ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

MaldOS:
a Moderately Abstracted Layer for Developing Operating Systems

Relatore:	**Presentata da:**
Chiar.mo Prof.	**Mattia Maldini**
Renzo Davoli	

Sessione III
Anno Accademico 2018-2019

Were it so easy

Abstract

The course of Operating Systems is arguably one of the most crucial part of a computer science course. While it is safe to say a small minority of students will ever face the challenge to develop software below the OS level, the understanding of its principles is paramount in the formation of a proper computer scientist. The theory behind Operating Systems is not a particularly complex topic. Ideas like process scheduling, execution levels and resource semaphores are intuitively grasped by students; yet mastering these notions through abstract study alone will prove tedious if not impossible. Devising a practical - albeit simplified - implementation of said notions can go a long way in helping students to really understand the underlying workflow of the processor as a whole in all its nuances.

Developing a proof-of-concept OS, however, is not as simple as creating software for an already existing one. The complexity of real-world hardware goes way beyond what students are required to learn, which makes hard to even find a proper machine architecture to run the project on.

This work is heavily inspired by μMPS2 (and μARM), a previous solution to this problem: an emulator for the MPIS R3000 processor. By working on a virtual and simplified version of the hardware many of the unnecessary tangles are stripped away while still mantaining the core concepts of OS development. Although inspired by a real architecture (MIPS), μMPS2 is still an abstract environment; this allows the students' work to be controlled and directed, but might leave some of them with a feeling of detachment from reality (as was the case for the author). What is argued in this thesis is that a similar project can be developed on real hardware without becoming too complicated. The designed architecture is ARMv8, more modern and widespread, in the form of the Raspberry Pi education board.

The result of this work is dual: on one side there was a thorough study on how to develop a basic OS on the Raspberry Pi 3, a knowledge that is as of now not properly documented for those unprepared on the topic; using this knowledge an hardware abstraction layer has been developed for initialization and usage of various hardware peripherals, allowing users to buid a toy OS

on top of it. While the final product can be used without knowing how it works internally (in a similar fashion to the μMPS2 emulator), all the code was written trying to remain as simple and clear as possible to encourage a deeper study as example.

Sommario

L'insegnamento di sistemi operativi è forse una delle componenti piú cruciali di un corso di studi in informatica. Pur sapendo che solo una piccola percentuale di studenti affronterà la sfida di sviluppare software al di sotto del livello del sistema operativo, la comprensione dei suoi principi di funzionamento è essenziale nella formazione di un informatico. In sé, la teoria dietro ai sistemi operativi non è particolarmente complessa. Concetti come scheduling, livelli di esecuzione e semafori di accesso alle risorse sono intuitivamente comprensibili per gli studenti; tuttavia appropriarsi pienamente di queste nozioni soltanto tramite lo studio teorico è quasi impossibile. Al contrario, progettarne un'implementazione pratica ma semplice può fare la differenza nell'assimilazione dei meccanismi di gestione di un processore, in tutti i loro dettagli.

Sviluppare un sistema operativo come progetto accademico, tuttavia, è diversi ordini di grandezza piú difficile rispetto alla creazione di software che si avvalga di un ambiente di lavoro giá esistente. La complessità aggiunta dell'hardware va spesso oltre a quello che ci si aspetta che gli studenti conoscano, il che rende difficile anche soltanto la ricerca di una architettura su cui eseguire suddetto progetto.

Questo studio è fortemente ispirato da precedenti soluzioni a questo problema, nella forma di μMPS2 (e μARM), un emulatore per il processore MIPS R3000. Lavorando su una virtualizzazione semplificata dell'hardware gli studenti si possono concentrare sui concetti chiave dello sviluppo di un SO senza essere rallentati da dettagli meno significativi. Anche se ispirato a un'architettura reale (MIPS), μMPS2 rimane comunque un ambiente astratto; pur permettendo di controllare e direzionare con precisione il lavoro, nel corso dello studio non si può eliminare del tutto la sensazione di distacco dalla realtà (come è stato per l'autore). In questo studio si sostiene che un progetto simile possa essere sviluppato su hardware reale senza che questo diventi troppo complicato. L'architettura scelta è ARMv8, più moderna e diffusa rispetto a MIPS, nella forma della board educativa Raspberry Pi.

Il risultato del lavoro è duplice: da una parte è stato portato avanti

uno studio dettagliato su come sviluppare un sistema operativo minimale sul Raspberry Pi 3, un processo che al momento non è adeguatamente documentato per chi non fosse già competente nel campo; forti di questa conoscenza è stato sviluppato un layer di astrazione per l'hardware che si occupa di inizializzare e semplificare l'approccio per alcune periferiche, permettendo agli utenti di costruirci sopra un sistema operativo giocattolo. Il prodotto finale può essere utilizzato senza sapere nulla sulla sua struttura interna (come l'emulatore μMPS2), e il codice interno è stato scritto con l'obbiettivo di essere il piú semplice e chiaro possibile per non scoraggiare uno studio approfondito in qualità di esempio.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
  - 1.1.1 μMPS and Similar Emulators . . . 2
  - 1.1.2 ARMv8 and Raspberry Pi . . . 3
  - 1.1.3 Kaya . . . 4
  - 1.1.4 Existing Work . . . 4
  - 1.1.5 Organization of This Document . . . 5

2 Discarded Options **7**
- 2.1 Raspberry Pi 2 (ARM32) . . . 7
- 2.2 Raspberry Pi Zero (ARM32) . . . 8
- 2.3 Raspberry Pi 3 (ARM64) . . . 9

3 Overview of the ARMv8 Architecture **11**
- 3.1 Exception Levels . . . 11
  - 3.1.1 Changing Exception Level . . . 12
- 3.2 Registers . . . 13
  - 3.2.1 General Purpose Registers . . . 13
  - 3.2.2 Special Registers . . . 14
  - 3.2.3 System Registers . . . 15
  - 3.2.4 PSTATE . . . 16
- 3.3 Exception Handling . . . 16
  - 3.3.1 Interrupts . . . 18
  - 3.3.2 Aborts . . . 19
  - 3.3.3 Reset . . . 19
  - 3.3.4 Exception Generating Instructions . . . 20
- 3.4 Caches . . . 20
- 3.5 Multiprocessor . . . 21
- 3.6 ARM Timer . . . 22

4 The Memory Management Unit 25
- 4.1 Address Translation . . . 26
  - 4.1.1 Granule Size . . . 26
- 4.2 Table Descriptor Format . . . 28
  - 4.2.1 Table Descriptors . . . 29
  - 4.2.2 Block Descriptors . . . 30
- 4.3 Memory Types and Attributes . . . 32
  - 4.3.1 Shareability . . . 32
  - 4.3.2 Cache policies . . . 33
  - 4.3.3 MAIR Configuration . . . 34
- 4.4 Kernel Space Virtualization . . . 35
  - 4.4.1 EL2 and EL3 Translation Process . . . 36
- 4.5 Translation Lookaside Buffer . . . 37
  - 4.5.1 Trivial Approach . . . 37
  - 4.5.2 ASID Approach . . . 37

5 Overview of the BCM2837 39
- 5.1 Boot Process . . . 39
  - 5.1.1 MicroSD Contents . . . 41
  - 5.1.2 Configuration . . . 41
- 5.2 Videocore IV . . . 42
  - 5.2.1 Mailboxes . . . 42
- 5.3 Peripherals . . . 45
  - 5.3.1 GPIO . . . 45
  - 5.3.2 External Mass Media Controller . . . 47
  - 5.3.3 UART Serial Interface . . . 49
- 5.4 Interrupt Controller . . . 52
  - 5.4.1 Inter Processor Interrupt (IPI) . . . 53

6 Emulated peripherals 55
- 6.1 Emulated Device Interface . . . 56
- 6.2 Printers . . . 59
- 6.3 Disks . . . 59
- 6.4 Tapes . . . 59

7 Project Internals 61
- 7.1 Design Principles and Overall Structure . . . 61
  - 7.1.1 Implementation Language . . . 62
  - 7.1.2 Build Tools . . . 62
  - 7.1.3 Linker Script . . . 62
- 7.2 Initialization . . . 63

7.3 Interrupt Management 63
7.4 Emulated Devices 65
    7.4.1 Tapes and Disks 65
    7.4.2 Printers 67
    7.4.3 Timer Queue 67
7.5 Hardware Library 68
7.6 Memory Management Unit 68
    7.6.1 Design Choices 70
7.7 Exception Levels and Virtualization 70

8 Student's Perspective **73**
8.1 ROM Functions 73
8.2 System Initialization 75
8.3 Exceptions 76
8.4 Multicore 76
8.5 Devices 77
    8.5.1 Real Peripherals 77
    8.5.2 Emulated Peripherals 77
8.6 Memory Management Unit 77

9 Usage and Debugging **81**
9.1 Final Result 81
9.2 Compiling 81
9.3 Qemu 83
    9.3.1 Create a Disk Image 85
9.4 Debugging 87
    9.4.1 Memory Management Unit 88

10 Conclusions and Future Work **91**
10.1 Extending Qemu 91
10.2 Debugging with GDB 92
10.3 Other ARM64 SoC 92
10.4 Other Programming Languages 93
10.5 Emulation Layer on Videocore IV 94
10.6 Course Organization 94

Bibliography **97**

List of Figures

3.1 ARMv8 Exception Levels 12
3.2 32 bit alias 14
3.3 special registers 15
3.4 Exceptions 19
3.5 Cache Levels 21

4.1 Address Translation Example 27
4.2 Address Translation Example 28
4.3 MAIR Register 34
4.4 Kernel memory virtualization 36

5.1 BCM2837 Boot Sequence 40
5.2 Raspberry Pi UART pins 49
5.3 Interrupt Controllers 52

6.1 Mailbox Structure 57

7.1 IRQ Switch 64
7.2 HAL Execution Thread 66

8.1 Memory Map 79

9.1 Gparted Partition Table 86
9.2 Gparted FAT32 Partition 87
9.3 Gdbgui 89

List of Tables

3.1 PSTATE fields 17
3.2 Exception Table 18

4.1 Page Entry Types 29
4.2 APTable 30
4.3 APBits 31

5.1 Mailbox Registers 43
5.2 Mailbox Message Format 44
5.3 Mailbox Tag Structure 45

6.1 Device Registers Layout 57
6.2 Emulated Interrupt Lines 58

Chapter 1

Introduction

1.1 Background

An Operating System is, in a nutshell, a very complex and sophisticated program that manages the resources of its host machine. Proper studying on the topic should yield higher understanding on many fields of the likes of parallel programming, concurrency, data structures, security and code management in general.

As mentioned in the abstract, an Operating Systems course should ideally include field work. This can be done through several different approaches, which have already been covered by previous works like μARM and μMPS2 [4] [5]. To quickly recap the most notable mentions:

Study of an existing OS: the most theoretical approach, it involves reading and analyzing the source code of an existing Operating System. There is no short supply of such examples; historically Minix is cited [1], but a quick research will reveal countless small kernels for embedded platforms and emulators.
The biggest downside of this approach is that the esamination of the source code may end up not having more educational value than a pseudocode snippet found in the textbook. The fact that the example is indeed practical is lost in the lack of application by the student.

Modification of an existing OS: this approach can be seen as a slight revision to the study-only policy. If the work under examination can indeed be run in some environment, students might find themselves modifying and testing small parts even if unprompted by the professor.

Construction from scratch: this is the idea behind projects μMPS, μMPS2, μARM and the KayaOS specification [3]. Creating an entire OS will be undoubtedly be the most formative experience as it leaves no room for lack of preparation when implementing the studied solutions. Its biggest downside is the risk of being too much for undergradates to handle, only hurting their self confidence in the process.

It is argued that the last approach is the most interesting and valuable for the students: if they are to study an existing Operating System at all, it is either the case that said OS would be too complex or simple enough for them to implement. In the first scenario the studying program must skip the most cumbersome parts and only cover what is essential, in which case the completeness of the example loses meaning. In the latter there is no reason not to follow the constructionist route and let the disciples create their own OS.

1.1.1 μMPS and Similar Emulators

Every learning project must find a balance between abstraction and concreteness. Developing a real world application with value outside of the academic context brings the most satisfaction to the scholar; frequently, however, an entirely practical assignment would lose a lot of learning value due to hindrances spanning outside of the course program.

In the frame of this work said hindrances would be the complexities tied to hardware architecture of peripherals and CPU that, although interesting in their own right, are unnecessary for the students' formation process. The μMPS emulator provides an environment fairly similar to real hardware while still being approchable for an undergraduate student; it positions itself in a sweet spot between abstraction and concreteness, allowing just enough of the underlying hardware to pass through and keeping the focus on theoretical topics like memory management, scheduling and concurrency.

After successfully concluding his or her work on μMPS the student has a firm grasp on said topics and has grown significantly in the ability to manage large and complex projects. There can be, however, a lingering confusion on the attained result, which is limited to a relatively small niche. The software itself may be compiled for a real architecture, but the final binary can only run on the simplified emulator, making it a trial for its own sake.

To be fair, the final end of μMPS is, in fact, learning, so this is not really a shortcoming. What is attemped with this work is to take a small step towards concreteness in the aforementioned balance without falling into a pit of unnecessary complexity. The occasion to do so is presented by the rise of a widespread and relatively clean architecture: ARMv8, specifically using the Raspberry Pi 3 educational board and the Qemu architecture emulator.

1.1.2 ARMv8 and Raspberry Pi

The passage from MIPSEL to ARM is not new to the μMPS family of emulators; the previous work of μARM was already pointed in this direction. μARM had the goal to modernize the μMPS experience and still maintained its emulator-only approach. In fact, when this work started the goal was to create an hardware abstraction layer to be able to run an μARM project on Raspberry Pi (which coincidentally has an ARMv7 core for the model 2).

In general, the MIPSEL architecture is losing more and more market ground as time passes. Two of its main features were being a 64-bits architecture and a faily simple one on top of that; the coming of ARMv8 superseded both aspects. Although it made history with successful devices like the original Playstation and Nintendo64 consoles, in the last decade MIPSEL processors found few practical applications (with the notable exception of the computer vision chip found on the Tesla Model S). Those considerations were valid when μARM vas conceived and obviously still stand.

Moreover, the ARMv8 architecture choice fixes most of the problems that previously arose while considering real hardware as an environment:

- **Widespread use**: the success of the ARM architecture in general makes it an interesting candidate for an undergraduate project; specifically, it is used by the whole Raspberry Pi family of educational boards (which needs no introduction), but also by a vast and growing majority of all mobile devices. Today, one can reasonably assume that an undergraduate student will know what a Raspberry Pi is at least by the end of his or her course of study.
- **Simplicity**: it will be argued over the dissertation that the 64-bits ARMv8 architecture is fairly simple compared to its predecessors, thus making it suitable even for a software-focused study.
- **Future prospect**: More and more devices are running on ARM. The smartphone market is almost entirely dominated by the family of processors, which is now expanding into notebooks and other handheld/ wearable/portable appliances. Having an experience - albeit small - in the field can prove useful for some students.

Being able to run on a real device is an added satisfaction but is mostly a nuisance during the development process, which is yet another problem that had been solved by μMPS2. Recently however an official patch has been added to the Qemu emulator that allows to emulate a Raspberry Pi 3 board and debug the running software with GDB. Working with Qemu and GDB brings, in the author's perspective, the important advantage of interacting with comprehensive and popular tools instead of a niche academic emulator, provided that said tools are sufficiently apt for the task.

1.1.3 Kaya

The end result is an hardware abstraction layer compiled for 64-bits ARMv8 architecture which provides initialization and a partially virtualized peripheral interface, to be linked with the student's own code. It was developed around the Kaya Operating System Project [3], with the main influence being the implementation of a virtual interface for emulated peripherals not present in any Raspberry Pi board: the HDMI connected display is split into four regions that act as printer devices, and the microSD card can contain several image files interpreted as disks and tapes. The presence of those emulated devices is important, as the Raspberry Pi boards are otherwise missing any pedagodically meaningful peripheral (the only exception being two UART serial interfaces).

1.1.4 Existing Work

Surprisingly, there is not much existing work on OS development for Raspberry Pi boards and the Broadcom *SoC* which the board builds on is shamefully undocumented. Obviously most existing Operating Systems for the board are licensed as open source, but their sheer dimension make them unsuitable for study. Therefore μMPS2, μARM, and the Kaya OS project were the only references taken for theoretical composition and precepts. Some of the few, closest available projects are:

- **BakingPi**: the only real academic effort in this direction. It is an online course offered by the University of Cambridge [2], but is more focused on assembly language and ARM programming than on real Operating Systems topics: it explains how to boot, receive input and present output on the Rapsberry Pi 1.
- **Ultibo**: Ultibo core is an embedded development environment for Raspberry Pi [8]. It is not an operating system but provides many of the same services as an OS, things like memory management, networking, filesystems and threading. It is very similar to the idea behind this work as an hardware abstraction layer that alleviates the burden of device management and initialization. Though not specifically created for OS development it might have been a useful reference if it was not written entirely in Free Pascal.

- **Circle**: Similar to Ultibo, but with a less professional approach and written in C++ [9]. In the same way it might be considered an already existing version of the presented work: however the initial approach for the user was judged too complicated and it was only used as a reference.
- **Raspberry Pi OS**: An online step-by-step tutorial on OS development inspired by the Linux kernel [7]. In many ways it is very similar to MaldOS, but has meaningful differences: it is incomplete and still ongoing, it does not contemplate an emulator and references mainly the Linux kernel, making it more complex to understand.

In particular, none of the existing work can be considered a complete and detailed guide on how to develop an Operating System for Raspberry Pi, a void that this project intends to fill.

In regard of the ARMv8 specification and AArch64 programming the main resource is the *"bare metal"* section of the official Raspberry Pi forums and the thriving production of examples created by its users. Even if the focus of that community is more shifted on embedded programming than Operating Systems development, their effort in hacking and reverse engineering the hardware proved an invaluable resource.

1.1.5 Organization of This Document

This chapter introduced the motives and the objective of this work. In the following chapters an overview of all the components involved is presented.

Chapter 2 briefly explains the thought process that went from the initial idea to the final realization, detaling the reasons behind the choice of the environment.

Chapter 3 describes the functioning principles of the ARMv8 specification and the Cortex-A53 implementing it. It is not meant to be an exhaustive reference (as it would be impossible to condense the whole ARM reference manual in this document), but it should clearly delineate the main foundations needed to understand this work.

Chapter 4 is an extension of chapter 3 focused on the Memory Management Unit; virtual address translation plays a huge part in developing Operating Systems, so it is believed it deserves a chapter of its own.

Chapter 5 gives an overview of the System-on-Chip the Raspberry Pi 3 is built upon, with attention to the peripheral devices reputed most useful from an educational perspective. Register sets for said devices are briefly listed for context.

Chapter 6 covers the "emulated peripherals"; those are the devices available in μMPS2 and μARM, absent in a real system such as the Raspberry

Pi. The hardware abstraction layer uses the existing mailbox interface and fast interrupts to seamlessly emulate said devices on top of other resources. For the end user, the illusion to use a real peripheral is almost perfect.

Chapter 7 describes the project from an internal point of view, including some topics already covered in previous sections with a more in-depth explaination of the functioning principles.

Chapter 8 dwells on the interface that is presented to the end user like ROM functions, special memory locations and mailboxes.

Chapter 9 mentions the base usage of this project. The actual product is nothing but a few precompiled elf binaries and a linker script, to be used when compiling to proof-of-concept OS, which can then be debugged step-by-step using GDB under any of its forms.

Finally, in chapter 10 a recap is made about the success of this project and directions for future works are listed.

Chapter 2

Discarded Options

Before settling for 64-bits ARMv8 on Raspberry Pi 3 several other options were probed; what follows is a recap and explaination on why they were discarded in favor of the latter. As mentioned before, the work began as an attempt to silently port kernels compiled for the μARM emulator to real hardware to provide students with a better sense of accomplishment.

2.1 Raspberry Pi 2 (ARM32)

The first *SoC* (System On Chip) to be experimented on was the Raspberry pi 2 (model B). The initial idea was to replicate as closely as possible the μARM experience, which runs on an emulated ARM7TDMI: although the RPi2 board uses a quad-core Cortex-A7 ARM it is still fairly similar, maintaining most of the registers and the 32-bits model.

