0\0\0\0\0\0\0\0\0\3\0\24\0\0\0\1\0\0\30\300\0\0\0004"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=22956, ...}) = 0
mmap(0xffda000, 87872, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xffda000
mprotect(0xffe0000, 61440, PROT_NONE)   = 0
mmap(0xffef000, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x5000) = 0xffef000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libncurses.so.5", O_RDONLY)  = 3
read(3, "\177ELF\1\2\1\0\0\0\0\0\0\0\0\0\0\3\0\24\0\0\0\1\0\0\300\0\0\0\0004"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=273552, ...}) = 0
```

Figure 4.19: Strace sample output

The above fragment is only a small piece of strace output displayed after running aumix on the audio device gateware. It shows that it opens the application aumix and related libraries inorder to run it. This utility was helpful in identifying the ioctls that aumix is interested and helped in speeding up device driver development.

dmesg

A useful troubleshooting Linux command for displaying kernel buffer messages.

Cape Town

Table 4.2: Summary table of the *OF* implementation done for automating gateware detection

Gateware	**Device Detected**	**Kernel device driver**	**API**	**Output**
UARTLite†	Serial device	Built-in driver	Serial device API provided	Achieved
ADC as audio device *	Audio device	Custom built driver	Aumix (Existing audio API)	Achieved

† – Existing UARTLite module from Xilinx
* – Gateware contributed by A Martens [1]

University of Cape Town

4.7 Chapter Summary

This chapter provides the implementation of the various design blocks for creating an automated gateware detection mechanism using *OF*. The embedded development environment and the porting of U-Boot and Linux to the ROACH platform is mentioned. Table 4.2 provides a summary of the automated gateware discovery work that has been implemented and tested successfully.

The high level block diagram of automated gateware detection mechanism using *OF* was drawn with each block representing a stage in the implementation. Firstly, we look into the gateware implementation stage on the FPGA. The serial device gateware implementation was done with the intention to demonstrate that gateware implementations can be treated as any other physical device attached to the system and using *OF* describe it in a device tree and operate it using existing serial device driver. The data capture gateware implementation was done with the aim to demonstrate that devices with different personalities can be detected, described using *OF* and operated using a custom-built device driver.

The subsequent stages explore how to use *OF* inherited device trees to describe devices in the form of gateware implementations. The gateware implementations are described in OF device tree format, DTS. The devices are listed and explored from U-Boot using FDT commands. Devices can be added and removed on the fly using FDT built-in U-Boot commands. The FDT is passed to the kernel which it uses for device enumeration and maps device drivers to operate on the devices discovered.

The last stage of providing device drivers for operation and control was implemented with the intention to utilise existing device drivers and userspace applications thereby reducing the effort to write software. The serial UARTLite example uses an existing UARTLite driver for operation and control. Custom-built device drivers are developed with reduced effort as many examples relating to the personality of gateware are available. The audio device driver developed is one such example. Userspace applications that were of use during the course of implementation is discussed.

The implementation, tests and associated results of the implementation demonstrates two important concepts that 1) gateware images programmed on the FPGAs can be treated just like any other physical peripheral and that it can be described in the format of an *OF* inherited device tree format 2) the conventional method of binding device drivers (existing or custom-built) to devices can be utilised here through *OF* infrastructure, thereby reducing the effort for software development.

University of Cape Town

Chapter 5

Conclusions and Future Directions

5.1 Summary

This dissertation can be summarised into the following:

- Literature reviews on device detection for conventional computer platforms and FPGA based software systems were presented. The reasons to choose Open Firmware technology for this dissertation was done from the review.
- The finer objectives enumerated in Section 3.2 have been validated and the user specifications have been met.
- A detailed system architecture diagram was drawn to illustrate the various stages of the design.
- The results of the tests were discussed to validate categories of gateware detection, loading correct device drivers and extension of OF infrastructure for probing hardware devices.
- Another by product of this research, a memory mapped device driver for FPGA configuration and access was also developed.

5.2 Conclusions

In this dissertation we have described the development of an automated gateware detection systems using *OF* that can be used for projects relying heavily on FPGA. This dissertation studied some of the device detection mechanisms that are available for conventional computer platforms and FPGA based software systems. The reasons to choose *OF* and Linux

University of Cape Town

for automating gateware detection on FPGA based reconfigurable hardware platform is presented.

A system architecture design diagram was derived to elaborate the different design stages involved in building a gateware detection system. The embedded development enviroment and associated test setup for developing an automated gateware detection system were mentioned. The details of implementation of a serial device and audio device and the analysis of the results obtained were presented. The serial device implementation establishes the concept that gateware programmed on the FPGA can be treated just like any other physical peripheral, *OF* device trees can be extended to describe the gateware and further it can be operated on by an existing device driver that gets loaded by the Linux kernel during run-time. The audio device implementation establishes the concept that *OF* probe and infrastructure can be utilised to load custom device drivers for operating instruments programmed on the FPGA.

Thus this dissertation provides the basic infrastructure for device detection in FPGA systems using *OF* and Linux.

5.3 Dissemination Strategy

The work implemented in this dissertation has been presented at CASPER [33, 35] and SKA SA bursary conferences [32, 34]. The memory mapped device driver work has replaced the BORPH method as an alternate means of communication with the FPGA. The MeerKAT project has implemented this work on the ROACH2 board.

The device driver work has been submitted to be included in the standard Linux PowerPC kernel inorder to streamline support and maintenance. The device driver source code has been made available through SKA SA Github[1]. Further, paper and poster presentations for various conferences are also planned.

5.4 Recommendations for future work

The present implementation emphasises on providing the basic infrastructure for building an automated gateware detection system using *OF*. We used simple gateware designs like the serial and audio implementations for demonstrating proof of concept of device detection. It will be interesting and useful to extend the infrastructure by testing complex gateware designs, especially instrument designs that are used for radio astronomy like correlator and spectrometer. More contributions in the form of Linux kernel device drivers