As the first real approach to the problem, this was mainly a learning experience for the author. After understanding the basics of the system it became obvious that the differences between μARM and any Raspberry Pi board were too great to consider a simple porting of the projects meant for the emulator. This was evident especially for the emulated peripherals: like μUMPS2, μARM offers to the user 5 types of peripheral devices (network interface, terminal, printer, tape, disk) that find no immediate counterpart on the British family of boards.

This prompted to reconsider the objective of the work from a simple port to a different and autonomous educational trial. Thus, effort was bent into searching for a better way to develop OSes on a Raspberry Pi board while still using Kaya, μARM and μUMPS2 as reference.

With the new goal in mind there were two main issues with the Raspberry Pi 2:

1. **Ease of development**: if students are to develop software for a specific board it should be cheap and easily obtainable if not for them at least for the institution they study under. These characteristics are the signature of success for the Raspberry Pi foundation; still, version 2 is not the top product for either of those. Also, as will be described in more detail, running a custom kernel on a the Broadcom *SoC* requires copying the binary on a microSD card, inserting it and resetting the board. This, together with the lack of readily available debugging facilities lead to searching other options.
2. **Popularity**: the Raspberry Pi 2 was definitely superseded by the 3+ version in march 2018. It was assumed any work on it would have risked lack of support in the following years (assumption that was somehow confirmed with the new release, which follows the wake of version 3).

2.2 Raspberry Pi Zero (ARM32)

The model Zero was the second option to be considered for this work. It is significantly cheaper (with prices as low as 5$ for the *no-wireless* version) and compact. It runs on a single core ARM1176JZF-S, not too different from version 2 or the μARM emulated processor.
What made this model especially interesting was the ability to load the kernel image in memory through an USB connection, without using a microSD card altogether. The board has USB On-The-Go capabilities, allowing it to appear as a device if connected to an host; at that point it's possible to load the kernel using the official *rpiboot* utility.

In an ideal scenario, the user would compile his or her OS, connect the board via USB to the host PC, load it with *rpiboot* and then interact with a serial output from the same USB connection. Unfortunately the last step would have required a massive amount of work to write a bare metal OTG USB driver and have the Raspberry Pi Zero appear as a serial console. Without it the only way to receive actual output was to have a second USB to serial converter connected to the GPIOs, which is no less cumbersome than with other models.

This, along with the lack of usable debugging tools, led yet again to look for a better option.

2.3 Raspberry Pi 3 (ARM64)

The final choice was the Raspberry Pi 3 (any model, in theory). Although it shares some of the shortcomings of previously considered alternatives like lack of peripherals and a difficult development cycle, it offered a significant advantage: the availability of an emulator in the form of Qemu [1]. The support of the `raspi3` machine on Qemu came only recently (version 2.12.1, August 2018) and the opportunity was seized immediately. Qemu support means kernels meant for the board can be more easily tested on the emulator and debugged with GDB. This allows to keep the advantages of a virtual environment like in μMPS2 and μARM while at the same time taking a step further towards practical usage when the kernel is run seamlessly on real hardware too.

Qemu has some limitations that can be overlooked. It supports only some of the hardware peripherals of the Raspberry Pi 3, with notable exclusions being the System Timer and the USB controller (that manages Network peripherals as well). Of those two limitations only the USB controller cannot be overcome, as the System Timer can be replaced by the internal ARM Timer. USB and Network Interfaces are missing from this project.

Lastly, with Raspberry Pi 3 came also the change to the architecture, from 32 to 64-bits. The Cortex A-53 running on the board follows the ARMv8 specification, which adds 64-bits support while still keeping backward compatibility for 32-bits applications. In theory, the student developed kernel could still use a 32-bits architecture; however, after studying thorughly the new AArch64 it was decided to switch to it. The main reasons for this decision are two: first, the Kaya OS project (and other similar projects as well) does not have any particular reference to the width of a word on the host architecture; provided they have to manage different registers, the underlying architecture is transparent to students. Second, it is the author's belief that the new ARMv8 specification for AArch64 is significantly simpler than its predecessors. As an example, it has only four execution levels (out of which two are used in this work), opposed to the nine execution-state division of ARMv7.

---

[1] Qemu has since supported Raspberry Pi 2 as well, but by the time the author realized it version 3 was already the designated board. It still retains many advantages over 2.

Chapter 3

Overview of the ARMv8 Architecture

What follows is a description of the ARMv8 architecture at a detail level deemed sufficient to understand the entirety of this project. The main references are of course the Programmer's Guide [13] and the Arm Reference Manual [14].

ARMv8 is the latest generation of ARM architectures, following ARMv7. It brings an enourmous list of changes from its predecessors, finally adding a 64-bits option to the family; still, it does so while still keeping backward compatibility towards 32-bits code and applications. The execution state in which an ARMv8 processor runs 64-bits code is called *AArch64*, while *AArch32* identifies the compatibility state for 32-bits applications. The AArch32 is very similar to the previous ARMv7 specification; in fact, when the scope of this project was still moving from the Raspberry Pi 2 (ARMv7) to the Raspberry Pi 3 (ARMv8) the source code and toolchain used were initially unchanged. Being the first attempt for the ARM consortium at 64-bits machines it takes advantage of a fresh start, removing many elements of complexity found in past entries while copying positive qualities from competitors that came before them (like 64-bits MIPS).

3.1 Exception Levels

When executing in AArch32 state the registers and system configuration is almost identical to ARMv7, separated in no less than 9 encoded processor modes with one of 3 possible privilege level. AArch64 significantly simplifies this model with just 4 exception levels, ranging from EL0 to EL3. Compatibility is achieved with non-injective, surjective mapping from processor

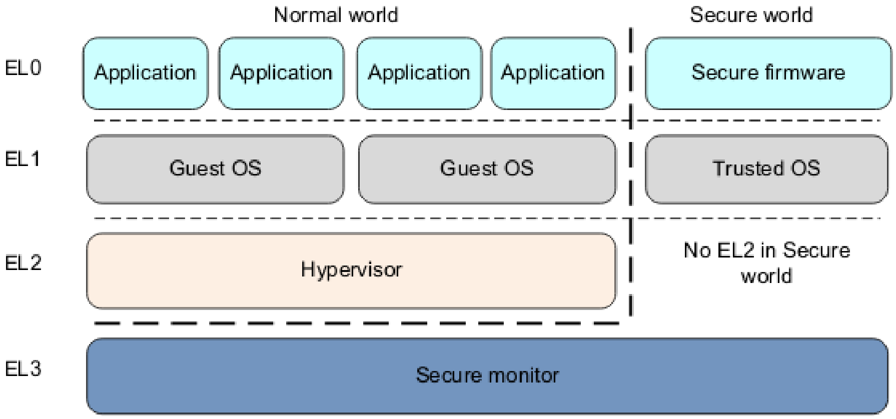

Normal world
Secure world
EL0
Application
Application
Application
Application
Secure firmware
EL1
Guest OS
Guest OS
Trusted OS
EL2
Hypervisor
No EL2 in Secure world
EL3
Secure monitor

Figure 3.1: ARMv8 Exception levels and their main purpose

modes to exception levels.

We describe briefly the function of each exception level:

- **EL0** is the lowest exception level, often referred to as “unprivileged” in opposition to every other, “privileged”, level. It has severe limitation in accessing system registers and failure to respect them is met with a synchronous abort exception. It is meant to run user applications, processes below the kernel.
- **EL1** is the first privileged level. It is where most interrupts end up and is meant for the OS kernel.
- **EL2** is the Hypervisor level; here resides hardware support for virtualization, a level meant to supervise virtual machines. For example, KVM is an in-kernel virtualization running at level **EL2** and supervising the virtual kernel at **EL1**.
- **EL3** is used to separate the system into secure partitions with the hardware TrustZone support.

3.1.1 Changing Exception Level

A change in the current exception level can be either caused by a willing decision of a higher privilege **EL** to a lower privilege **EL** or following an exception. Moreover, an exception cannot be taken to a lower exception level (e.g. if the core is currently at **EL2** and an interrupt line that should be handled at **EL1** is asserted it will be ignored as long as the exception level is not lowered, regardless of interrupt enabling). To access a lower exception level an `eret` instruction is required: `eret` loads the state stored in **SPSR_ELn** (see 3.2.2), where **ELn** is the current exception level, as the new system status (exception level included). Since no exception is ever handled at **EL0**, **EL0** is only reachable through `eret` instructions.

Exceptions are normally taken to **EL1** but can be set to run in **EL2** or even **EL3** by configuring corresponding system registers **HCR_EL2** and **SCR_EL3**, called *Hypervisor Configuration Register* and *Secure Configuration* Register respectively.

It is also possible to change execution *state* (i.e. AArch64 or AArch32) during runtime but that is irrelevant for the scope of this work, as it lies entirely in AArch64.

3.2 Registers

3.2.1 General Purpose Registers

One of the immediate benefits of a 64-bits architecture is a larger register pool: ARMv8 uses 31 64-bits wide general purpose registers, more than doubling from ARMv7. The registers are numbered from **x0** to **x30**. Although they are freely accessible the developer should be mindful of their secondary purpose for function calling convention (both C and Assembler):

- **x0** to **x7** are used to hold both arguments and return value (only **x0**) of a C function.
- **x8** is used to pass an indirect result value (e.g. a returned structure, in which case x8 holds the address to a properly set memory location).
- **x9** to **x18** are used to hold local variables in a routine call. They are caller-saved, which means that it is the caller responsibility to preserve their content before issuing a C function call.
- **x19** to **x28** are similar temporary registers, but for the callee to restore before returning; they are referred as callee-saved.
- **x29** is the frame pointer.
- **x30** is the link register.

Every general purpose register also has a 32 bit alias obtained replacing “x” with “w” in the register’s name (from **w0** to **w30**) that permits access to the lower (i.e. least significant) 32 bits of the register; the upper 32 bits are ignored.

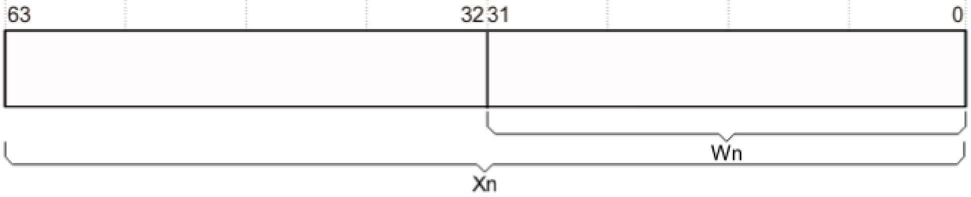


Figure 3.2: 64 bit register with “x” and “w” access

3.2.2 Special Registers

There are 5 special registers:

Zero Register: **xzr** and **wzr** provide access (as 64 and 32 bits register respectively) to a special register that ignores write attempts and always read as zero.

Program Counter (pc): up until ARMv7 the program counter was a general purpose register held in **r15**. In ARMv8 it has a very limited access, being read only and only implicitly used in certain instructions. This is one of the biggest differences with previous architecture and caused a lot of initial confusion; its restrictiveness results nonetheless in a much clearer and less error prone program flow.

Exception Link Register (elr): without free access to the program counter the system must provide an alternative way to restore a process' execution point. The exception link register holds the exception return address: it is automatically filled when one is fired and can be overwritten. Upon executing an `eret` instruction the value in **elr** is set as the program counter.

Saved Process Status Register (spsr): similarly to **elr**, this register is automatically initialized with various status informations upon taking an exception, and is restored (after eventual modification) with an `eret` instruction.

Stack Pointer (sp): The current stack pointer. It is freely accessible both in read and write operations.

As depicted in figure 3.3 some special registers have different versions for different exception levels: there is a separated stack pointer for all four of them and **EL0** is the only level missing **spsr** and **elr** (owing to the fact that they are exception related registers, and **EL0** never deals with exceptions or `eret` instructions).

Access to a special register from a different exception level is permitted if said register belongs to a lower level: for example **EL3** can set all the other

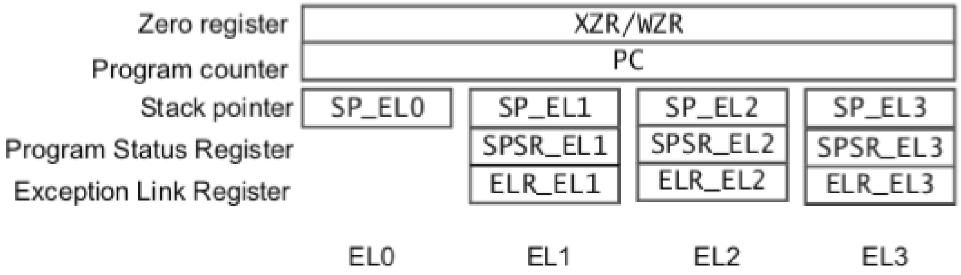

Zero register
Program counter
Stack pointer
Program Status Register
Exception Link Register
XZR/WZR
PC
SP_EL0
SP_EL1
SP_EL2
SP_EL3
SPSR_EL1
SPSR_EL2
SPSR_EL3
ELR_EL1
ELR_EL2
ELR_EL3
EL0
EL1
EL2
EL3

Figure 3.3: AArch64 Special Registers

stack pointers (including its own), but **EL1** trying to do the same will trigger an abort for **sp_el2** and **sp_el3**.

3.2.3 System Registers

Another significant turn from ARMv7 is the absence of a coprocessor interface. A coprocessor is an auxiliary core used to supplement the functions of the primary processor; ARMv7 specified a generic coprocessor interface to connect up to 15 assisting cores, one of which was reserved for system registers management. While coprocessors had to be controlled via specific instructions ARMv8 system registers are directly accessed in Assembler with the `mrs` and `msr` instructions as per any other register. This is a welcome change that simplifies the developer's approach to system configuration.

Similarly to special registers many system registers have different, banked versions for some or all exception levels (usually not **EL0**), each with the suffix *_ELn* to indicate the corresponding level. These registers are usually 32 bits wide. What follows is a list of system registers considered most important for the purpose of this work; for a detailed description of the various bit fields refer to the ARM reference manual [14].

Exception Syndrome Register: ESR_EL*n*, for each exception level holds the information regarding the last occurred exception (only for synchronous and SError, not for IRQs and FIQs. See 3.3 for more on exceptions). It is necessary to distinguish between exception classes and to find details specific to the cause of disruption in normal program flow.

Fault Address Register: FAR_EL*n*, it is used in pair with **ESR_EL*n*** to find which address caused a Data or Instruction synchronous abort.

Hypervisor Configuration Register: HCR_EL2, controls virtualization settings and trapping of exceptions to **EL2**.

Memory Attribute Indirection Register: MAIR_EL*n*, stores the user-provided memory attribute encodings corresponding to the possible values in a MMU translation table entry for translations at level *n*.

Multiprocessor Affinity Register: MPIDR_EL1 is the executing core id, used mainly to distinguish on which core the code is running on.

Secure Configuration Register: SCR_EL3 controls Secure state and trapping of exceptions to EL3.

System Control Register: SCTLR_EL*n* controls architectural features, for example the MMU, caches and alignment checking.

Translation Table Base Register 0: TTBR0_EL*n*, holds the address to the MMU translation table used normally at each exception level.

Translation Table Base Register 1: TTBR1_EL1, holds the address to the a special translation table used to separate application and kernel space. See section 4 for more.

Vector Based Address Register: VBAR_EL*n* is a pointer to the exception vector table for level *n*.

3.2.4 PSTATE

A reader with experience in ARM architecture will surely notice the lack of a current program status register holding informations like the current exception level, aritmetic flags, interrupt mask and so on. The AArch64 version of said register is implicitly present and not directly accessible. Instead, the single fields are supplied to read and write independently; this collection of “fake registers” is globally called **PSTATE**; the single fields and their meaning are listed in table 3.2.4. Curiously, querying for the **CPSR** register in a GDB debugger will correctly display the **PSTATE** components as a whole, although no such register can be loaded from or stored to even with Assembler instructions.

3.3 Exception Handling

In ARM architecture exceptions are conditions or system events that require some action by privileged software to ensure smooth functioning of the system; said condition is taken care of immediately by interrupting the normal flow of software execution and starting another routine (the exception handler). There are several classes of exceptions; every class can branch

Field name	Register handle	Description
N	None	Negative condition flag
Z	None	Zero condition flag
C	None	Carry condition flag
V	None	Overflow condition flag
D	daifset and daifclr	Debug mask bit
A	daifset and daifclr	SError mask bit
I	daifset and daifclr	Interrupt mask bit
F	daifset and daifclr	Fast interrupt mask bit
SS	None	Software Step bit
EL	CurrentEl	Current exception level
nRW	None	Current execution state (AArch32 or AArch64)
SP	None	Stack pointer selector

Table 3.1: PSTATE fields definitions

in different kinds, and every exception can be either synchronous or asynchronous (see figure 3.4).

The code to run when an exception is fired is specified by the developer in an exception vector table. The pointer to the exception vector table is written to the **VBAR_EL*n*** register, with n ranging from level 1 to 3, so every exception level has its own table (nothing prevents multiple levels to point to the same table however). For exceptions fired while at **EL0** the table for **EL1** is referenced.

The exception table can be anywhere in memory but must be 128 bytes aligned and must have the format specified in table 3.2. Each entry in the table is 16 instructions long, allowing for some control logic to be present in the top level handler as well before branching to a more complex routine. The table can be divided in four sections:

1. handlers to be used when the exception does not change neither the current exception level nor the stack pointer.
2. handlers to be used when the exception does not change the current exception level but should use a specific stack pointer.
3. handlers to be used when the exception elevates the privilege level and the execution state is in AArch64.
4. handlers to be used when the exception elevates the privilege level and the execution state is in AArch32.

Address	Exception type	Context
VBAR_ELn + 0x00	Synchronous	Current EL with SP0
VBAR_ELn + 0x80	IRQ/vIRQ	
VBAR_ELn + 0x100	FIQ/vFIQ	
VBAR_ELn + 0x180	SError/vSError	
VBAR_ELn + 0x200	Synchronous	Current EL with SPx
VBAR_ELn + 0x280	IRQ/vIRQ	
VBAR_ELn + 0x300	FIQ/vFIQ	
VBAR_ELn + 0x380	SError/vSError	
VBAR_ELn + 0x400	Synchronous	Lower EL using AArch64
VBAR_ELn + 0x480	IRQ/vIRQ	
VBAR_ELn + 0x500	FIQ/vFIQ	
VBAR_ELn + 0x580	SError/vSError	
VBAR_ELn + 0x600	Synchronous	Lower EL using AArch32
VBAR_ELn + 0x680	IRQ/vIRQ	
VBAR_ELn + 0x700	FIQ/vFIQ	
VBAR_ELn + 0x780	SError/vSError	

Table 3.2: Exception table format.

Each section has four different handlers for synchronous exceptions, IRQ, FIQ and SError. The Stack Pointer to be loaded is chosen by the stack pointer selector field in the current state register (see table 3.2.4); it can either be the current stack pointer or a memory address banked for the destination level.

3.3.1 Interrupts

Interrupts can be fast interrupts (FIQ) or normal interrupts (IRQ). Aside from the fact that FIQ have higher priority these two types of exception are virtually identical, and fast interrupts are considered a vestige from past architectures.
Usually it is the developer's responsibility to route an interrupt source to IRQ or FIQ. Interrupts are tipically associated with external hardware and connected via input pins to the processor; The connection can be direct or, more commonly, pass through an external device called interrupt controller that elaborates interrupt priorities and organization (see section 5.4).

Because the occurrence of interrupts is not directly related to the instruction cycle being executed by the core at any given time, they are classified as asynchronous exceptions.