[1]https://github.com/ska-sa

University of Cape Town

to support these instrument designs would also be useful not only for radio astronomy but also for the open source community. Another important contribution can be writing the initialisation code like the timing and synchronisation infrastructure used for controlling signal processing designs in Forth, thereby reducing the software logic required to control various radio astronomy instruments.

Appendix A

Source Code

This DVD attachment is included with the dissertation and gives all the source code, project files and documentation not included in the written dissertation.

The folder structure organised in the DVD is as follows:

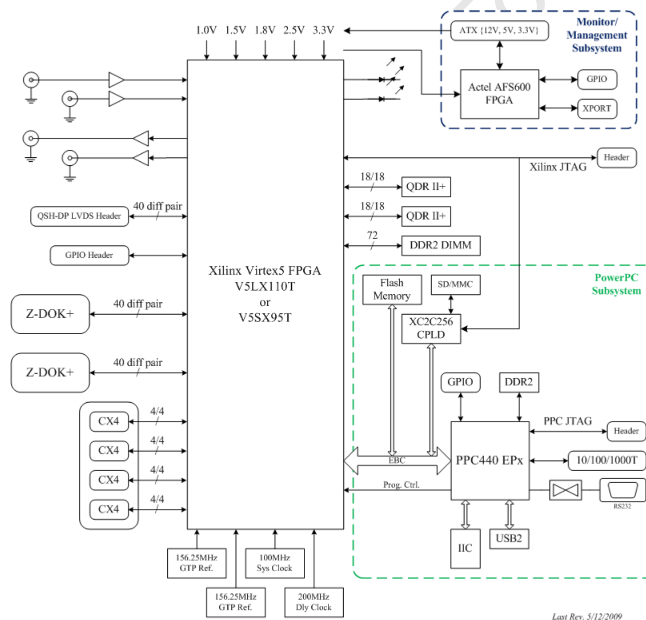
- Source Code (Contains all the project files relevant to the research)
  - Device Drivers (Driver developed for supporting gateware designs)
  - ROACH (U-Boot, Linux and gateware support files)
    - Linux
    - U-Boot
    - gateware
    - roach.dts
  - ROACH2 (U-Boot, Linux with memory mapped device driver, gateware support)
    - Linux
    - U-Boot
    - gateware
    - roach2.dts
  - Utilities
    - Serial Application
- Images (Some ROACH and ROACH2 images)
- Main_Dissertation-S_RAJAN.pdf (Main dissertation in pdf format)

University of Cape Town

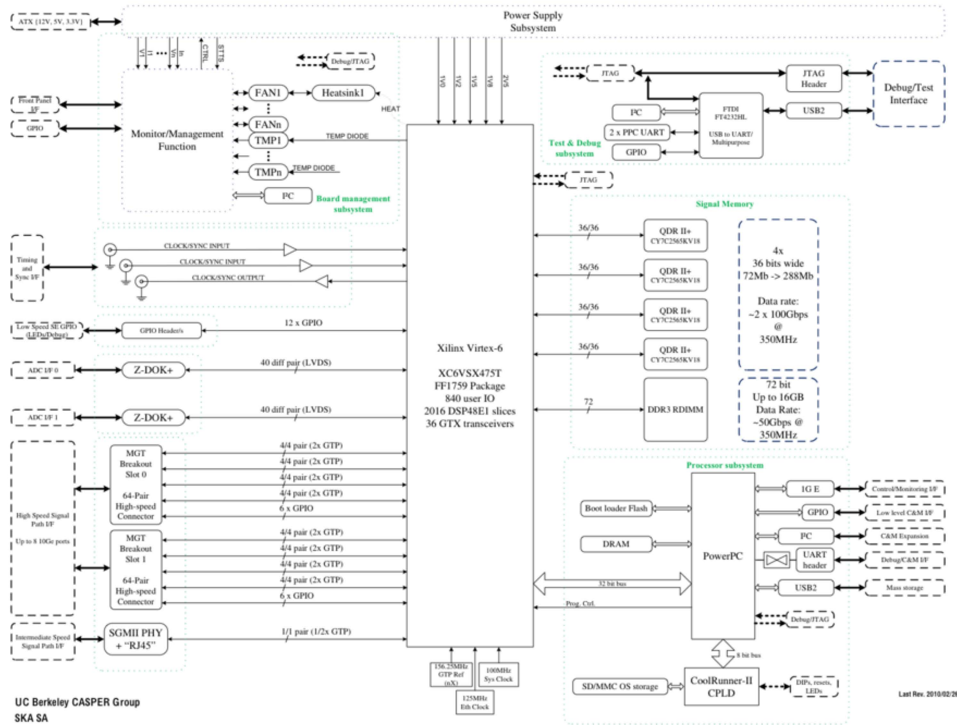
Appendix B

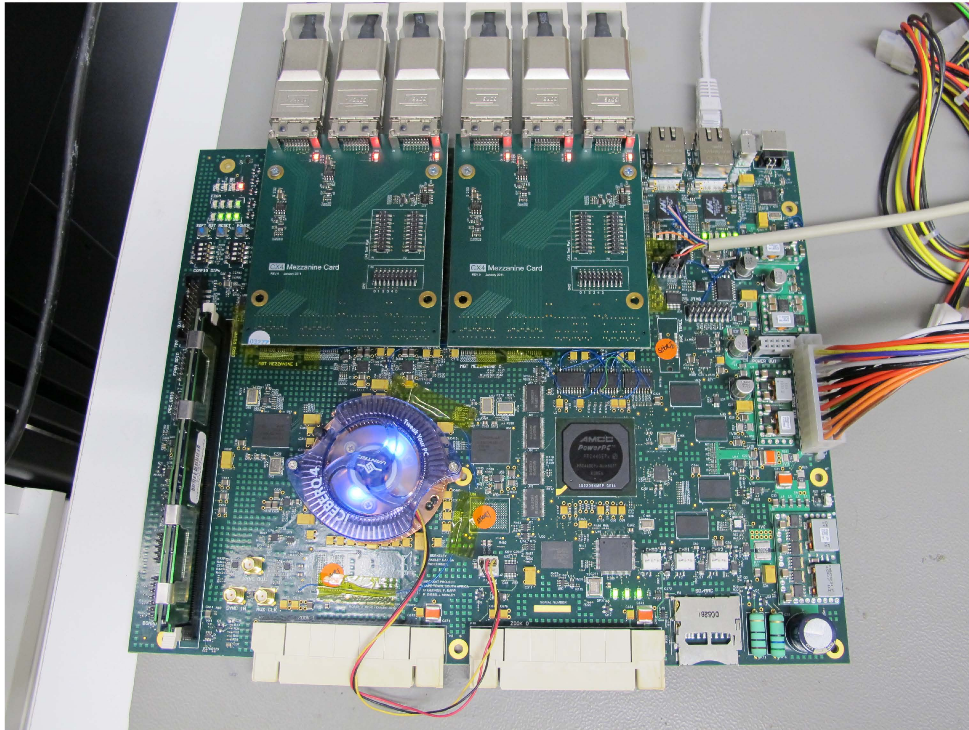
ROACH Board Connections

ROACH



ROACH2





Appendix C

DTC Compiler Usage

```
Usage:
        dtc [options] <input file>

Options:
        -h
                This help text
        -q
                Quiet: -q suppress warnings, -qq errors, -qqq all
        -I <input format>
                Input formats are:
                        dts - device tree source text
                        dtb - device tree blob
                        fs - /proc/device-tree style directory
        -o <output file>
        -O <output format>
                Output formats are:
                        dts - device tree source text
                        dtb - device tree blob
                        asm - assembler source
        -V <output version>
                Blob version to produce, defaults to 17 (relevant for dtb
                and asm output only)
        -d <output dependency file>
        -R <number>
                Make space for <number> reserve map entries (relevant for
                dtb and asm output only)
        -S <bytes>
                Make the blob at least <bytes> long (extra space)
        -p <bytes>
                Add padding to the blob of <bytes> long (extra space)
        -b <number>
                Set the physical boot cpu
        -f
                Force - try to produce output even if the input tree has errors
        -i
                Add a path to search for include files
        -s
                Sort nodes and properties before outputting (only useful for
                comparing trees)
        -v
                Print DTC version and exit
        -H <phandle format>
                phandle formats are:
                        legacy - "linux,phandle" properties only
                        epapr - "phandle" properties only
                        both - Both "linux,phandle" and "phandle" properties
```

Figure C.1: dtc compiler usage

Bibliography

[1] A Martens. KAT-7 F-Engine registers and their use. Technical Memo NRF-KAT7-5.0-MEM-024, SA-SKA, June 2011.

[2] Adelstein, Frank and Stillerman, Matt and Kozen, Dexter. Malicious Code Detection for Open Firmware. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 403–, Washington, DC, USA, 2002. IEEE Computer Society.

[3] AMCC. *PPC440 Processor Users Manual*, March 2008.

[4] Applied Micro. PowerPC 440EPx Embedded Processor, May 2012. Datasheet.

[5] B Hamilton. BORPH Correspondence. Personal Correspondence, 2009.

[6] B W Kerninghan, D M Ritchie. *The C Programming Language*. Prentice Hall Software Series, Second edition, 1988.

[7] C Brune. Introduction to Das U-Boot, the universal open source bootloader. http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Introduction-to-Das-UBoot-the-universal-open-source-bootloader/, August 2004.

[8] C Steiger and H Walder and M Platzner. Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-time Tasks. *IEEE Transactions on Computers*, 53:1393–1407, 2004.

[9] D G Bailey and K T Gribbon and C T Johnston and M Siripruchyanun. GATOS: A Windowing Operating System for FPGAs. In *Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA 2006), 17-19 January 2006, Kuala Lumpur, Malaysia*, pages 405–409. IEEE Computer Society, 2006.

[10] D George, S Malan, P Gibbs. Poster Abstract: ROACH-2 Development. In *CASPER Conference*, August 2011.

[11] E Gerald. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM.

University of Cape Town

[12] E Sabaki. Open Firmware - A New Class of Software Tool. http://www.openfirmware.org/1275/mejohnson/, 1996.

[13] Firmworks. Open Firmware Features. http://www.firmworks.com/www/features.htm, May 2009.