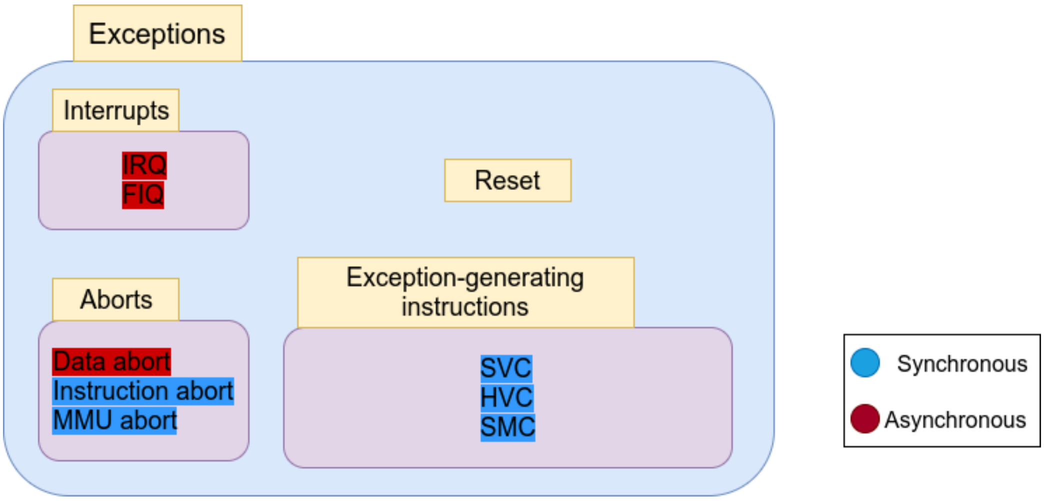

Exceptions
Interrupts
IRQ
FIQ
Reset
Aborts
Data abort
Instruction abort
MMU abort
Exception-generating instructions
SVC
HVC
SMC
Synchronous
Asynchronous

Figure 3.4: Tree of exception classes.

3.3.2 Aborts

Abort exceptions, also called system errors (SError), occur every time an abnormal condition is met during a memory access. Instruction Aborts result from an error during an instruction fetch cycle, while Data Aborts follow failed data access.

Despite the names depicting error conditions, aborts can work in perfectly normal and predictable flows. This is the case of MMU faults, generated by the Memory Management Unit on occasions like access to dirty page entries. The severity of conditions that set off abort exceptions can be configured to some extent with system registers; for example, a TLB miss can be ignored or fire an exception, and memory accesses can pass through address alignment and permission checks which may or may not interrupt the process.

Aborts can be both synchronous and asynchronous: MMU faults and alignment induced aborts are always synchronous, while data aborts can be asynchronous in certain situations.

3.3.3 Reset

Reset is a special exception, fired on power up of the processor. Its handler is implementation-specific and presumably located at address `0x80000`, the starting point for the loaded kernel, in the case of BCM2837.

3.3.4 Exception Generating Instructions

We have seen that a core can lower its exception level with `eret`, but can only increase it through an exception. For this purpose there are Assembler instructions that induce an exception to a higher execution level, usually to require a service paired with a higher privilege. The most obvious example of this behaviour are system calls.

- **SVC:** the *supervisor call* instruction fires an exception handled at **EL1**. Used by user programs to require kernel services.
- **HVC:** the *hypervisor call* instruction fires an exception handled at **EL2**. Used by the guest OS to require hypervisor services.
- **SMC:** the *secure monitor call* instruction fires an exception handled at **EL3**. Used to require *secure world* context switch.

Since those exceptions follow an instruction execution they are by definition synchronous.

3.4 Caches

A cache is a block of memory with faster access than the memory normally used. Whenever the RAM is read the resulting value can be stored in this intermediate, efficient ephimeral storage unit; future references can then be extracted from the cache instead (provided the original memory location was not modified) for a significant increase in performance. Since slower is normally cheaper, adding a small amount of high quality memory can speed up the whole system without having to use an expensive replacement.

The ARMv8 specification defines three different cache levels; the processor can then implement an arbitrary number of them. The cache levels are distinguished based on their position on the route from the single core to the external memory.

In figure 3.5 we can see the disposition of the levels. The distance from the core issuing a memory access is also an important factor in performance. When a core requires a memory location, the address is first searched for in the level 1 cache, which is core specific and not shared. If nothing is found the lookup continues hierarchically to the level 2 cache, which is shared among cores but internal to the processor. Another miss results in the third cache level, which is eventually external to the processor.

A specialized device, the cache controller, ensures this search/update mechanism works smoothly. Care must be taken to avoid problematic coherency situations, where the content of caches and main memory differ.

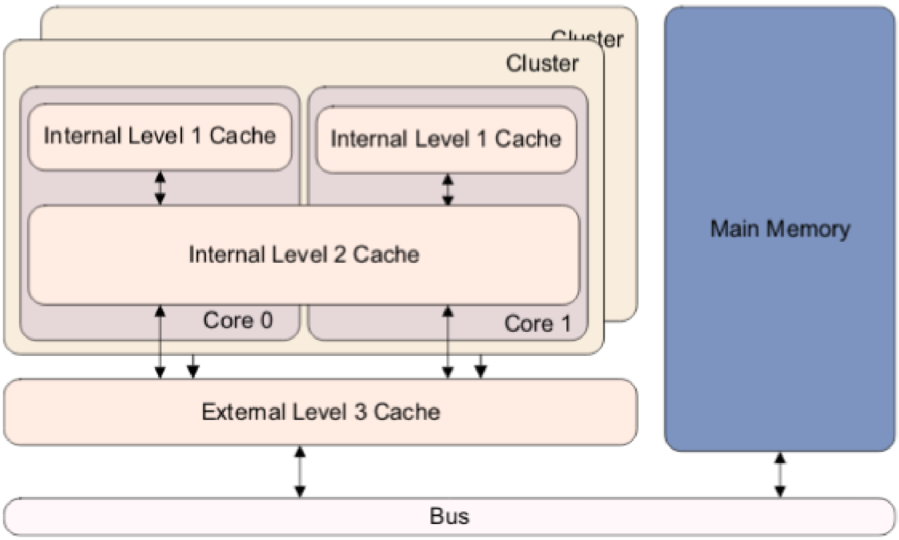


Figure 3.5: Position of different cache levels in the system.

Since the objective of this work is to build an educational experience performance is not a relevant factor; therefore, caches are simply disabled altogether. Yet a generic understanding of caching is required to correctly setup memory attributes when configuring the MMU.

3.5 Multiprocessor

Although not mandatory, the ARMv8 architecture is specifically structured to host systems with more than one core, like the Cortex-A53 CPU. In a multicore environment each core has both personal and shared resources: the registers (general purpose and system) are tipically duplicated and have a localized effect, while the main memory is shared by all cores at all times. As seen in the previous section the same applies to caches, with memory banks closer to the core itself being personal while others are shared.

Each core has its own execution thread, identical or different from the others. The A-53 processor has a single clock input, so every core runs at the same frequency.

At a shallow level, the only register one should be concerned with when dealing with parallelism is **MPIDR_EL*n***. It contains the identifying number for both cluster and core that is currently executing, so reading it yields different results depending of who is doing it. On the A-53 there is only one cluster and four cores (indexed from 0 to 3).

When the reset interrupt is fired execution starts only for the first core; the remaning three are held in a waiting state (as if by executing a `WFE` instruction). They can be woken up by executing a `SEV` (send event) instruction, at which point execution starts at address `0x80000` for everyone else. The most immediate way to split execution into different threads is to check **MPIDR** and follow up with a conditional branch. From there each core should be parked in a spinlock, waiting to be released and directed towards the code it should execute.

3.6 ARM Timer

The Cortex A-53 implements a generic ARM timer interface that can be used for interrupts and time scheduling. Despite its apparently simple purpose this is a very convoluted internal device used to virtualize timers for guest OSes as well. Given such functions are not needed in this work, the explaination will focus on the physical counter and interrupt setting registers.

First of all, the timer must be initialized. It has a fixed running frequency of 62.5 MHz that increments a 64-bits counter on each tick, but by default the timer registers cannot be accessed at lower exception levels. Setting the proper bits of **CNTHCTL_EL2** and **CNTKCTL_EL1** signal clearance of access from **EL2** and **EL1** to lower levels, respectively.

From there, every exception level (and each core) but **EL0** have its own timer to configure and use freely. Additionally, **EL1** and **EL2** can benefit from a virtual timer with a counter value equal to the physical one minus a specified offset (register **CNTVOFF**), for a total of five channels. Each timer is controlled through four classes of registers:

- **Control:** 32-bits registers named **CNT*t*_CTL_EL*n***, where t represents the kind of timer (e.g. P for physical, V for virtual, H for hypervisor) and n is the exception level. They only use 3 bits to determine whether the timer is enabled, if the interrupt should be masked and if the condition has been met (i.e. the time alarm has been reached).
- **Counter:** the ever growing 64-bits counter register, named **CNT*t*CT_EL*n***.
- **Compare Value: CNT*t*_CVAL_EL*n***, holds the compare value of the corresponding timer. The condition for the timer is met when $Counter - CompareValue >= 0$.
- **Timer Value: CNT*t*_TVAL_EL*n***, is a convenience 32-bits register

for setting the next timer interrupt. On write, the compare value is set to the current timer plus the written timer value.

The memory mapped generic interrupt controller can be used to decide which kind of interrupt (IRQ or FIQ, if any) is fired on condition met. Then, an appropriate value can be loaded onto the timer value register, and eventually an exception will be fired. Timer interrupt lines are de-asserted either by setting a new timer in the future or by disabling the counter altogether. Even at the same exception level, physical and virtual timers have separate interrupt lines. In this project both are used at **EL1** by the abstraction layer: the physical line is routed to normal interrupts while the virtual one is managed by a FIQ.

Chapter 4

The Memory Management Unit

The Memory Management Unit is a device found in most CPUs tasked with the objective of translating from virtual memory addressing to physical addressing. The Cortex-A53 is no exception and has an advanced internal MMU. It is such an important component of the system and of any Operating System that even if it's technically part of the ARMv8 specification it deserves a chapter on its own.

The main purpose of address translation is to allow each process to have its own virtual address space that has nothing to do with how much memory is available (and where this memory is located) on the system, with hardware MMIO and other processes hidden from its view. If the memory management unit is active any address referenced by a process is first elaborated and translated: different sections of the 64-bits address (starting from the most significant) are used to index different levels of a tree containing translation entries. While translating there can also be additional checks on whether the current exception level has the proper permissions to access the resulting physical memory. For example memory blocks where the kernel is loaded should not be accessed in any way by **EL0** code; in a similar way memory that is writable at **EL0** cannot be allowed to execute at higher permission levels. The latter constriction is built in the specification by default; the former should instead be enforced by the developer via MMU configuration. Only after translation and permission checks are done the actual memory access is performed; however, most of the process is handled by the hardware and can be almost entirely ignored by the programmer after the initial setup.

4.1 Address Translation

As briefly mentioned in the introduction, when the MMU is active every address is treated as an array of indexes for the translation table. The translation table is a tree of translation entries that maps a section of RAM: each level of the tree has entries covering a certain number of elements in the next level until the last, that corresponds directly to memory.

The example that follows considers a tree where each entry before last level points to 512 more entries and depicts a trivial "identity" mapping: every virtual address is translated in the same physical address. The bottom level covers a 4KiB block of memory directly (the reason for these numbers will be explained in section 4.1.1). There are four levels in the tree, level 0 to 3. Presume we want to translate a 64-bits address:

- the 16 most significant bits are reserved for kernel space virtualization (more on this topic in section 4.4).
- bits 47:39 are the level 0 table and reference a level 1 entry. Each level 0 entry spans a 512 GiB memory range (2^39).
- bits 38:30 are the level 1 table and reference a level 2 entry. Each level 1 entry spans a 1GiB memory range (2^30 or $512GiB/512$).
- bits 29:21 are the level 2 table and reference a level 3 entry. Each level 2 entry spans 2 MiB memory range, similarly to previous levels.
- bits 20:12 are the level 3 table and reference the last level, made of direct memory blocks. Each memory block is 4KiB wide.
- bits 12:0 are the offset for the last memory block and index the actual word referenced.

In this mundane example it is evident how the translation process is arbitrary; every level can simply be cut off and the resulting address be obtained by adding the intermediate indexes and the remaining bits (to be considered as the final offset).
This is not possible anymore if the translation function is not an identity, in which case the translation function is codified by the pointers in the table entries; the table entry marks the beginning of the next level of tables and the corresponding piece of virtual address indexes the chosen entry.

4.1.1 Granule Size

With granule size we refer to the smallest possible block of memory that can be indexed by the MMU tables; in the previous example, 4KiB. The

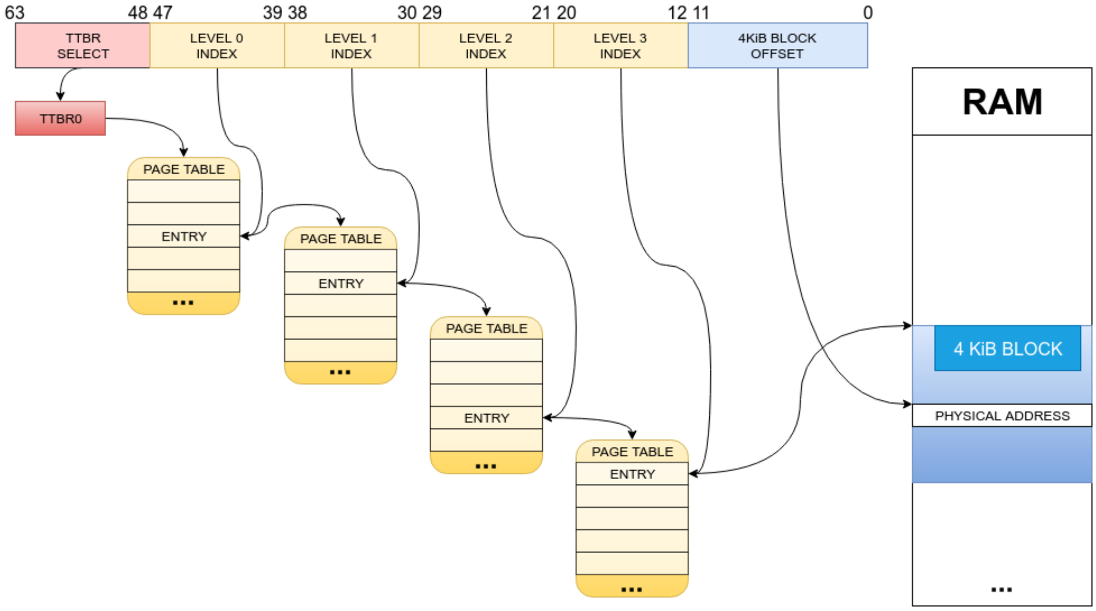

63
48 47
39 38
30 29
21 20
12 11
0
TTBR SELECT
LEVEL 0 INDEX
LEVEL 1 INDEX
LEVEL 2 INDEX
LEVEL 3 INDEX
4KiB BLOCK OFFSET
TTBR0
PAGE TABLE
ENTRY
...
PAGE TABLE
ENTRY
...
PAGE TABLE
ENTRY
...
PAGE TABLE
ENTRY
...
RAM
4 KiB BLOCK
PHYSICAL ADDRESS
...

Figure 4.1: Example of the address translation process .

ARMv8 specification allows for three different granule sizes: 4KiB, 16KiB and 64KiB. The actual ARM processor abiding to the standard can in turn implement those granules only partially, and fortunately the Cortex-A53 implements them all. The granule size is a global setting, affecting the entirety of the page table. Different granule size dictate how many levels is possible to have and how many entries are in each table (for example, a granule of 64 KiB allow only 3 levels and a granule of 16KiB will result in 10-bits wide address sections).

A small granule size will result in more control but also in a bigger table; to divide a 2GiB RAM into 4KiB blocks an Operating System will need 524288 64-bits wide table entries, for a total of 4MiB of allocated memory. Choosing a fine grained control does not mean commiting to it, however. If there are large sections of memory with the same memory attributes and virtual addressing (e.g. MMIO memory that should generally not be accessed by user processes) the developer can “cut early” the page table tree and use any intermediate memory range. In the above example we could arbitrarily stop at level 2 and create a part of the table with direct entries spanning 2 MiB each, situation photographed in figure 4.2.

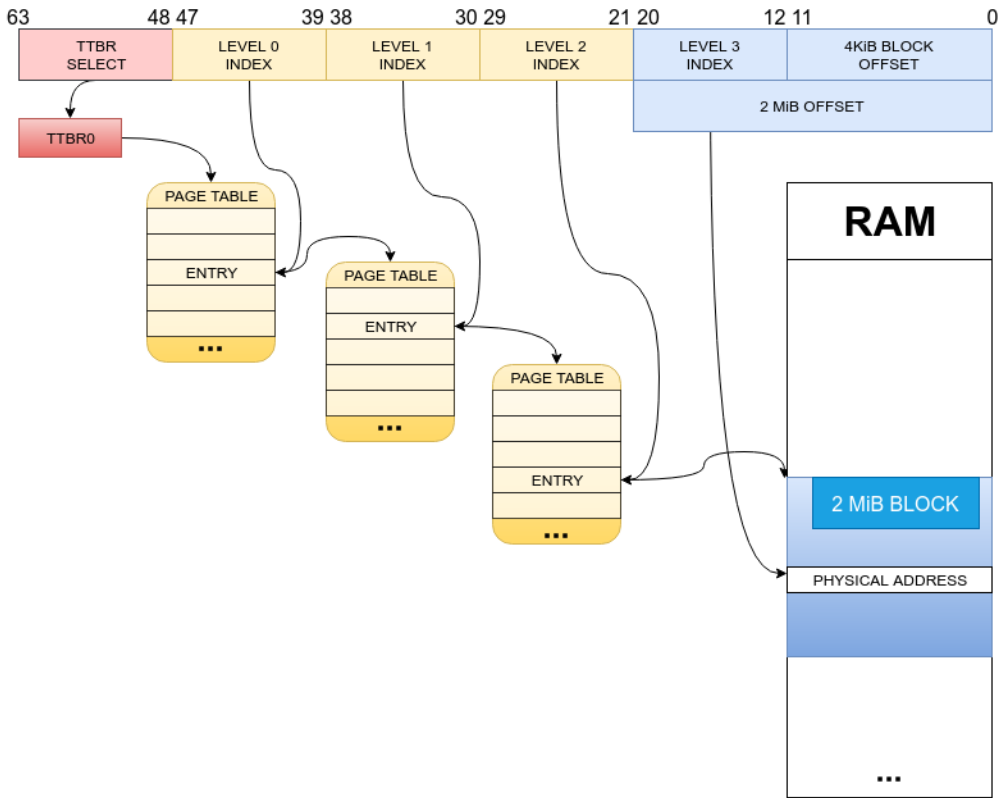

63
48 47
39 38
30 29
21 20
12 11
0
TTBR SELECT
LEVEL 0 INDEX
LEVEL 1 INDEX
LEVEL 2 INDEX
LEVEL 3 INDEX
4KiB BLOCK OFFSET
2 MiB OFFSET
TTBR0
PAGE TABLE
ENTRY
...
PAGE TABLE
ENTRY
...
PAGE TABLE
ENTRY
...
RAM
2 MiB BLOCK
PHYSICAL ADDRESS
...

Figure 4.2: Here the last 21 bits are used as a bigger offset for a 2 MiB block of memory

4.2 Table Descriptor Format

When in AArch64 there is a single accepted format for table entries. We will now consider the configuration consequential to a 4 KiB granule size; other choices differ sligtly in translation indexes width and position, but maintain the same core concepts. The table descriptor is 64-bits wide, separated in differently sized fields.

Any table entry is one of two types:

1. **A table descriptor** that points to another table entry in the next level.
2. **A block entry** that resolves directly into a memory range of variable size, depending on the level.

The entry type is defined by the two least significant bits in the descriptor, as depicted in table 4.2; an invalid entry leads to an MMU fault exception.

Entry Type field	Meaning
00	Invalid
01	Block Entry
10	Invalid
11	Table Descriptor

Table 4.1: Page entry types.

Note that not every level can host a block entry; with a granule size of 4 KiB, level 0 does not admit that kind of descriptor.

4.2.1 Table Descriptors

Table descriptors point to a table entry in the next level. Bit fields have the following meaning:

- [**0**] marks the validity of the entry; 1 is valid, 0 is invalid.
- [**1**] is the entry type. It is 1 for table next level descriptors.
- [**2:11**] are ignored/reserved bits.
- [**12**[1]**:47**] is the address of the next level table. It is codified as if it started from the least significant bit, with [11[2]:0] bits assumed as 0. Because of this all page tables must be 4096 (2^{12}) bytes aligned.
- [**48:58**] are ignored/reserved bits.
- [**59**] PXNTable field: *private execute never* bit for subsequent levels of lookup; if set the memory range covered by this and following entries cannot be executed by code at level **EL0**.
- [**60**] XNTable field: *execute never* bit for subsequent levels of lookup; if set the memory range covered by this and following entries cannot be executed.
- [**61:62**] APTable field: *access permission* bits for subsequent levels of lookup; this field enforces permission rules for the memory range indexed by this and following entries, combined in a hierarchical fashion (see table 4.2.1). Subsequent table entries can further restrict the permission rules but cannot loosen them; failure to heed said rules will result in an appropriate abort exception.