[14] G Hill. Expanding Your Market with Open Firmware. Firmworks Presentation.

[15] G Likely, J Boyer. A Symphony of flavours: Using the device tree to describe embedded hardware. In *Proceedings of the Linux Symposium*, volume 2, pages 27–37, July 2008.

[16] H Kwok-Hay So. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, University of California, Berkeley, 2007.

[17] H Kwok-Hay So and Brodersen, R. Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1 –6, aug. 2006.

[18] H Kwok-Hay So and R Brodersen. A unified hardware/software runtime environment for FPGA-based reconfigurable computer using BORPH. *ACM Transactions on Embedded Computing Systems*, 7, 2008.

[19] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoneix Technologies Ltd, Toshiba Corporation. Advanced Configuration and Power Interface Specification.

[20] IEEE. IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices, 1994.

[21] J Antlab. Overview of peripheral buses. Presentation, August 2011.

[22] J C MacInnis. Implementing Firmware For Embedded Intel Architecture Systems: OS-Directed Power Management (OSPM) through Advanced Configuration and Power Interface (ACPI). *Intel Technology Journal*, 2009.

[23] J Corbet, A Rubini, G K Hartman. *Linux Device Drivers*. O'Reilly, Third edition, 2005.

[24] Kwok-Hay So, H and Brodersen, R. Runtime Filesystem Support for Reconfigurable FPGA Hardware Processes in BORPH. In *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, pages 285 –286, april 2008.

[25] E. Lübbers and M. Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9(1):8:1–8:33, Oct. 2009.

University of Cape Town

[26] M Peattie. *Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or selectMAP Mode*. Xilinx, August 2009.

[27] M T Jones. Anatomy of the Linux kernel. Website, June 2007.

[28] M Welz. User Requirements. Personal Correspondence, March 2009. Draft.

[29] J. C. Merino, Pedro and López and M. F. Jacome. A Hardwar Operating System for Dynamic Reconfiguration of FPGAs. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, FPL '98, pages 431–435, London, UK, UK, 1998. Springer-Verlag.

[30] MSI. ACPI. Presentation.

[31] P L McMahon. Adventures in Radio Astronomy Instrumentation and Signal Processing. Master's thesis, University of Cape Town, 2008.

[32] S Rajan. Automated gateware discovery using Open Firmware. SKA Bursary Conference, University of Stellenbosch, December 2010.

[33] S Rajan. A simple memory mapped driver for FPGA access. CASPER Conference, Pune, India, October 2011.

[34] S Rajan. Automated gateware discovery using Open Firmware. SKA Bursary Conference, University of Stellenbosch, December 2011.

[35] S Rajan. Optimizing Processor-FPGA interface under Linux. CASPER Conference, Green Bank, USA, August 2012.

[36] So, H Kwok-Hay and Brodersen, R. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *ACM Trans. Embed. Comput. Syst.*, 7(2):14:1–14:28, Jan. 2008.

[37] So, H Kwok-Hay and Brodersen, R. File system access from reconfigurable FPGA hardware processes in BORPH. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 567 –570, sept. 2008.

[38] So, H Kwok-Hay and Brodersen, R. Runtime Filesystem Support for Reconfigurable FPGA Hardware Processes in BORPH. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 285–286, Washington, DC, USA, 2008. IEEE Computer Society.

[39] T Shanley, D Anderson. *PCI System Architecture*. Mindshare, Inc, fourth edition, 1999.

[40] tldp. PCI. http://tldp.org/LDP/tlk/dd/pci.html, May 2010.

University of Cape Town

[41] Wigley, G B and Kearney, D A and Warren, D. Introducing ReConfigME: An Operating System for Reconfigurable Computing. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, pages 687–697, London, UK, UK, 2002. Springer-Verlag.

[42] Xilinx. *OPB UART Lite (v1.00b).* Xilinx, October 2007.

[43] Xilinx. *Virtex-5 FPGA Configuration.* Xilinx, November 2011.

[44] Xilinx. *Virtex-6 FPGA Configuration.* Xilinx, November 2011.

University of Cape Town