---

[1] This value can be different for granule sizes other than 4KiB

[2] see footnote 1

- [**63**] NSTable field: when in secure state this bit specifies the security state for subsequent levels of lookup. When not in secure state it is ignored.

APTable[1:0]	Restriction
00	No effect on subsequent levels of lookup
01	Any access to this memory range from EL0 is forbidden.
10	Memory is read-only.
11	Any access to this memory range from EL0 is forbidden, while it is read-only for higher privilege levels.

Table 4.2: Access Permission Table bit fiels. The two bits can be separated and seen as read-only bit ([0]) and **EL0** access ([1])

4.2.2 Block Descriptors

Block descriptors represent direct access to a block of memory; when one is reached, it is the last stage of the translation process. They contain the following bit fields:

- [**0**] marks the validity of the entry; 1 is valid, 0 is invalid.
- [**1**] is the entry type. It is 0 for block descriptors.
- [**2:4**] memory attributes index field. The value found here indexes a memory address configuration defined in corresponding the **MAIR_EL*n*** register (see section 4.3).
- [**5**] is the non-secure bit. For memory accesses from secure state specifies whether the output address is in the secure or non-secure address map. For accesses from non-secure state this bit is ignored.
- [**6:7**] are data access permission bits. Similar to APTable bits, they are referred to the immediate block of memory (and can be further restricted by previous APTable settings). For possible values see table 4.3.
- [**8:9**] sets the shareability of the memory block, configuring caching capabilities.

AP[2:1]	Access from privileged EL	Access from EL0
00	Read and write	Forbidden
01	Read and write	Read and write
10	Read-only	Forbidden
11	Read-only	Read-only

Table 4.3: Access Permission Bits values. similarly to 4.2.1, the two bits 1 and 0 can be interpreted separately as write restriction for higher exception level and access for **EL0**, respectively.

- [**10**] is the access flag (AF). If it is not set it means the selected entry is accessed for the first time, in which case an MMU abort will be fired. The exception handler should take care of the initialization, set the access flag to 1 and attempt again the memory access.
- [**11**] is the not global bit (*nG*). If a lookup using this descriptor is cached in a TLB, determines whether the TLB entry applies to all ASID values or only to the current ASID value (see section 4.5 for more).
- [**12:47**] is the address to the memory block that is pointed by the descriptor. It is aligned similarly to the address in a table descriptor, but it contains actual memory instead of a next level page table.
- [**48:50**] are ignored/reserved bits.
- [**51**] is the Dirty Bit Modifier (DBM); it is used to keep track of differences between caches and real memory. If it is set, caches should be checked for stale entries. It can be managed either via hardware or software.
- [**52**] is the Contigous bit, a hint bit indicating that the translation table entry is one of a contiguous set or entries and may be be cached in a single TLB entry.
- [**53**] PXN bit: *privileged execute never* bit. If set, the memory block's execution at unprivileged exception levels (**EL0**) is forbidden.
- [**54**] XN bit: *execute never* bit. If set, the memory block's execution is forbidden.
- [**55:63**] are ignored/reserved bits.

4.3 Memory Types and Attributes

As previously mentioned the block and table descriptors contain information about what kind of memory they point to. Every memory address is one of two possible types: normal or device. Normal memory is the one used for most memory regions and where there can be the most invasive optimizations. Normal memory can be heavily cached and is considered to be weakly ordered: the actual number, time and order of access can differ from the logical flow of the program without causing semantic errors.

Device memory is used instead for memory mapped peripherals. When the subject is a device register the order of operations and even repeating the same instruction might yield unexpected side effects; caching must be disabled too, as the content of a device bus is often non deterministic.
Other than the memory type there are fine tuned attributes to distinguish what levels of caching and optimization are allowed.

4.3.1 Shareability

In a multiprocessor system the main memory is obviously shared among all the cores. This can lead to coherency problems in caches external to the individual core (level 2 and level 3, see section 3.4). Hinting who is going to use a certain memory block can greatly help the system when managing caches, improving performance and energy cost. There are three possible shareability options:

Non-Shareable: memory marked as non shareable is assumed to belong to a single actor, thus synchronization of access from different cores is not needed.

Inner Shareable: this memory can be shared inside the processor (i.e. between the cores) but not outside of it (i.e. in the rest of the system).

Outer Shareable: an outer shareable domain is publicly available for every actor in the system and needs to be synchronized on each access.

The shareability attribute is found in the specific field of the block entry descriptor (see section 4.2.2)

4.3.2 Cache policies

When a cache search misses and the entry is not found, the respective cache entry is updated with the value found on RAM. When the search is successful and the operation is a write, deciding what location to update is not trivial.
Regarding normal memory, the OS developer can specify cache policies as any combination of the following three classes:

1. **Cacheability:** Non-Cacheable, Inner Cacheable or Outer Cacheable.
2. **Update policy:** Write Back or Write Through.
3. **Transient:** Transient or Non-Transient

Different cacheability options refer to which fast memory block should be used - internal to the core, internal to the cluster or external - and were described in section 3.4.
When *write back* memory locations are updated only the cache entry is modified and marked as dirty; corresponding main memory locations are updated only when the cache line is evicted or explicitly cleaned. *Write through* is instead a safer (but slower) approach, where an update changes both the cache and the system memory.
Transience is an hint to the cache controller about how long an address should be kept in cache for. This behaviour is implementation defined, not configurable by the developer.

Device memory is treated differently, and thus possesses its own set of attributes:

1. **Gathering:** this property determines whether multiple accesses can be merged into a single bus transaction for this memory region. If the address is marked as *gathering* the processor can, for example, merge two byte writes into a single-word half write.
2. **Reordering:** attribute defining whether accesses to the same device can be re-ordered with respect to each other. If the address is marked as *non Re-ordering* then accesses within the same block always appear on the bus in the order specified by the code, while *Re-ordering* ranges of memory will be subject to read and write chains optimization.
3. **Early Write Acknowledgement:** identified as *E* or *nE*, declares if an intermediate write buffer between the processor and the slave device being accessed is allowed to send an acknowledgement of a write completion. It is only used for closely intertwined operation with special peripherals.

Refer to the Architectural Reference Manual [14] for the exact values and codes of the memory attributes combinations.

4.3.3 MAIR Configuration

Due to the high number of possible combinations (many of which are problably scarcely used), the memory attributes configuration procedure follows a peculiar and somewhat unconventional approach. Instead of having separate fields in the already filled block descriptor format, the programmer defines up to eight memory attribute combinations codified with 8 bits each which are then pushed into the **MAIR_EL*n*** register as if in an array. The block descriptor in turn has a 3-bits field, the memory attribute index, that indexes which configuration to use in said array, in a two step process.

In the image example, the **MAIR** register is populated with two options for cached and non-cached normal memory in the first two bytes. Page table entries will then select a combination using the memory attribute index. Note that any combination can be placed anywhere in the register and it does not need to be correctly initialized in all the eight bytes, as long as only correct entries are indexed by the page table.

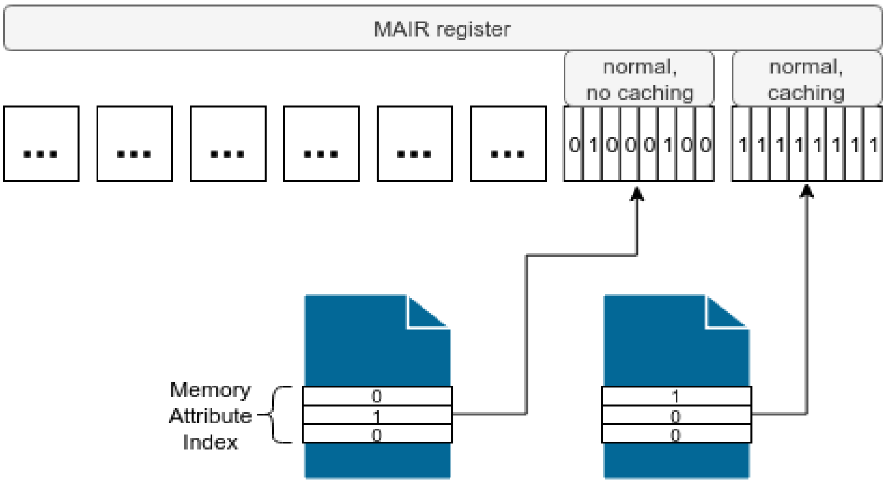


Figure 4.3: MAIR register configuration and indexing by the page table entries.

4.4 Kernel Space Virtualization

In a tipical OS environment multiple processes run concurrently and use dynamically allocated memory and resources. The memory management unit serves to this purpose: every process has its own set of translation tables managed by the kernel and sees a contiguous range of memory at its disposal while in truth it is loaded somewhere in main memory - maybe not even adjacent or complete.
In this multitude of actors we can consider the kernel as a process in its own right - albeit of a superior kind - and because it has no peers and often uses a static memory space the translation mechanism is meaningless, if not cumbersome.

To fix this situation the ARMv8 provides a number of features. One may intuitively imagine different page table references for each Exception Level, but this is only partially the case. **EL0** and **EL1**, the main levels of operation for user processes and kernel, share in fact the same two page table registers: the one assigned to “normal” operation, **TTBR0_EL1**, and the register actually in charge to properly divide user and kernel pages, The former is used to translate addresses in the vast majority of the immense 64-bits addressing space, holding the base address of the page table constructed by the developer; **TTBR1_EL1**, the latter, works in the same way but is only selected when the 16 most significant bits of the address under scrutiny are set to 1, forming a location absurdely far away for any memory bank designed in the foreseeable future.

The kernel can then operate in this mock memory range with a personal page table (possibly as an identity transformation) while normal processes live in lower scope of virtual memory. To accomodate the kernel in this inexistent range the addresses in its code must be set properly. This would require either a linker script instructing to compile for memory starting at `0xFFFF000000000000` (in which case the MMU must be configured and enabled immediately) or compiling the code for relative branch instructions only. The latter approach is considered easier as it works even if the MMU is disabled and only requires the kernel entry point (the exception table) to be tweaked when address translation is eventually turned on.

As a side note, the condition to select the **TTBR1_EL1** register can be limited to the second most significant byte of the address set to `0xFF`; the first byte can then be freely used by the kernel software for personal purposes.

An example use case might be in support of object-oriented programming languages: as well as having a pointer to an object, it might be necessary to keep a reference count that keeps track of the number of references, pointers

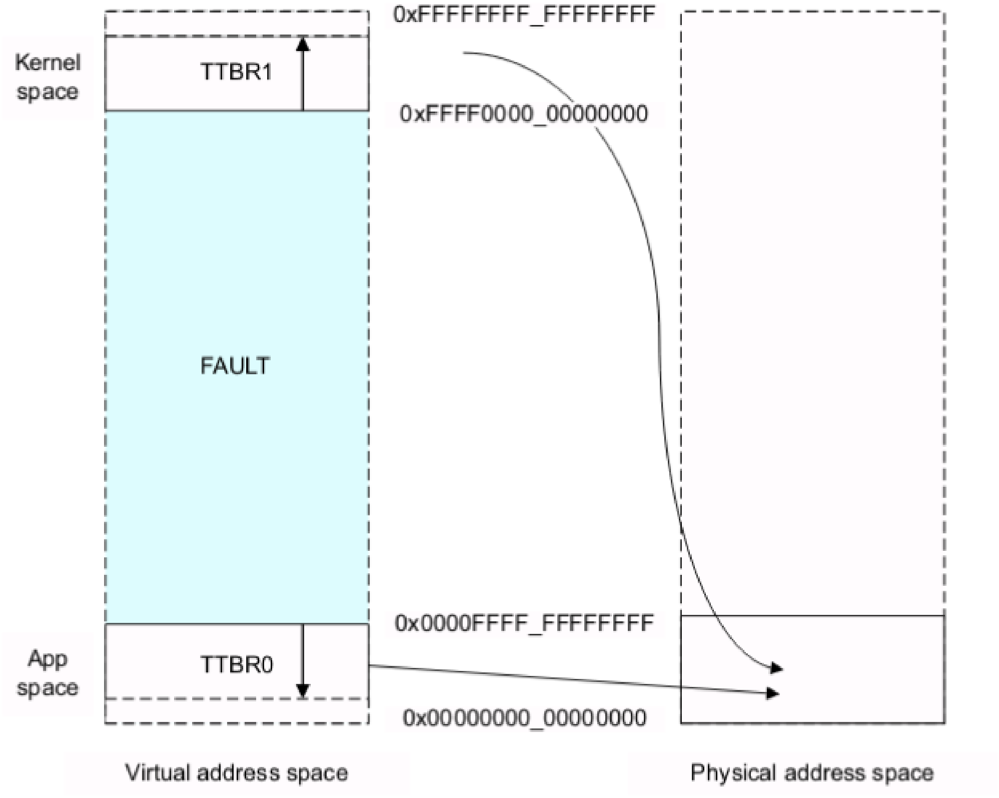

0xFFFFFFFF_FFFFFFFF
Kernel space
TTBR1
0xFFFF0000_00000000
FAULT
0x0000FFFF_FFFFFFFF
App space
TTBR0
0x00000000_00000000
Virtual address space
Physical address space

Figure 4.4: In this image kernel and user spaces are positioned at the opposites of the address range but both are mapped in the same physical area.

or handles that refer to the object, for example, so that automatic garbage collection code can deallocate objects that are no longer referenced. This reference count can be stored as part of the tagged address rather than in a separate table, speeding up the process of creating and destroying objects.

4.4.1 EL2 and EL3 Translation Process

Virtualization and secure exception levels have their own page tables. Since they act as overlay for one or more guest operating systems when enabled there are two translation stages: the first one is performed by the OS, using **TTBR*n*_EL1** as already explained; the result from this stage is then fed to a second stage, with tables found in **TTBR0_EL2** and **TTBR0_EL3**.

Such complex tools are out of the scope of this work.

4.5 Translation Lookaside Buffer

The Translation Lookaside Buffer (or TLB) is simply cached memory for address translation results: when a virtual address is to be translated the TLB is checked first; if the address is found (TLB hit) then the cached value is used. If not (TLB miss) a page table walk is performed and the result is stored in the TLB. Alternately, an MMU fault can be configured to fire.

The TLB and MMU intertwined operation works very differently in ARM compared to MIPSEL architecture and with an arguably easier approach for a novice. The TLB component activity is almost invisible to the OS developer not caring for particular performance optimizations, so it can be safely ignored. The only essential part is the configuration of proper page tables for each process.

4.5.1 Trivial Approach

Once the MMU is activated and page tables are initialized the kernel must make sure every process can only see its virtual memory share during its designated time slice. This can be achieved by creating a different page table per process and simply substituting it completely every time there is a context switch. A problem presents when a process asks for the same virtual address, that should however be translated in a different physical address, of one of its peers; we shall call them process a and b. If the entry relative to a is found in the TLB by b its value will be returned without performing a page table walk, and will result in the wrong memory being accessed.

Without delving too deeply into MMU operation, a simple solution will be to flush the TLB cache every time there is a context switch. This will effectively deny most of the optimization brought by the cache, but will also prevent incorrect translations.

4.5.2 ASID Approach

A more elegant and efficient solution consists in using an Address Space ID (ASID) to keep track of process property in TLB entries. The ASID is arbitrarily assigned to processes by the kernel and stored in the two most significant bytes of either **TTBR0_EL1** or **TTBR1_EL1** When the non-global (nG) field of a block entry in the page table is set the current ASID is saved alongside the address in a TLB entry. Subsequent lookups for that address in the TLB cache only match if both the address and the saved ASID correspond to present values.

Chapter 5

Overview of the BCM2837

The BCM2837 is the System-on-Chip produced by Broadcom that is used for most of the Raspberry Pi family of boards, and for the third version specifically. Some of them are built with variants like BCM2836 (for the Raspberry Pi 2) and BCM2835 (the first used, for the Raspberry Pi 1): the scarce documentation is only available for BCM2835 [10] (and partially for BCM2836 [11]) allegedly because nothing changes from the developer perspective; the actual differences have been figured out mostly through reverse engineering from the source code of various Linux distributions.

The BCM2837 contains the following peripherals, accessible by the onboard ARM CPU:

- A system timer.
- Two interrupt controllers.
- A set of GPIOs.
- An USB controller.
- Two UART serial interfaces.
- An external mass media controller (the microSD interface).
- Other minor peripherals (I2C, SPI,. . .).

5.1 Boot Process

As is the case for many similar boards the ARM CPU is not the main actor, but actually more of a coprocessor for the Videocore IV GPU installed alongside it.

On reset the first code to run is stored in a preprogrammed ROM chip read by the GPU, called the first-stage bootloader. This first bootloader

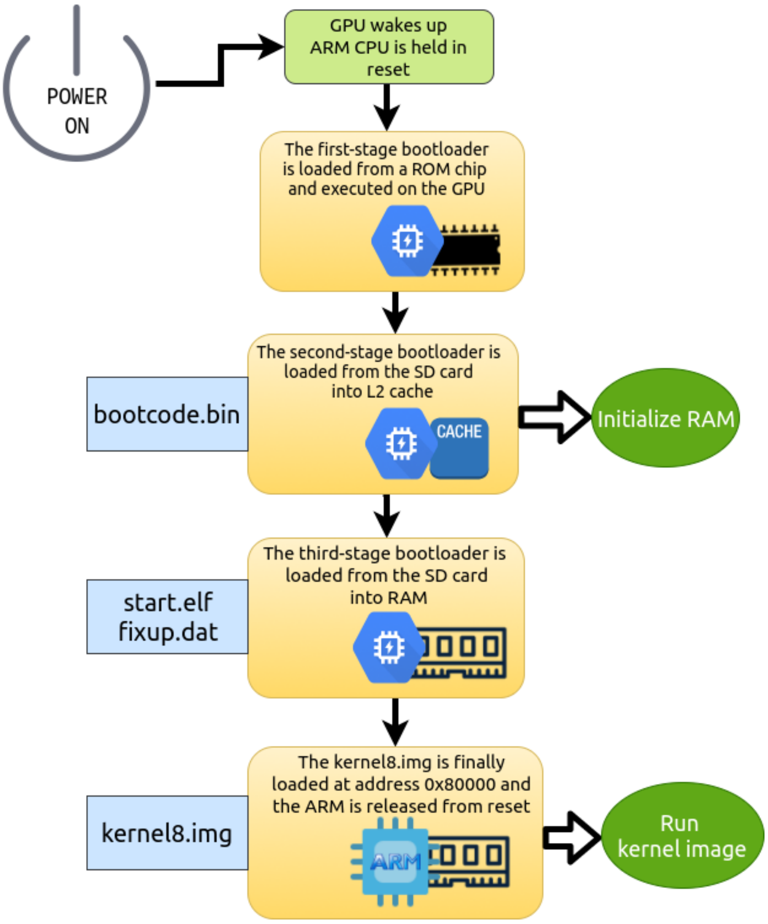

POWER
ON
GPU wakes up
ARM CPU is held in reset
The first-stage bootloader is loaded from a ROM chip and executed on the GPU
bootcode.bin
The second-stage bootloader is loaded from the SD card into L2 cache
CACHE
Initialize RAM
start.elf
fixup.dat
The third-stage bootloader is loaded from the SD card into RAM
kernel8.img
The kernel8.img is finally loaded at address 0x80000 and the ARM is released from reset
ARM
Run
kernel image

Figure 5.1: Explanatory diagram on BMC2837 boot sequence

looks for the first partition on the microSD card (which has to be formatted as FAT32), mounts it and loads (if present) a file called `bootcode.bin` from the partition. This binary is part of the Broadcom proprietary firmware package, and is considered the second-stage bootloader.
At this point of the boot sequence the main memory is still not initialized, so the second-stage bootloader is run from the L2 memory cache. This firmware initializes the RAM and in turn loads on it another file from the microSD card, start.elf.
Another firmware for the Videocore, `start.elf` has the responsibility to split the RAM in two parts for the GPU and the CPU; after that it reads the config.txt file (if present) and loads its parameters starting at address 0x100. Finally it does the same with the kernel image and passes control to the ARM CPU.

Every step up until the loading of the kernel image in memory is handled by the GPU and can be safely ignored after an initial setup.

5.1.1 MicroSD Contents

The microSD must have its first partition formatted as FAT32; there are no further restrictions on following partitions. The absolute bare minimum contents are just four files:

1. **bootcode.bin**: second-stage bootloader, necessary for the GPU to load the third-stage bootloader.
2. **start.elf**: third-stage bootloader, necessary for the GPU to load the kernel image into main memory.
3. **fixup.dat**: a file containing relocation data to be referenced by `start.elf` when loading into RAM; this allows for the same firmware to be used for all versions of the Raspberry Pi, which range in memory from 256MB to 1GB. If not included the board might still boot, but it will likely only report a total of 256MB regardless of the actual installed memory banks.
4. **kernel8.img**: kernel binary for the ARM CPU.

Of those four files only the kernel image is user provided; the remaining firmware is distributed and updated in compiled form by the Raspberry Pi foundation with proprietary licensing from Broadcom.

5.1.2 Configuration

It is possible to configure in different ways the boot process by combining different firmware binaries and `config.txt` options, but MaldOS always uses the default with no extra steps needed; this is to ensure the usage is kept as simple as possible and because the base behaviour never presented any issue. Of all the available options, only the following two were ever considered (but still never implemented).

Architecture

The Cortex A-53 can run both ARM32 and ARM64 code. The choice is dictated by the name of the kernel image: `kernel8.img` makes the CPU start in AArch64 mode, while `kernel7.img` would start in AArch32. The latter option is preferred.

Kernel Loading Address

The GPU loads the kernel image starting at address `0x80000` in RAM for the Raspberry Pi 3. By adding a `config.txt` file to the microSD card and using the `kernel_address` parameter the image file will be loaded at the specified starting point. Similarly, by setting the `kernel_old` parameter to 1 the binary will be loaded at the beginning of the main memory, at address `0x0`. This alias is present for compatibility reasons.

Although these options can bring a more clean memory disposition, it was decided the advantages were not worth adding an additional file to the necessary setup. Additionally, while the Raspberry Pi hardware correctly interprets these commands the Qemu emulated machine is not entirely loyal to reality and actively resists any attempt to move the kernel to locations other `than 0x80000` (more details can be found in section 9.3).

Memory Split

As previously mentioned the two main actors on the BCM2837, the quad-code Cortex-A53 ARM and the Videocore IV GPU, share the same 1GiB RAM space. When not instructed otherwise the `start.elf` bootloader fixes the separation at address `0x3C000000`, keeping 64MiB to himself and leaving the rest to the CPU.

This split can be increased in favor of the GPU or minimized even further using specific `config.txt` parameters. The only graphical feat required by this work is the display of a simple framebuffer to present textual output; therefore a reserved memory partition of 64MiB is more than sufficient. It could be in fact reduced further to 16MiB, but as for the kernel load address adding the `config.txt` file was judged unneded effort on the user's side.

5.2 Videocore IV

After the control is passed to the ARM CPU it is never returned to the GPU. The graphical processor however still has responsibility over some peripherals and can carry on work under specific requests. The mean of communication between the two processing units is the shared RAM memory (and part of the interrupt controller), specifically under the Mailbox interface.

5.2.1 Mailboxes

Mailboxes are the primary means of communication between the ARM and the Videocore firmware running on the GPU. A mailbox is nothing but a memory address with special access modes tied to an interrupt signal for the receiving end. Mailboxes consist of one or more 32-bits registers providing status information, read and write access. If a value is written on the right memory location and the mailbox is ready to accept data, an interrupt will be fired and the receiver will have the chance to read the message and act accordingly. The data is usually another memory location, whose pointer has been written into the actual mailbox, containing arbitrarily long and elaborate parameters. ARM cores can interrupt each other by using single register mailboxes for inter processor communication while the relative extra cluster transmission mean have as many as 9 registers each, although the function of most of them is unknown.

Register Name	Function
read	register to be read for command responses. The 4 least significant bits are the mailbox channel
reserved	
reserved	
reserved	
peek	like the read register but does not remove the value from the message FIFO
sender	information about the sender
status	status register, carrying information about the mailbox state (e.g. the message FIFO is empty or full)
config	unknown function
write	register where to write a pointer to the command structure. Adds the message to the GPU's FIFO.

Table 5.1: CPU-to-GPU mailbox registers list.

Regarding the CPU-to-GPU mailbox, additional care must be taken to check whether the mailbox is full or empty by inspecting the two most significant bits of the status register. After the command has been executed the status register should be polled again for change and the memory structure passed to the other core will contain the requested information (if any). The Videocore accepts lists of multiple commands as messages and organizes responses in an ordered FIFO that is emptied by reading the mailbox until the status register signals there is no more mail.

The data address to be written on the mailbox must be 16 bytes aligned in memory, as the lowest 4 bits must be overwritten with the so called mailbox channel number, a parameter detailing the nature of the request. As of time of writing only two channels are defined: channel 8 for requests from ARM to the Videocore and channel 9 for requests from the Videocore to the ARM. Apparently, channel 9 exists but has no definied behaviour.

The buffer whose address is written on the mailbox must contain properly structured data for specific requests. Some of the possible commands from the ARM CPU to the Videocore include:

- Get Broadcom firmware revision number.
- Get board model and revision number.
- Get board MAC address.
- Get current CPU-GPU memory split.
- Get or set power state for all the devices on the board.
- Get or set clock state for all the devices on the board.
- Get on board temperature readings.
- Control special GPIOs, like the on board activity led.
- Execute code on the Videocore.
- Require and manage a framebuffer to be displayed over the HDMI.

Command structure

The memory pointed by mailbox messages is interpreted by the GPU in a specific and mostly homogeneous format (see table 5.2): there is an header and a tail enclosing a list of tags (or commands) that can have a slightly variating structure but are generally regular.

Field Size	Meaning
32 bits	buffer size in bytes (including the header values, end tag and padding)
32 bits	holds request/response code; success or failure and number of bytes returned
variable	list of concatenated tags
32 bits	0x00000000 to signal message end
variable (if any)	padding to 16 bytes aligned address

Table 5.2: Mailbox message format; tag structure is specified in table 5.3.

Field Size	Meaning
32 bits	tag identifier (specific 4-bytes code)
32 bits	value buffer size in bytes
32 bits	request or response code (the most significant bit selects which one)
variable	value buffer for parameters and data
variable (if any)	padding to 4 bytes aligned address

Table 5.3: Mailbox tag structure.

Framebuffer

The HDMI controller is managed entirely by the GPU, and the ARM core has no way to interact with it directly. Instead, it can ask through the mailbox property channel for the Videocore to set up a framebuffer in its own memory share and directly access it, configuring screen resolution in the process. The Videocore will then proceed to continously flush the framebuffer's contents on the screen. This is a very convenient design choice, removing a great deal of effor from the OS developer to see output displayed on screen.

5.3 Peripherals

What follows is a list of all the peripherals used in the project with the core functioning (registers and command codes) explained for each of them. Device peripherals are connected to the ARM CPU through memory mapped I/O (MMIO); their registers and buses are mapped in RAM starting from address `0x3F000000`, as if the main memory of the system extended beyond 1GiB. Only the peripherals that find a purpose in MaldOS will be explored here.

5.3.1 GPIO

The Raspberry Pi 3 board contains 54 General Purpose Input Output pins, out of which only 26 are directly present in the 40 pin strip on the board. The ones that are missing serve either internal purposes (see section 5.3.2) or are not used altogether.

Such a simple and low level peripheral is unlikely to be included into an Operating System oriented project; nonetheless a simple yet effective library

was created when first approaching the hardware (blinking a light is a close second to the obligatory “Hello World” example) and is still necessary to set up some other devices like the EMMC and UARTs. A brief description of the device’s memory map is included for completeness.
There are 13 different kinds or 32-bits register in this peripheral. To accomodate for 54 GPIOs each register class can repeat a number of times, depending on how many bits are dedicated to the individual pin (i.e. 3 bits per GPIO result into 6 registers; a single bit each can be covered by just two 32-bits registers). The GPIO memory map starts at address `0x3F200000`

Function Select: GPIOs can have 8 possible functions: input, output and “alternate function *n*”, with *n* spanning from 0 to 4. Alternate functions are relative to specific peripherals (see sections 5.3.2 and 5.3.3). The function is codified using 3 bits adequately positioned in the register (e.g. bits [0:2] of the first register refer to GPIO 0)

Output Set: when GPIO *n* is configured as an output the *nth* bit of this register class can be set to pull high the corresponding latch circuit. Writing 0 has no effect.

Output Clear: when GPIO *n* is configured as an output the *nth* bit of this register class can be set to drive low the corresponding latch circuit. Writing 0 has no effect.

Pin Level: read only register that yields a bitmap with the actual voltage level registered in every pin.

Event Detect Status: bits in this register are set whenever an event is detected. Events are configured by the following six registers.

Rising Edge Detect Enable: setting the *nth* bit in this register will result in the event detect status register being updated on a rising edge (i.e. a transition from low to high voltage) for GPIO *n*.

Falling Edge Detect Enable: setting the *nth* bit in this register will result in the event detect status register being updated on a falling edge (i.e. a transition from high to low voltage) for GPIO *n*.

High Detect Enable: this register ties a registered high level in a GPIO to a value of 1 in the corresponding bit of the event detect status register.

Low Detect Enable: this register ties a registered low level in a GPIO to a value of 1 in the corresponding bit of the event detect status register.

Asynchronous Rising Edge Detect Enable: akin to the rising edge detect register but works asynchronously with respect to the system clock, allowing to register faster transitions.

Asynchronous Falling Edge Detect Enable: akin to the falling edge detect register but works asynchronously with respect to the system clock,

Pull Up/Down: internal pull up or down register. Once a value is written the actuation must be finalized with the next register.

Pull Up/Down Clock: after writing to the pull up/down register the corresponding bit in this register must be swapped to “clock in” the change.

5.3.2 External Mass Media Controller

The MultiMediaCard (MMC for short) is the open memory card standard the vast majority of producers turn to when adding considerable amounts of storage to size-restricted solutions. The BCM2837 includes an MMC standard-abiding controller provided by Arasan to access microSD cards, called External Mass Media Controller (EMMC) in the peripheral datasheet [10].

The peripheral operates with a fair degree of autonomy from both the CPU and the GPU. It has a configurable clock separated from the board and is accessed through a set of memory mapped registers. The command set accepted by the controller has a direct reference to the MMC standard but the device also manages independently the most mundane operations (i.e. calculating cyclic redundancy checksum for command payloads).

The creation of a driver library for this kind of device is no trivial task; it was kept as simple as possible and only made possible by existing examples in turn extracted from a reverse engineering effort of professional software (i.e. Raspbian and various RTOSes for Raspberry Pi). This is also due to the fact that the detailed description of the peripheral should be found, according to Broadcom’s documentation, in an elusive datasheet (*“SD3.0_Host_AHB_eMMC4.4_Usersguide_ver5.9_jan11_10.pdf”*) that instead is not publicly distributed by Arasan outside of their direct collaborators (and thus should not have been mentioned by Broadcom in the first place). Consequently the author has only a vague understanding of how the controller actually works.

For this reason rather than listing the registers and their function the explaination for this peripheral will follow the initialization and read/write procedures of the MMC. It is by no means complete and should not be used as reference.

Initialization

First and foremost, the pins that are physically connected to the MMC interface must be properly set up; GPIOs 47 to 53 are set to alternate function number 3 with an internal pull up attached. Those seven lines make up the serial interface to the media: one to distinguish between commands and data, one is the clock feed, one is used to detect whether a card is actually plugged in and the remaining four form up a nibble-sized parallel bus. On the controller side it is good practice to read as soon as possible the specification version of the controller, which can have significant differences in the command protocol from one version to the next. Once the GPIOs are ready the controller should be reset and the clock speed initialized. After that there is a fixed sequence of commands to boot up the controller and have it ready to receive directives.

In no particular order during this process interrupts should be enabled to avoid heuristic delays when waiting for a response. The device signals interrupt lines through a couple of dedicated registers that can be polled for activity but it is unclear whether those same interrupts can be routed to the ARM handlers: to the best of the author's efforts it was not understood (and ultimately unnecessary for proper functioning of the system, given that the abstraction layer does not expose direct access the the EMMC).

SD Commands

Sending a specific command to the controller is a complex routine in itself. First, a status register should be inspected to check if the command line is still being used by a previous order; then the command code and arguments can be passed to the corresponding registers. Finally the interrupt register must be polled to know precisely when the controller has finished elaborating and the response registers (16 bytes in total) can be read to know the result.

Reading or Writing a Block

The microSD card is both read and written in 512 bytes long memory blocks. After checking if the data line is free via the status register the controller should be notified of the block size (512 bytes) and number that compose the data transfer.

Different commands can be given to the controller depending whether the operation is a read or write and if it is comprised of one or multiple blocks; regardless, starting block index should be passed as argument. When everything is ready the data register can be read or written and each access will push the seek index forward by 8 bytes (the size of the register). A final specific command code will instruct the transfer to cease.

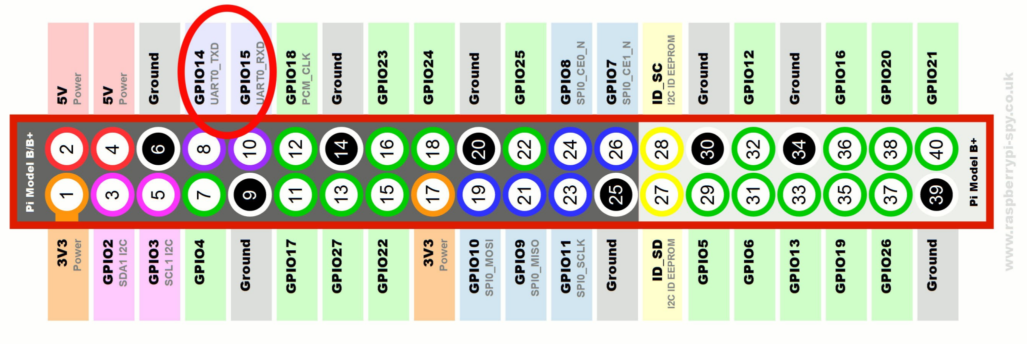

5V Power
5V Power
Ground
GPIO14 UART0_TXD
GPIO15 UART0_RXD
GPIO18 PCM_CLK
Ground
GPIO23
GPIO24
Ground
GPIO25
GPIO8 SPI0_CE0_N
GPIO7 SPI0_CE1_N
ID_SC I2C ID EEPROM
Ground
GPIO12
Ground
GPIO16
GPIO20
GPIO21
Pi Model B/B+
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
Pi Model B+
3V3 Power
GPIO2 SDA1 I2C
GPIO3 SCL1 I2C
GPIO4
Ground
GPIO17
GPIO27
GPIO22
3V3 Power
GPIO10 SPI0_MOSI
GPIO9 SPI0_MISO
GPIO11 SPI0_SCLK
Ground
ID_SD I2C ID EEPROM
GPIO5
GPIO6
GPIO13
GPIO19
GPIO26
Ground
www.raspberrypi-spy.co.uk

Figure 5.2: Highlight of UART reserved GPIOs

5.3.3 UART Serial Interface

There are two UART serial peripherals on board of the BCM2837: UART0 and UART1. They can both be connected to the same group of six GPIOs to relocate the transmit and receive line; however, of those six pins only two (GPIO 14 and 15) are externally accessible on the Raspberry Pi. This means that, at any time, either of those pins can be connected and work for only one of the two serial interfaces. Even if this is undoubtedly a limitation it can pose an interesting concurrent programming challenge for a student, as both can run successfully if hardware settings are properly alternated.

Those devices bear a strong similarity to μMPS2's terminal devices, both having similar registers to check the current status and read or write a character on the interface. For this reason, except for the initialization of the peripheral which is done entirely by the hardware abstraction layer, they are left essentially untouched to be managed by students approaching the project. In comparison to the emulated devices the only real difficulty lies in a less organized register format, having about four registers scattered over a larger memory area instead of a compact structure. After providing a focused and complete documentation of said registers, this complication should be easily overcome.

UART0

The UART0 is a fully fledged asynchronous serial interface, abiding to the PL011 ARM specification [12]. To properly run on real hardware, the corresponding pins must be configured to use the alternate function number 0 with no internal pull up or down. Its register are located starting at the address `0x3F201000`, each of them is 32 bits wide and they are organized as follows (some unimportant ones are omitted for brevity):

Data: this register contains the first character present in the receive FIFO and can be written to send an outgoing character to the transmit FIFO. Addidionally, it presents an error report of the ongoing connection, with a specific bit for every condition (overrun, break, parity, framing).

RSRECR: a redundant register for error conditions.

Flag: contains various flags on the current state of the UART, like state (full or empty) of the transmit and receive FIFOs and whether the UART device is busy or idle.

IBRD: integer part of the baudrate divisor: when configuring the device the baudrate is established as a floating point divisor prescaling the system clock. This is the integer part.

FBRD: Floating point part of the baudrate divisor.

Line control: this register manages configuration options like parity, number of stop bits, word length and FIFO abilitation.

Control: this register controls the actual peripheral; mainly used for enabling and disabling the whole device.

IFLS: interrupt FIFO level selection register. It is used to establish at which percentage each FIFO (transmit or receive) triggers the corresponding interrupt. Possible values range from 1/8 to 7/8.

Interrupt mask: allows to mask specific interrupts tied to the peripheral, such as those fired on reception and transmission of a character

Raw interrupt: read only register updated with currently pending interrupts, regardless of the mask settings.

Masked interrupt: same as the raw interrupt register but with the masked interrupt lines excluded.

Interrupt clear: register to be written to clear pending interrupts.

Of all those registers, the only ones a student should really care about are data, flag, interrupt mask, masked interrupt and interrupt clear. All the others are used for the initialization of the peripheral, which is handled by the hardware abstraction layer and should not be changed.

The serial interface is configured as 8 bit wide, no parity bit and with a baudrate of 115200. The FIFOs are disabled for simplicity, so they act like a 1-character deep buffer.

UART1 or Mini UART

The UART1 is part of the group of auxiliary peripherals, together with two SPI interfaces. In comparison with UART0 it has much more restricted functionality, but still enough for a simple educational project. For example, it does not provide framing error detection or parity bit management, features that are either disabled or ignored even in its more complete counterpart. To properly run on real hardware, the corresponding pins must be set to use the alternate function number 5 with no internal pull up or down. Its registers are located starting at the address `0x3F215040`, each of them is-32 bits wide and they are organized as follows (some unimportant ones are omitted for brevity):

IO: reading from this register yields the first character present in the receive FIFO, while writing it inserts the data into the write FIFO.

IIR: register for enabling receive and transmit interrupts. If the first bit is set an interrupt line is asserted whenever the transmission FIFO is empty; if the second bit is set an interrupt line is asserted whenever the reception FIFO is not empty.

IER: register holding information about which interrupt is pending (if any).

LCR: controls whether the Mini UART works in 8 bit or 7 bit mode.

LSR: line control status; used to determine if the device is ready to accept new data or if there are received characters to be read.

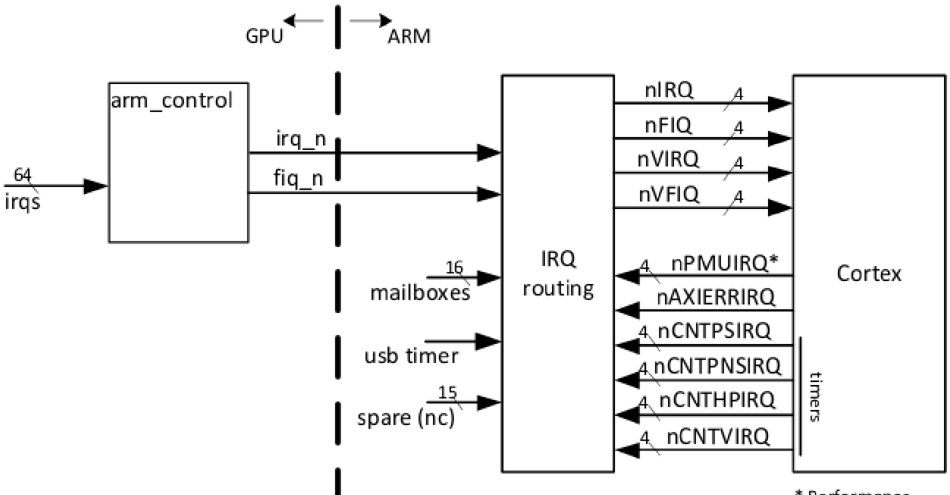
CNTL: control register to enable (in a separate fashion if so desired) the receive and trasmit lines.

BAUD: 16 bit baudrate counter, to be set directly to the desired value.

Again, since the abstraction layer takes care of the initialization procedure the user should really care about four registers: IO, IIR, IER and LSR. The serial configuration is the same as the UART0.

5.4 Interrupt Controller

The BCM2837 SoC has at least two devices acting as interrupt controllers. One of them is clearly defined in the peripheral datasheet [10], while the other is not clearly identified but hinted at thorugh register definition in a later revision [11]. Those are here arbitrarily named Base Interrupt Controller (BIC) and Generic Interrupt Controller (GIC). These two interrupt controllers are cascaded, meaning that 64 interrupt lines are wired to the BIC which in turn compresses them into 2 interrupt lines for the GIC controller; additionally, the GIC also receives some interrupt lines from mailboxes and USB.


GPU
ARM
arm_control
64
irqs
irq_n
fiq_n
16
mailboxes
usb timer
15
spare (nc)
IRQ
routing
nIRQ
4
nFIQ
4
nVIRQ
4
nVFIQ
4
4
nPMUIRQ*
nAXIERRIRQ
4
nCNTPSIRQ
4
nCNTPNSIRQ
4
nCNTHPIRQ
4
nCNTVIRQ
timers
Cortex

* Performance monitoring Unit

Figure 5.3: BCM2837 interrupt controllers configuration

From a practical standpoint there are often several redundant registers indicating which interrupt line is being asserted at any given moment. There is no apparent drawback in ignoring most of them and just reading each device-specific register to discern which source fired the exception, as the general interrupt organization is very confused and obscure. Interrupt functionality was achieved mainly through examples and reverse engineering regarding the specific device taken in consideration at the time. What follows is a brief listing of interrupt related configuration for the devices used in this work.

UART Both UART devices are cascated through the two interrupt controllers; although they can be checked via registers in both controllers, it is suggested to only read the Masked IRQ and IIR registers of the respective peripheral.

ARM timer Being this an interrupt internal to the ARM processor its status has only been checked against the innermost interrupt controller (GIC). It is not clear whether it is present in the BIC as well.

Inter processor mailboxes Possibly the sole source clearly depicted from the documentation, its presence can be understood from the corresponding register in the Generic Interrupt Controller (as indicated by 5.3).

EMMC Its interrupt lines are asserted by a device-specific register and seemingly nowhere else.

5.4.1 Inter Processor Interrupt (IPI)

In a multicore system such as the Raspberry Pi 3 the need arises for a privileged communication channel between each core. The ARM Cortex-A53 does not provide an explicit method to do so, and it is left to the Generic Interrupt Controller to provide. Similarly to the interface between ARM and Videocore there are mailboxes between the four cores of the CPU as well.

The operation of those inter core mailboxes is much more straightforward than the CPU-GPU counterpart. There are four mailboxes for each core and for each one the GIC exposes three classes of control registers, for a total of 36 registers [1].

Mailbox Control four registers of this type in total, one for each core and covering its four mailboxes. They enable an interrupt or fast interrupt line for each mailbox.

Mailbox Write-Set four registers for each mailbox in every core, so sixteen of them in total. They are write only and are used to put the actual data in the mailbox. Upon write the corresponding enabled exception (if any) is fired for the selected core.

Mailbox Read and Write-Clear one register for each corresponding Write-Set register. They can be read to receive the data sent by writing in the Write-Set register, and have to be written to disarm the interrupt line. Each bit of the register is independent in firing the interrupt, so to completely clear it the same content that was read from the register must be written back on it.

---

[1] Note: the first kind of register covers all four mailboxes for each core

Chapter 6

Emulated peripherals

The Raspberry Pi 3 (or any other version or model) does not have many peripheral devices to toy with. In part this is due to its heritage of low resource board, and in part to extensibility through four generic USB ports and 40-pin header, allowing for a wide range of HAT (Hardware Attached on Top) extensions and external USB devices. In the perspective of an educational project however this is a severe limitation. While μMPS2 and μARM can each bring five device types with eight possible instances per type, the Raspberry Pi has only two really usable devices: the two serial interfaces (that strongly resemble μMPS2 terminals).

Other options cannot be considered for multiple reasons:

- The screen is simple and usable, but lacks educational value. It is nothing more than a buffer to write on; the GPU then manages actually sending the data to the screen.
- The EMMC interface is far too complex to be used by students. The professor would need either to spend a great deal of time and effort to explain how it works or provide a library to access it, in contrast with the philosophy of this work.
- The USB controller suffers a even worse degree of complexity, to the point where developing a support library would be a monumental task in itself. Last but not least, it is not supported by Qemu.
- The network interface is unfortunately not directly connected to the ARM but instead managed by the USB controller.
- Other auxiliary peripherals like the two SPI interfaces would be perfect for the task: although arguably too low level, many modern motherboards include SPI or I2C controlled peripherals, making it an intersting addition to the program. However, those are not supported by

Qemu.

To mitigate this problem, three classes of new devices have been implemented as emulated peripherals in the hardware abstraction layer. Using μMPS2 as a reference, these classes are tapes, disks and printers.

While building an entire emulator would give full control over the device interface, in this work the emulation is carried on to the best level permitted by a bare metal environment, leaking some imperfections on the exposed controls.

6.1 Emulated Device Interface

Initially emulated devices were made accessible via fake registers: simple pre-established memory locations that were frequently polled (once every 100 μs) by the abstraction layer. Though most similar to the μMPS2 approach, this idea had significant flaws.

- fake registers had no read or write limitations; location that should logically have been read only could be modified without limit, leaving the device in an incoherent state.
- polling was a frail mechanism, prone to error and race conditions. A real device starts working the moment its registers are written, while in this scenario the user had to wait for the contents to be read by the abstraction layer. This leads to an unintuitive programming path, requiring the user to either poll for changes in turn or use an `swi` assembly instruction to wait for the polling interrupt.
- generally speaking, it is good practice to avoid polling when possible.

A solution was found that strays from the previous work's approach but better fits the new environment and allows for a cleaner emulation: using mailboxes.

Some of the peripherals on the BCM2837 board are already managed by the GPU through mailboxes, like the HDMI controller or the on-board activity led. In a very similar way, the abstraction layer is notified of a new command for printers, tapes or disks by a write to the inter core communication mailbox. Specifically, the mailbox 0 of the first core is reserved for emulated devices control. This behaviour is transparent to the user because it raises a FIQ instead of a normal interrupt, and thus it can be received at any moment.

A command to a emulated device is then issued by writing some value to the mailbox 0 write-set register of the first core, found at memory address

`0x40000080`. The value must have the following format: the two least significant bits are the device number and the two following bits are the device class. The upper 28 most significant bits should point to a 16-byte aligned address containing a register structure for the selected device.

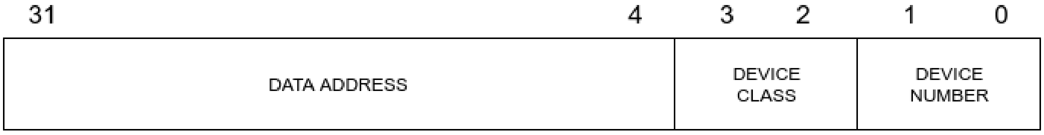


Figure 6.1: Mailbox structure

This should remind the reader of the mailbox communication protocol used by ARM to talk with the Videocore, with the channel number encoded in the 4 least significant bits. Since it is a mechanism already present in the system it fits naturally in the development process.

The “register” structure that should be pointed by the mailbox address is nearly identical to the device register layout in μMPS2 and μARM.

Field #	Address	Field name	Size
0	base+0x0	STATUS	32 bits
1	base+0x4	COMMAND	32 bits
2	base+0x8	DATA0	32 bits
3	base+0xC	DATA1	32 bits
4	base+0x10	MAILBOX	32 bits

Table 6.1: Device registers layout

Every device can have special functions for each register; what follows is a general description.

STATUS contains the device state.

COMMAND contains the command code to be executed.

DATA0 & DATA1 carry additional arguments for the command.

MAILBOX is written by the system to notify the command has been carried on.

Since this structure is nothing but a user memory location, fields like **STATUS** and **MAILBOX** are uninitialized at first; only **COMMAND**, **DATA0** and **DATA1** must contain proper data. Once the abstraction layer has received the fast interrupt and parsed the registers it copies the internal state of the device onto the provided memory location, populating all of its fields.

After receiving the mailbox the abstraction layer sets the **MAILBOX** field to 1. This however does not mean the operation has been finished successfully, because real world devices take time to operate; as such, there are fabricated delays between commands and execution. Both the initial interrupt and the subsequent delay are handled at **EL1** by fast interrupts.

Once the execution is complete an interrupt is asserted. Interrupt lines for emulated devices are emulated as well with a memory location allocated for the task, at base address `0x0007F020`.

Interrupt line #	Address	Device class	Size
0	base+0x0	Timer	8 bits
1	base+0x1	Disk	8 bits
2	base+0x2	Tape	8 bits
3	base+0x3	Printer	8 bits

Table 6.2: emulated interrupt lines.

Interrupt lines for emulated devices are nothing more than an array at a specific memory address that is kept updated by the abstraction layer. Whenever an operation terminates a bit is set for the user to clear; until that happens interrupts will keep being raised in the form of a zero-delay timer (i.e. until there is an emulated interrupt line asserted the abstraction layer sets a timer for 0 microseconds in the future, ensuring the kernel responsibility to clear it).

The next sections are dedicated to the specific device classes. The interface and operating rules were copied almost identically from μMPS2, so the interested reader is redirected to its documentation for the details [6]; here are listed some general informations and implementation peculiarities.

In addition to the commands defined in μMPS2 there is another recurring instruction, the **READ_REGISTERS**, an alias for the value codified as `2`. This command is instantaneous in every case and simply copies the device registers into the structure specified in the mailbox. It substitutes reading the system registers without causing any effect on the peripheral.

The next chapter covers in greater detail the inner mechanisms of the emulated devices.

6.2 Printers

MaldOS supports up to four parallel printer interfaces able to transmit a single byte of data at a time. Characters sent to a printer are displayed on the HDMI connected display in the printer's section. The data structure expected by the abstraction layer is the same as every other device (see table 6.2) despite the **DATA1** field not being used.

When the device writes receives a mailbox message and copies its state into the register structure **STATUS** is updated with the current state of the device (busy, ready, under error condition). Even an unrecognized or impossible command will result in this update, albeit followed by the proper error condition.

The **COMMAND** fields accepts directives to reset, acknowledge, print a character or read the current registers. Commands issued while the peripheral is busy will be ignored.

The printer's interface maximum throughput is 125 KB/sec.

6.3 Disks

MaldOS supports up to four readable and writable disk devices. The disk current position is indexed by a combination of head, cilinder and sector to select 4KiB blocks of persistent storage memory. Reading **DATA1** always returns the geometry of the installed disk: maximum number of cylinders, heads and sectors.

When reading or writing a block the register structure should specify a pointer to an adequately spaced memory range in **DATA0**. Disks are DMA operated, so the abstraction layer will take care of moving requested information to or from the medium.

Whenever a command is issued the **COMMAND** field should be filled with additional arguments, like the cylinder to reach or sector and head to read from.

The disk write and read speed are dependant on the (emulated) hardware and can be configured for each device before installing it.

6.4 Tapes

Tape devices are very similar to disks, with the two major differences being they are read only and are accessed in a sequential fashion. They are organized in 4KiB blocks as well, with each block terminating in a 4-bytes marker that is copied into **DATA1**. After a reset it reads *TAPE START*

and changes after reading a block or otherwise moving the head.
Possible values for markers are *END OF BLOCK*, *END OF FILE* and *END OF TAPE*. They are used to navigate among the files that have been assembled together when creating the tape.
Like disks they are DMA-operated and **DATA0** should contain a pointer to the memory block to populate.

Chapter 7

Project Internals

In this chapter we describe in reasonable detail the source code of the project. The discussion will tipically hover at a structural level, depicting the design choices and code organization. This part will be most interesting for those with the intent of maintaining or modifying the work, or to study ARM bare metal development.

The size of the project is comparatively small, only reaching about 5000 lines of code. The real weight of this work does not lie in the actual software that was written but in the idea and study of the environment, pioneering the possibility of developing a proof-of-concept OS on real hardware instead of an emulator.

7.1 Design Principles and Overall Structure

Besides creating a convenient abstraction layer, the whole code base is written with the goal of being an understandable example of bare metal development. Particular care is taken in making sure that every function is readable and understandable with a single glance even out of context and in using descriptive, self-explanatory names. Where deemed necessary, comments help to further clarify what is happening.

Source files can be grouped in three main categories. First, the core of the abstraction layer is comprised basically of the assembler entry point, the C entry point and the interrupt handling routines. Second, a small library used internally to access hardware peripherals; logging routines, microSD card reading and writing, timer management. The third category contains the modules of the emulated devices like tapes and printers, leaning on the previous utilities to create the illusion of physical peripherals.

7.1.1 Implementation Language

The choice of language is severely limited by the bare environment and fell unsurprisingly on C and Assembly. Such basic programming languages contribute to the overall simplicity, as there are no particular patterns or constructs used beside raw memory management.

The Assembler compoment was kept to a minimum for ease of understanding; from the moment the C stack is available there is no real reason not to jump into C code (unless the goal was to exercise Assembly programming, which is not our case).

Thus, there are only three Assembly source files: `init.S` is the absolute first entry point and provides initialization for system registers, bss section and multicore functionality; `asmlib.S` contains utility functions that make heavy use of general and specific purpose registers that would have required inline Assembly instructions anyway if implemented in C; `vectors.S` declares interrupt handlers.

7.1.2 Build Tools

Contrarily to the μMPS family of emulators, this work does not use the Autotool suite of building tools (GNU Automake and Autoconf) to manage source compilation and package installation. Not having a newly created graphical interface means there are no library dependencies such as Qt, weakening the need for strict dependency check. This, together with a smaller overall codebase prompted the author to search for a simpler and more modern build tool, and the final choice fell on Scons.

Scons has the advantage of being much more flexible and easy to use when compared to older tools. Instead of leaning on a brand new (and potentially cumbersome) language to configure the build process it relies on an already existing one, well received and praised for its approachable syntax: Python.

In fact, Scons can be assimilated to a Python library for declaring build dependency trees. Its philosophy is similar to Make but brings a much cleaner syntax and flexible user control over the process.

7.1.3 Linker Script

The linker script is an essential piece when compiling for the Raspberry Pi 3. It has to specify `0x80000` as the loading address for compatibility reasons with Qemu and it ensures the initialization code is at the very beginning of the kernel image. It also allocates some memory as stack to be used by the abstraction layer interrupt routines.

7.2 Initialization

After loading all necessary components, the on-board GPU launches ARM execution at address `0x80000`. There, we can find the compiled code from the `init.S` Assembly file. The first operations are:

1. Enabling access at **EL0** and **EL1** to the internal ARM timer registers.
2. Setting a separate stack for each core for internal interrupt handling.
3. Enabling AArch64 execution state.
4. Moving the execution level to **EL1** [1].
5. Setting up interrupt handling routines.
6. Preparing execution for all cores: while the first core jumps to C code, the remaining ones are parked in a waiting loop, ready to be fired.
7. The bss section (uninitialized data) is zeroed and the first core jumps to the `bios_main` function.

From there control is passed to C, with another series of initialization routines:

1. The memory locations dedicated to device emulation and user interrupts are cleared.
2. Every real device is initialized: GPIOs, UARTs, EMMC, display.
3. Every emulated device is initialized, building on the real hardware.
4. Cores 1, 2 and 3 are unlocked from their parked state and set to run an infinite wait loop.
5. The user provided `main` function is called.

7.3 Interrupt Management

The core of the abstraction layer lies in the interrupt handling routines. We refer to the handlers predefined in the abstraction layer as internal interrupt handlers; the students should define their own handlers, from now on referred to as user defined handlers. There are 4 possible (and real) IRQ sources:

1. ARM timer
2. UART0

---

[1] The Raspberry Pi 3 starts in **EL2**, while Qemu initially runs at **EL3**

3. UART1
4. Mailboxes

The `main` function is assumed to never return; inside it the user should prepare an appropriate time slice and then start executing the first process. The time slice is set using the `setTIMER()` function. Note that `setTIMER()` does not interacts with the ARM timer directly but through an internal delay system; since the physical ARM timer interrupt is also used to ensure interrupts are fired on emulated interrupt lines sometimes a 0 μs delay is set. To avoid losing the user requested timer a simple array holds the next delay for every core and is restored when no more virtual lines are pending.

Once the time slice is over the internal interrupt handler is called. It checks for real or emulated pending interrupt lines and immediately passes control to the user defined interrupt handler.

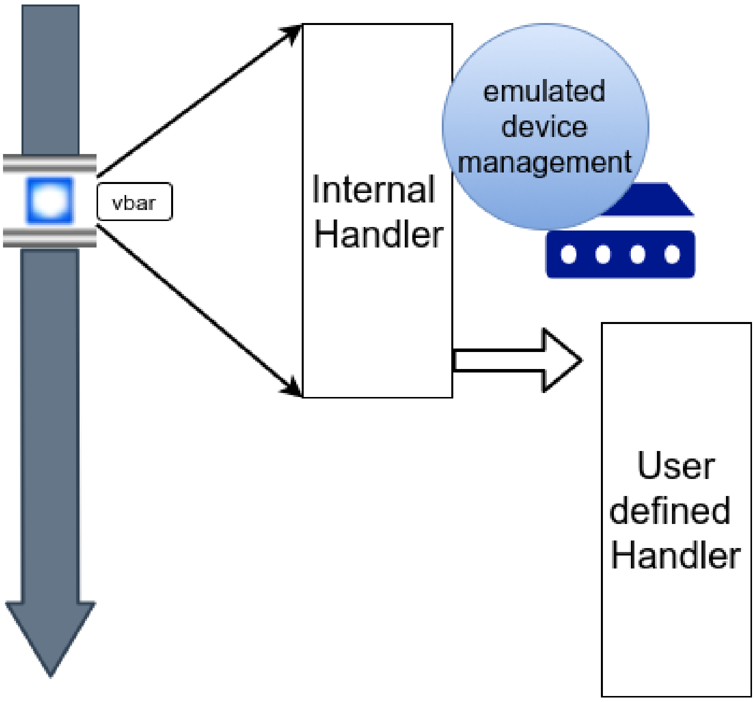


Figure 7.1: Interrupt handling schematic

Other exception handlers, like the synchronous exception handler, are even simpler, reduced to passing control to the user defined routine if present.

The abstraction layer does mainly two things before passing control to the user:

1. setting the stack pointer to a specified value for each core.
2. setting **TTBR0** to the kernel page table, giving the illusion of different page tables for different exception levels, which simplifies significantly the whole virtual memory system.

This meddling has a relatively negligible drawback: the interrupt routine is forced to corrupt two general purpose registers. The conscribed registers are **X27** and **X28**, and they should not be used for other purposes.

7.4 Emulated Devices

The idea behind emulated peripherals and their fabricated interface has already beed described in Chapter 6. Here we give a more detailed presentation about the principles under which they work.

A command to an emulated device is issued through a mailbox. For coherency reasons interrupts are however disabled at execution level **EL1** (the execution level of user defined interrupts). To maintain this precaution and still allow user code running at **EL1** to be properly served when sending a command, the special mailbox used for emulated peripherals fires a fast interrupt request (FIQ) instead. Fast interrupts are kept obfuscated to the user and managed only internally (in fact, for this single purpose). IRQs and FIQs are separated for historical reasons, so the abstraction layer can disable the former and enable the latter.

Some commands require a two-step management to more closely resemble a real peripheral. The first step is the fast interrupt, and is present for every command. For longer operations a virtual timer (different from the physical timer used through the `setTIMER()` function) is set to be executed by another FIQ interrupt after a certain amount of time.

7.4.1 Tapes and Disks

Tapes and Disks are very similar in their underlying functionality. While tapes are read-only and are accessed in a sequential fashion, disks can be written and read at random. Their content is transferred as queried through a DMA system.

The tape can be viewed as a sequential list of 4KiB blocks, each block followed by a marking 4 bytes delimiter denoting the nature of its content (last block of a file, last block of the tape or neither of those). The disk on the other hand has blocks organized with a three dimensional indexing that reproduces a head-cylinder-sector disposition.

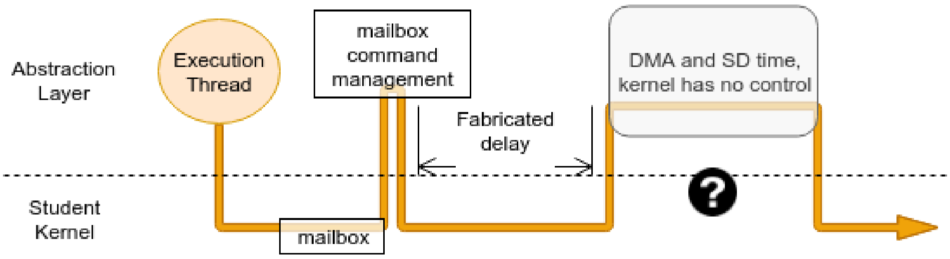


Figure 7.2: The user provided kernel might realize there are some "gray zones" in its execution time when the SD card is accessed. Proportions in the figure are exaggerated.

When a command is issued to the disk or tape device a FIQ for the first core is immediately fired. Depending on the command it might be evaded immediately or set a timer for later to emulate hardware delay; commands such as **RESET** and **ACK** finish on the spot, while operations like moving the device and reading/writing a block are postponed for a few milliseconds.

The actual content of both tapes and disks is saved as a file on the SD card. When the system boots it looks for files named `TAPEn` and `DISKn`(where n is an index from 0 to 3) and considers them as installed. The file format for tapes and disks is identical to the one used by μMPS2:

- Tapes have a 4-bytes identifier right at the beginning, then a list of 4 KiB blocks paired with yet another 4 bytes of ending marker.
- Disks have different identifiers at the start, then six 4-bytes configuration words describing the number of cylinders, heads and sectors plus the rotation speed, the seek speed and the data occupation ratio.

In fact, such files can be created and added to the microSD card using the same tool distributed with μMPS2, `umps2-mkdev`.

Thanks to the mailbox interface the illusion of a real hardware peripheral receiving orders and reporting with interrupts is almost perfect. The only real discrepancy is the Direct Memory Access function: from the student's perspective the device is writing or reading to or from the specified address; a scrupulous observer however may notice how the memory transfer operation is carried on instantaneously, and that there are comparatively big time gaps in kernel control when it happens, as depicted in figure 7.2. This is because the emulation layer takes place on the same execution thread of the kernel, forcing to share time and resources when the command is carried on [2].

---

[2] It should be noted that this inconsistency is only true for the first core.

Overall, this should not be a major concern for a novice approaching OS development.

7.4.2 Printers

Printer devices are little more than an adaptation of the HDMI display port. The Videocore IV manages almost everything: it holds a block of memory designated as framebuffer to display on screen in its own memory space and allows the ARM CPU to modify it freely. A binary font is included in the `.elf` file to print textual information and the screen is divided in four sections for different printer instances.

Printing a character merely writes it on screen, left to right and top to bottom. This is somewhat different from μMPS2, where printers more faithfully output their results to text files. To the author this approach seemed a little confusing, because it is not possible to view the output in real time and since other media devices are just files on the host filesystem they often felt like simplified tapes. Moreover, having them on the microSD filesystem would increase the need for a more immediate access since the latter needs to be mounted before reading the output file.

It would be indeed possible to flush printers on `PRINTERn` files just like tapes and disks, but doing so would significantly complicate the small FAT32 library included in the project. Disk and tapes are arbitrarily big but have a statically allocated size: printers need to grow indefinitely, demanding a function to allocate more blocks on the file system.

7.4.3 Timer Queue

The second step of some device commands is scheduled for execution after a while; since there is only one virtual timer at **EL1** to fire scheduled interrupts, this would eventually overwrite other pending timers. The physical timer is already allocated to schedule time slices, and two interrupts would not be enough anyway. To prevent this MaldOS uses a set of queue managing functions to schedule multiple virtual timers at once. The ROM function `setTIMER()` does not interact with the queue, but uses the physical timers instead.

The queue is kept ordered from the first timer that will occur to the last. When an interrupt is fired all the timers that were scheduled before the current time are popped out of the queue, and the first remaining element (if any) is scheduled again.

When a virtual timer for an emulated device is served it is likely that the corresponding interrupt line will be asserted following command completion.

For this purpose a phony "0 microseconds" **physical** timer is scheduled to ensure that the line is serviced as soon as the FIQ terminates.

Interestingly, the implementation of this module is heavily inspired by the list managing routines found in a past solution to phase 1 of the KayaOS project, covering process and semaphore queues.

7.5 Hardware Library

Modules under the `source/hal/` subdirectory contain functions to conveniently access and use hardware peripherals. They serve a purpose mainly for usage internal to the project, as the abstraction layer does not normally expose those functions (e.g. reading and writing the microSD card to emulate disk and tape devices). They could however be seen as one of the many educational examples about bare metal programming for the BCM2837 and ARM processors in general.

Notably, since disk and tape devices are "plugged" into MaldOS as files onto the microSD, a small FAT32 file system library was necessary over the EMMC controller integration. The system expects a master boot record signaling a first FAT32 partition. There, root entries (directory navigation is not supported) are listed and searched for the necessary *DISKn* and *TAPEn* files. Since both those device classes have a fixed size the library does not support file growth or shrinking, only read and write with eventual modifications. Partitions of any size permitted by FAT32 are accepted: the file allocation table is only partially loaded in main memory as a cache of cluster pointers.

7.6 Memory Management Unit

Using the MMU can be considered a higher level of complexity in its own right. One of the improvements brought by μMPS2 over μMPS was the possibility of having it disabled for the first phases of the project, since the real MIPSEL architecture had the virtual memory translation constantly active. In truth, there is little reason in practice to turn off such a feature: a microprocessor is born to run an Operating System, and every Operating System worthy of its name virtualizes memory for its processes. A smoother learning curve is quickly sacrified for the sake of efficiency.
Possibily because ARM lives in a world closer to embedded, low power, low resources, time constrained applications, its MMU devices are usually activated on demand, which is why the first phases of Kaya are still possible

on the Raspberry Pi without having to worry about page tables. When activated, however, the MMU leaves little to no room for anything else: it expects, from the very beginning of execution, the kernel to be loaded starting from a special address to use a separate page table from normal processes. This proved hard to integrate with an abstraction layer that provides a great deal of initialization code without knowing whether virtual memory will be used.

The solution is to move only the entry points of the user kernel, which are two: the interrupt address found in the **VBAR** register and the first `LDST()` function call to launch the first user process. On the interrupt table side the **initMMU()** function initializes system registers, disabling every cache and setting up for full range, 4KiB blocks page tables (using a simple identical mapping for the first translation function) and moves **VBAR** over to `0xFFFF000000000000`, making sure that **TTBR1** is used when jumping to the kernel. If the handler is reached through that jump for all intents and purposes the kernel believes to be running at the far end of the main memory. As for the first process context switch, it is a tricky situation: after enabling the MMU but before loading the process state the kernel is still using **TTBR0** because its entry point did not pass through the "rigged" **VBAR** register; therefore the TLB cache will contain entries from the kernel page table. The very last instruction before using `eret` to switch to **EL0** loads the new **TTBR0**. Even so, the `eret` instruction will fetch the new, user level page entry while still running as the kernel and cause an instruction abort for security reasons. To avoid this, one must make sure to call `LDST()` through **TTBR1**, which can be achieved by jumping to the address of `LDST` plus the most significant two bytes of the 64-bits register. In this way the kernel tables will be used until the user process is actually running.
Note that this problem should not present itself if the process' code is loaded in a different memory block. Nevertheless it is a theoretical inconsistency and should be considered.

The only other alternative would be to pretend to load the entire binary at that address by specifying it in the linker script, and then setting up the MMU at boot, before even jumping to C code. For reasons already listed, this was not acceptable.

7.6.1 Design Choices

While constructing the virtual memory environment the author came across design crossroads where the decision was not driven by technical considerations, but rather educational ones. For example, the abstraction layer can let students avoid ASID management by simply flushing the TLB cache every time there is a context switch; if not, failure to setup proper process IDs will result in abort exceptions. Ultimately the latter approach was taken, leaving to the user to write a new ASID to **TTBR0** every time a new process is launched. Again, user-defined handlers are written to special memory locations as addresses to jump to; if the MMU is active said addresses must be changed in order to have the CPU refer to **TTBR1** instead of **TTBR0** when calling those functions. This nuisance can be handled automatically by the abstraction layer (and indeed it is in the final implementation), but one could instruct students to change the handler's address after enabling the MMU.

Those choices are subjective and could be easily changed.

7.7 Exception Levels and Virtualization

The ARMv8 specifications counts a great deal of features for virtualization purposes. One of the four exception levels, **EL2**, is defined as the hypervisor level and has responsibilities revolving around multiple guest OS management, abstraction of system registers, nested virtualization of page tables (virtualized virtual memory, so to speak) and device emulation. Some functions at **EL3** are connected to those concepts, but it is a level mainly focused on TrustZone technology and security.

Many of the ideas enumerated in this description are at the base of this project, so one would naturally expect said features to be exploited to some extent. Although the possibility is not ruled out by the author, up until the time of writing the route of **EL2** has been deemed unattractive for the development of MaldOS.

After a thorough analysis, this should not come as a surprise. The hypervisor component that ARM refers to is for a complete Operating System to be taken advantage of in the context of multiple virtual guest OS hosting; while intuitively an abstraction layer emulating multiple device classes and a simpler develoment environment has many common points to such a scenario, it is also significantly different. Therefore, many configuration options offered by the hypervisor that could bring improvements to this work ultimately fail to fit into it. The following sections portray the most notable ones.

Trap General Exceptions

EL2 configuration is controlled by **HCR_EL2**, a 64-bits long register with many choices in the matter of virtualization. The number of available fields actually vary depending on the implemeted version of the specification: the last ARMv8.3 uses 44 bits, but the Cortex A-53 processor was created when version 8.0 was being defined, thus having only 33 usable bits. Out of those fields the most prominent one is **TGE**, short for Trap General Exceptions. When the **TGE** bit is set all exceptions that would normally be executed at **EL1** are routed to **EL2** instead.

This option has the potential to greatly simplify the actuation of an abstraction layer. Executing at the same level of the assisted code weakens it, forcing some tradeoffs with emulation fidelity: for example, having two separate exception levels could mean that the abstraction layer never needs to worry about virtual memory as MMU settings are banked between levels, avoiding the register juggling uncovered in section 7.6. Unfortunately this result is not achievable, as an active **TGE** apparently means **EL1** is not accessible altogether. `eret` instructions that would lower the exception level to 1 are treated as illegal exception returns and maintain **EL2** privilege instead (carrying on execution from where the return should have started). If the abstraction layer is unable to return to **EL1** when leaving the user code in control there is no point in working at **EL2** in the first place.

Considering this behaviour it is unclear what purpose the **TGE** bit is supposed to serve, especially since another field (**E2H**) seems to lead to the exact same scenario where the host Operating System lives in **EL2** instead of **EL1** [3].

AArch32

Section 2.3 hints at the 32-bits compatibility interface of ARMv8, an essential property of all 64-bits architectures to allow execution of legacy software. The **RW** bit of the **HCR_EL2** register controls the execution state of lower exception levels. In the scope of an educational project the 32-bits architecture could be seen as more historically interesting and appropriate as a first approach to the field; for this work however more value was given to clean structure and modern perspective, and AArch64 was chosen.

---

[3] It should be noted that the **E2H** is only available from version 8.1 of the specification.

Trap System Registers Accesses

Among the different variations that **HCR_EL2** allows for there is the option to trap several memory accesses, both read and write, or certain instructions (like TLB maintainance) to **EL2** with the intent of hijacking the result. This possibility was not found to be useful in the context of this work.

Partial Exception Routing and Virtual Exceptions

In opposition to the **TGE** complete superseding of **EL1** by **EL2** there are also separate fields in **HCR_EL2** that allow for single exception classes to be routed to a different level (**AMO**, **IMO**, **FMO** fields, corresponding to serrors, interrupts and fast interrupts respectively).
Those fields share some of the advantages listed about general exceptions routing while still retaining the possibility to return at **EL1**. Moreover, when the corresponding routing bit is enabled another field permits the hypervisor to set a virtual exception that lower levels will see pending and that could be used to more properly emulate μMPS2 devices.

Unfortunately this situation is not perfect either. First, not all interrupt classes are subject to this configuration, leaving out synchronous exceptions caused by instructions like `SVC`, used by system calls. While it is still possible to implement the `SYSCALL()` function as an `HVC` call (routed to the hypervisor at **EL2**), the difference in exception management is somewhat disorienting. As for virtual interrupts, they still have to be taken and handled directly by **EL1**. By committing to this choice the abstraction layer would find itself in an uncomfortable position with overhead handlers for both **EL1** and **EL2** instead of routing everything to **EL2** and then gracefully returning to **EL1** for user-defined routines, adding the complexity of the hypervisor without solving completely **EL1** related problems (i.e. MMU register switching).

Overall, these options seem to be tailored around different objectives, so they all fail to fix this project's main flaws or make other problems emerge.

Chapter 8

Student's Perspective

This chapter describes what MaldOS exposes to a student to develop a toy OS: provided ROM functions and mapped memory addresses that were decided and defined taking inspiration from Kaya and μMPS2. Barring some minor details, the illusion of touching an ARM system directly should be perfect for a novice. That being said, "jailbreaking" out of MaldOS is fairly easy; the abstraction layer relies on a good willed user to be used at most efficiency.
As a first, general rule, direct interaction with system registers is unnecessary (if not implicitly forbidden). Students can use a series of functions exposed by the abstraction layer to perform configuration and management.

8.1 ROM Functions

Strictly speaking, there is no Read Only Memory on the BCM2837 (or rather, not a freely accessible one). This section is dubbed "ROM Functions" in reference to the precompiled routines provided as object files by μUMPS2 and similar emulators. Those functions include:

HALT() : terminates the calling core execution, stopping in an endless loop.

PANIC() : similar to **HALT()**, prints a kernel panic message before stopping.

WAIT() : pauses execution until an interrupt occurs. Normally this is achieved by the `wfi` Assembly instruction; under MaldOS however multiple interrupts (FIQ) are fired without the user knowing, so the single instruction would be unreliable. The **WAIT()** function works because it checks on a spinlock opened by the abstraction layer when a real interrupt is reached.

LDST(void *addr) : Load State function call. Reads the address to load a system state: general purpose and system registers to resume execution of a saved process. Used by the kernel to context switch to a process.

STST(void *addr) : Store State function call. Writes the current execution state to the specified address; used by the kernel to save the interrupted process and resume it later. The saved state format is a structure (see 8.1).

getCORE() : returns the id of the core currently executing the function. Its implementation simply reads the **MPIDR** register.

getTOD() : get Time of Day function. Returns the number of microseconds elapsed since reset, reading it from the internal ARM timer registers. Since the architecture is 64 bit, the returned value is a complete 64 bit `unsigned long`, as opposed to the split **getTODLO()** and **getTODHI()** found in μMPS2's ROM.

setTIMER(unsigned long us) : a routine to set the next timer interrupt to be fired `us` microseconds in the future.

initMMU(unsigned long *table) : function to initialize the memory management unit. Its argument is the initial page table to set to both **TTBR0** and **TTBR1**; the kernel page table, so to speak.

isMMUACTIVE() : function returning either 1 or 0, depending whether the MMU for the current core has been activated or not.

SYSCALL(unsigned int, unsigned int, unsigned int, unsigned int) : function used to invoke a system call interrupt. In truth, nothing more than the `svc #0` Assembler directive.

These functions are linked in the `.elf` of the abstraction layer, thus can be simply referenced in the code after proper inclusion. When compiling separately a standalone executable to be loaded in RAM for the most advanced phases of Kaya this is not contemplated, as it would link the whole HAL to every binary. In this case it is left to the student to implement ROM functions: in reality, only **SYSCALL** is needed for a normal process, and it has a one line implementation.

Listing 8.1: Process state format

```
typedef struct _state_t {
    uint64_t general_purpose_registers[29];
    uint64_t frame_pointer;
    uint64_t link_register;
    uint64_t stack_pointer;
    uint64_t exception_link_register;
    uint64_t TTBR0;
    uint32_t status_register;
} state_t;
```

8.2 System Initialization

The student's kernel entry point is the `main` function. The first task of the Operating System should be populating the custom interrupt routines: contrarily to μMPS2 interrupt handlers are not specified with a full process state, but condensed into just the function and the stack pointer. The function pointer must be loaded into the `_HANDLER` locations, while the stack pointer is higher up and there is one for each core. There are no separate stack pointer for different exceptions: if an exception is fired from the kernel handler, the stack pointer is kept the same. An example of initialization sequence could be

```
*((uint64_t *)INTERRUPT_HANDLER)     = (uint64_t)&interrupt;
*((uint64_t *)SYNCHRONOUS_HANDLER)   = (uint64_t)&synchronous;
*((uint64_t *)KERNEL_CORE0_SP)       = (uint64_t)0x1000000;
*((uint64_t *)KERNEL_CORE1_SP)       = (uint64_t)0x1002000;
*((uint64_t *)KERNEL_CORE2_SP)       = (uint64_t)0x1004000;
*((uint64_t *)KERNEL_CORE3_SP)       = (uint64_t)0x1006000;
```

After this, the first process should be created and loaded with a proper time slice set as timer interrupt. The memory address from where processes can use stack starts where the kernel ends; said value is provided for convenience by the `_kernel_memory_end`, defined in the linker script.

8.3 Exceptions

By default, exceptions are handled at **EL1**. Control is given to the specified handler with normal interrupts disabled but fast IRQs enabled; the user is not supposed to change this settings, and doing so might result in untested behaviour. Thus, nested interrupts are not present. Interrupt exceptions are fired for the student's kernel if a physical device requires it (UART) or if an emulated device is asserting its line; while an interrupt line (real or emulated) is pending and not masked, the interrupt handler will be called repeatedly until it is properly acknowledged. Since interrupt lines for emulated devices are nothing more than RAM locations it is possible to forcibly deassert them by simply writing 0 instead of using the provided **ACK** command. This should not be done and can lead to unpredictable outcomes.

Synchronous exceptions (namely, system calls) are very simple; on `SVC` instruction the provided handler is called, and parameters can be passed using general purpose registers. In fact, to properly return a value from a system call the state saved in the corresponding *old area* must be loaded with a modified `X0` register.

Abort exceptions work similarly. Note however that unlike μMPS2, where TLB faults were part of the normal workflow, MaldOS does not contemplate recoverable aborts. The abort handler is however generic, and the possibility is not ruled out.

Whenever an exception is fired, the handler is loaded using the specified stack pointer for **EL1** and the executing core; if an exception condition happens while at **EL1** (e.g. an abort or a system call, even if it does not make sense to invoke one when at kernel level) the stack pointer is kept the same and grows from where it was left.

8.4 Multicore

MaldOS approach to multicore functionality is very simple. Exception handlers are shared among all cores, leaving to the user the responsibility to separate routines based on the executing actor. Old areas where states are saved and stack pointers on the other hand are defined in a per-core basis. Every core can have its exceptions and handle them accordingly.

After reset and initialization cores 1, 2 and 3 are left stuck in a wait loop and should be moved from there using core mailboxes to trigger interrupt handlers. Inter Processor Interrupts (IPI) are completely transparent and work as defined by the ARMv8 specification. The only exception is the first mailbox of the first core, reserved for emulated peripheral management. Data written to this mailbox is interpreted as a pointer to a device registers-holding structure and is not controlled by the student provided kernel. Other mailboxes on the other hand will immediately cause an interrupt in the corresponding core. Mailbox interrupts must be cleared by writing the same data in the write-clear register.

8.5 Devices

The student can use two classes of devices on MaldOS: emulated or real peripherals.

8.5.1 Real Peripherals

The only real peripherals accessible from the abstraction layer are the two built in serial interfaces, UART0 and UART1. Section 5.3.3 describes in greater detail the register configuration for said devices. Although clearly more complex and convoluted when compared to the clean interface for μMPS2 emulated devices, it is believed the handful of registers needed to use UART1 and UART0 is still well within reach for a computer science undergraduate student. Similarly to flying a plane, the hardest part is taking off, and device initialization is already prepared by the abstraction layer when the user kernel gains control.
Interrupt handling is self contained in specific device registers; once enabled the exception routine will behave as expected, continously being called until the specific acknowledgement is delivered.

8.5.2 Emulated Peripherals

The emulated peripherals interface is kept almost identical to the μMPS2 approach, and is detailed in sections 6 and 7.4. As mentioned before the commands are passed to the emulation layer with a memory structure shared through the first core's first mailbox (unavailable for other purposes). The memory address must be 16-bytes aligned, as the 4 least significant bits are used to index the device the command is meant for.

8.6 Memory Management Unit

The MMU operation is described in depth in chapter 4. It is a complex device, unsuitable for undergradates. This complexity is, in the author's

opinion, very much intrinsic in the nature of virtual memory management, and little can be done to ease it; even if possible it would not make sense to further simplify it as it would lose most of its educational value. Therefore, the MMU interface has been left mostly untouched by the abstraction layer. The sole real interference is copying **TTBR1** into **TTBR0** when handling an exception to make sure the memory can be accessed without trouble even at **EL1**. If this was not the case the kernel could end up receiving entry results of the process that was paused.
This passage is made essential by the fact that the MMU should be easily turned on and off without changing too much of the original code. Real OSes like Linux simply enable virtual addresses from the beginning and load the kernel code at the far end of memory, where **TTBR1** is referenced for page tables. Moreover, Qemu does not allow for loading addresses other than the default `0x80000`.

Other than that, the MMU is entirely managed by the user, even if registers should not be directly accessed and instead modified only through `initMMU()` and `isMMUACTIVE()` ROM functions. Just like μMPS2, how far one should go into implementing virtual memory policies is up to the professor or course using this tool. For example:

- Page tables support a Dirty Bit Modifier field; when an entry is found (either in the TLB cache or after a table walk) that has the DBM bit cleared an MMU fault is raised expecting the software to handle the situation and re-try the same memory request. This is usually used to only load memory blocks in memory when they are accessed, thus allocating less physical memory for each process.
- To avoid confusion where multiple processes find the same cached entries in the TLB each one of them should be paired with an ASID copied to the **TTBR0** register. If page entries are then marked as non-global using the nG bit a TLB search hits only if the current ASID is equal to the one saved with the entry.
- In an exaggerated simplification the TLB cache can be flushed every time a context switch occurs. In this situation no ASID is needed as the stale entries are just wiped before they can cause issues.

Note that the memory management unit configuration is set on a per-core basis: there can be cores where virtual memory is enabled and active, and others where normal physical memory is used.

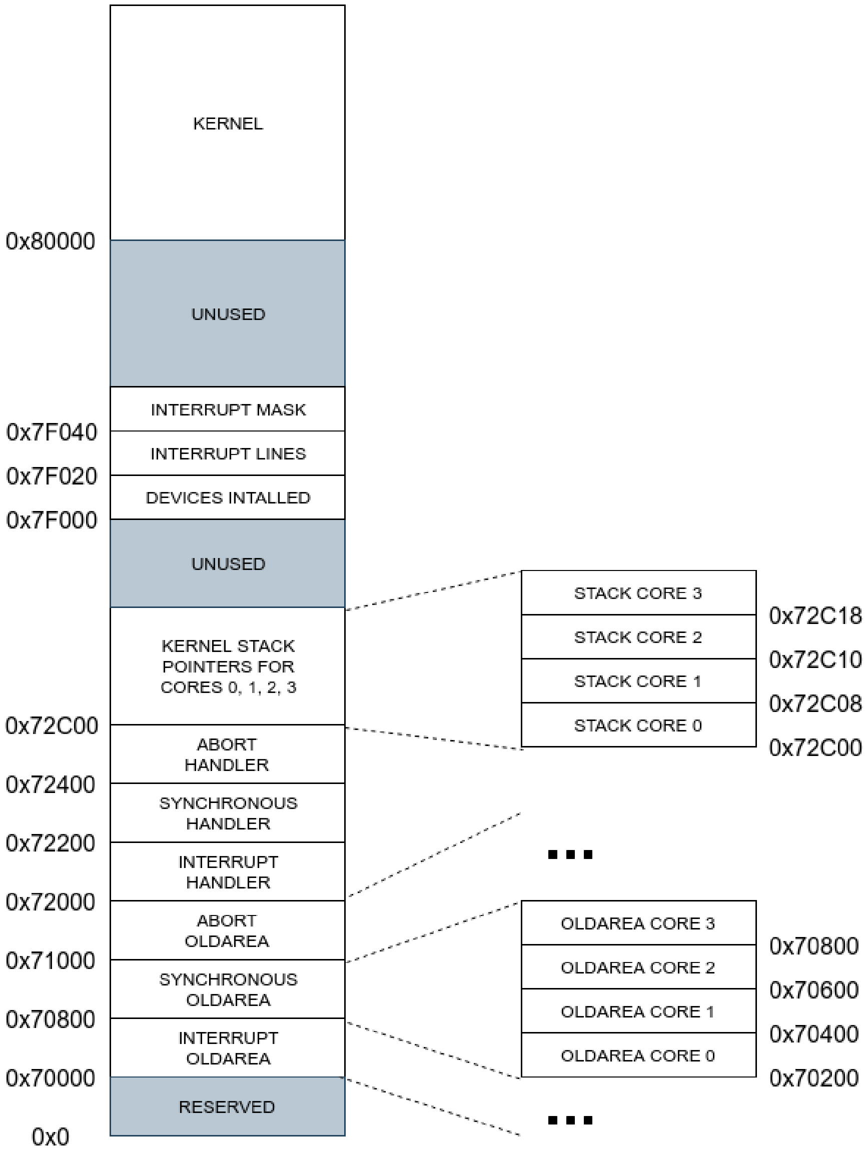

KERNEL
0x80000
UNUSED
INTERRUPT MASK
0x7F040
INTERRUPT LINES
0x7F020
DEVICES INTALLED
0x7F000
UNUSED
KERNEL STACK POINTERS FOR CORES 0, 1, 2, 3
0x72C00
ABORT HANDLER
0x72400
SYNCHRONOUS HANDLER
0x72200
INTERRUPT HANDLER
0x72000
ABORT OLDAREA
0x71000
SYNCHRONOUS OLDAREA
0x70800
INTERRUPT OLDAREA
0x70000
RESERVED
0x0
STACK CORE 3
0x72C18
STACK CORE 2
0x72C10
STACK CORE 1
0x72C08
STACK CORE 0
0x72C00
...
OLDAREA CORE 3
0x70800
OLDAREA CORE 2
0x70600
OLDAREA CORE 1
0x70400
OLDAREA CORE 0
0x70200
...

Figure 8.1: Memory locations for interrupt old areas, handlers and stack pointers

Chapter 9

Usage and Debugging

9.1 Final Result

The final result of this work consists, from the user perspective, solely of two files: `hal.elf` and `hal.ld`. The first is the the hardware abstraction layer compiled for an ARM64 target, containing system initialization and emulated devices management; the second is its linker script, to be used to link an application to the hal.

The hal performs all the necessary routines and then calls a `main` function. There is a weak-defined `main` included with the hal that just echoes every character received on UART0 that is supposed to be overwritten by a new `main`. symbol. From there, the user provided code is expected to write specific memory addresses to define new exception handlers and control emulated devices.

One of the objectives of this work was to avoid creating ad hoc software and relying as much as possible on widespread tools. Because of this, there is no custom package like the μMPS2 emulator to install; instead the user needs a proper cross compile toolchain for ARM64 (or an ARM64 device, like the Raspberry Pi itself) and eventually Qemu.

9.2 Compiling

MaldOS provides a single `.elf` file containing initialization code, abstraction layer and some functions; given that, there are several ways to compile it into a project. Here an example is shown, where a single module contaning the `main` function is linked to the HAL.

```
$ aarch64-linux-gnu-gcc -o main.o -c -Wall -ffreestanding \\
    -nostdlib -nostartfiles -O0 -g -march=armv8-a \\
    -mtune=cortex-a53 -fPIE -ffixed-x27 -ffixed-x28 \\
    -Iinclude/app main.c
$ aarch64-linux-gnu-ld -o app.elf -r main.o$
```

Each command option has its own meaning:

-Wall requires to print all warnings during compilation. Not Strictly necessary, but good practice.

-ffreestanding assert that compilation targets a freestanding environment, one in which the standard library may not exist and program startup may not necessarily be at main.

-nostdlib instructs not to use the standard system startup files or libraries when linking.

-nostartfiles commands not to use the standard system startup files when linking.

-O0 is the optimization level; here it is 0, with no optimization at all. One might specify other levels, but in a project where the focus is on the learning experience and not in performance compiler meddling should be kept to a minimum.

-g includes debugging symbols into the executable.

-march=aromv8-a specifies the target architecture.

-mtune=cortex-a53 specifies the target processor.

-fPIE generates position independent code; it is necessary for when the executable will be linked with the abstraction layer in the next step.

-ffixed-x27 -ffixed-x28 instucts the Assembler generator to avoid using general purpose registers **X27** and **X28**. As described in section 7.3 those are corrupted during context switch, and thus are not available.

Unless otherwise specified, all options are strictly necessary for a working result. The absence of some flags might not create immediate issues but are nevertheless to be considered a mistake (e.g. registers **X27** and **X28** are rarely used).

Given a compiled elf with the user's code called `app.elf` and assuming to use `aarch64-elf-gcc` as a cross compiler, the process to create a kernel image would be

```
aarch64-elf-ld -nostdlib -nostartfiles -Thal.ld \
    -ooutput.elf hal.elf app.elf
aarch64-elf-objcopy output.elf -O binary kernel8.img
```

The resulting binary can then be placed on a microSD card and run on a Raspberry Pi 3 or on Qemu.

9.3 Qemu

Since version 2.12 Qemu supports a Raspberry Pi 3 emulated machine. The official version for the Linux distro of choice may be less recent, in which case the user needs to compile the package from source. Particular care was taken in assuring the same code runs with no discernible difference on the emulator and the device, which was not a difficult task. Usually, in the rare situations where virtual and real boards differ in their behaviour the real hardware is in the right (as one would expect). Some examples found along the way are:

- Uninitialized memory location will inevitably contain null values if running under Qemu; the real world RAM is not so clement, and will live up to the tale of having its content randomized after a reset.
- The MMU memory configuration includes distinguishing between device and normal memory: while the latter ban be subject to caching to increase performance, the former will not be optimized. Device memory is meant for memory mapped areas that are connected to peripherals, as their volatile nature would mix with caching for incoherent results. Failing to set the device area as device memory will be forgiven on Qemu as there are no real peripherals; instead, the Raspberry Pi board will most likely not behave as expected.
- Similarly to the previous example, while to operate both serial interfaces on real hardware one must switch GPIO configuration as they both use the same pins, Qemu will allow the two consoles to print and read as if on separate lines without swapping between hardware setup.
- Qemu is whimsical about the memory address where to load the kernel image. The emulator's boot sequence is different from the real device as the `kernel8.img` file is not read from the microSD card but passed from the command line. Qemu invariably starts the execution by jumping at `0x80000`; if that is not the same address referenced by the linker script the kernel will fail to run.

Note that, beside those immediate idiosyncrasy, emulator and hardware can behave differently while still being both correct. Quoting a Qemu developer,

> *"QEMU only promises to run architecturally correct code the way the architecture says it should run. It doesn't guarantee to run incorrect code in the same way the hardware happens to run it. QEMU aims to be an architecturally valid implementation, not an*

> *implementation that matches the real hardware CPU. (That is, we are free to behave differently for things which the architecture manual defines as IMPLEMENTATION DEFINED or UNPREDICTABLE.)"*

One brilliant example of this situation can be found in TLB behaviour. A cached page table entry can live inside the TLB for an arbitrary amount of time, extremities included: it is architecturally correct to **both** immediately discard it or keep it forever. In an early test of virtual memory translation, a single process was made to run at **EL0** after being launched by the kernel from **EL1**. Kernel and process code were part of the same executable, and thus shared the same memory block. Two page tables were set up for the kernel and the process; they had to be different to allow correct execution permissions: the processes' entries had the **AP** field set to no limit, while the kernel ones restricted **EL0** access [1]. At first no ASID was used, and the same binary image was running fine on Qemu but not on real hardware. The reason for the hardware fault was found in the presence of stale entries in the TLB: initially kernel entries were cached; when the process was launched it tried to fetch the same memory block as the kernel, hitting a cached version that forbade **EL0** access, thus resulting in an instruction abort. On the other hand, Qemu was running fine because it does not faithfully emulate the TLB. When any **TTBR** register changes the entire TLB cache is flushed behind the scenes, immediately deleting any stale entry. The code was wrong, but it happened to run anyway on the emulator.

To fix the example ASIDs were used, setting a different ID for kernel and process tables. This time however the parts were inverted: the real Raspberry Pi 3 board started working while Qemu was constantly stuck into abort exceptions. Again, the bug was due to the same TLB difference: between swapping **TTBR0** with **EL0** process tables and actually switching the context there were some instructions at **EL1**. For those, fetching a virtual page entry yielded a TLB miss because the ASID had changed, so a new entry from the process' table was returned, leading to yet another permission fault. On the Raspberry Pi 3 the same code still ran fine, probably because the change in **TTBR0** had yet to be registered before the context switch and the kernel was still hitting the correct TLB cached entries. The final solution was to run the kernel code at address `0xFFFF000000000000` to make sure **TTBR1** was always selected.

---

[1] Note that, by default, a memory block that is not restricted for **EL0** write access **cannot** be executed by higher exception levels due to obvious security concerns.

Qemu requires a kernel image and a microSD card image to be passed as command line arguments. An example command to run the emulator is:

```
qemu-system-aarch64 -M raspi3 -kernel kernel8.img \
    -drive file=drive.dd,if=sd,format=raw \
    -serial vc -serial vc
```

Where the command line options have the following meaning:

-M raspi3 specifies the machine to emulate.

-kernel kernel8.img specifies the kernel image to run.

-drive file=drive.dd,if=sd,format=raw attaches the microSD card, here using an image file. Note that a real device can be used in the same way, for example using `file=/dev/mmcblk0`, allowing to run both on the board and the emulator with the same exact drive.

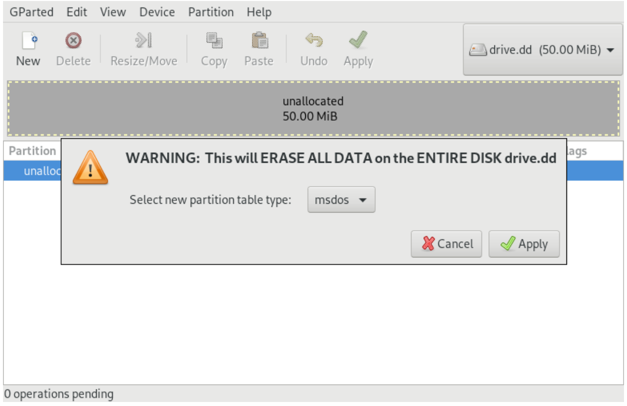
-serial vc each serial option accounts for a UART interface (UART0 and UART1, in this order). `vc` stands for "virtual console" and will open a tab in the Qemu window. Another possible value is `stdio`, which will conveniently pipe the serial output of the chosen interface on the shell (obviously available for only one of the two UARTs).

9.3.1 Create a Disk Image

As is indicated in previous command examples Qemu accepts a block device to be plugged into the `raspi3` machine as if by a microSD card slot. The specified path can either be real, as in a physical disk installed in the host system, or fake, in the form of a binary file representing an abstracted file system.
In the first scenario it is sufficent to pass the device file path through the `file` argument of the `-device` command line option: for all intents and purposes the emulated machine will communicate with the drive using an EMMC controller. Unless specific rules are present into `/etc/fstab` that allow normal users to access disk drives, root permissions will be necessary for this route; as always, special care should be taken lest risking corruption of a vital drive due to a bug in the emulated OS. For this and other convenience reasons, using a virtual disk file is usually preferable. A virtual disk file can be created by various means; here is brought an example using the Gparted tool.

The file itself must be already present in the file system before running Gparted. A conventient way to create it is using the `dd` utility:

```
$ dd if=/dev/zero of=drive.dd bs=1M count=50
```



WARNING: This will ERASE ALL DATA on the ENTIRE DISK drive.dd

Select new partition table type: msdos

0 operations pending

Figure 9.1: How to create a new partition table in Gparted.

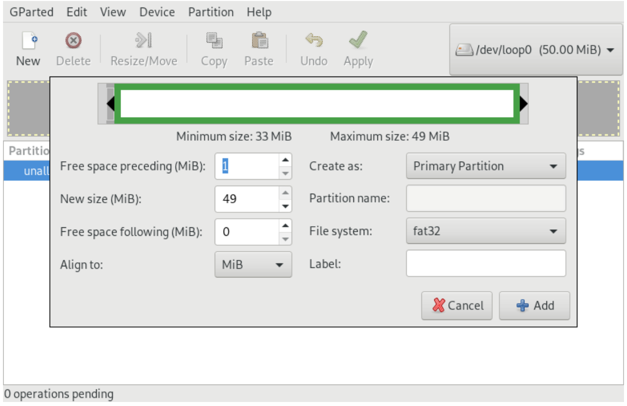
The following procedure is slightly complicated by the fact that the final result is not a simple FAT32 partition encoded in an image file, but an entire disk complete with partition table. A suitable way to handle such a requirement consists in pointing Gparted to a loopback device mapped to the file; this is achieved with the command:

```
$ sudo losetup -fP drive.dd
$ losetup -j drive.dd
```

The second of the two commands prints the loopback device that was mapped to `drive.dd`. After this, the disk configuration can be accessed in Gparted by running `sudo gparted /dev/loop0` (provided that `loop0` was the mapped block device). The first step is to create a new partition table via *Device > Create Partition Table*; the table type `msdos` should be selected and written to the file, as in figure 9.1.

After this the actual partitions can be written. Despite the microSD emulation, Qemu does not follow the same boot process as a real Raspberry Pi 3; therefore, the disposition of partitions on disk and the presence of an actual `kernel8.img` binary on the first FAT32 partition is irrelevant. Nonetheless, MaldOS mimics the hardware restrictions and expects a single FAT32 partition. This configuration step is depicted in figure 9.2.

Once everything is set the image file can be mounted and accessed again through a loopback device. Note that, let `/dev/loop0` be the associated loopback device, the actual partition to be mounted is different, probably in the form `/dev/loop0p1`. All files copied into the image will be available for the emulated machine; tape and disk devices are added to MaldOS this


GParted
Edit
View
Device
Partition
Help
New
Delete
Resize/Move
Copy
Paste
Undo
Apply
/dev/loop0 (50.00 MiB)
Partitio
unall
Cancel
Add
0 operations pending

Minimum size: 33 MiB
Maximum size: 49 MiB
Free space preceding (MiB): 1
New size (MiB): 49
Free space following (MiB): 0
Align to: MiB
Create as: Primary Partition
Partition name:
File system: fat32
Label:

Figure 9.2: How to create a new FAT32 partition in Gparted.

way, as files name `TAPEn` and `DISKn` (with n ranging from 0 to 3), respectively.

Another, perhaps faster method for achieving a suitable disk image consists in simply cloning a real disk (possibly a Raspberry Pi-ready microSD card) to a binary file with the `dd` tool, similarly to how the initial `drive.dd` was created:

```
$ dd if=/dev/mmcblk0 of=drive.dd bs=1M
```

Where `/dev/mmcblk0` is the disk device to be cloned.

9.4 Debugging

The debug of the compiled kernel can be carried over Qemu with GDB. Using the `-gdb tcp:1234` parameter Qemu opens a debugging tcp port for a GDB client to connect to (another port can be specified). The `-s` command line flag brings the same result in a shorter format, and by adding `-S` as well the emulator will not start the execution, allowing the developer to connect.

Once the emulator is ready, a GDB client can connect to it. A client for ARM64 should be present within the toolchain used to compile the kernel. A simple command line client may attach using the following commands (assuming the `aarch64-elf-gcc` toolchain is installed)

```
$ aarch64-elf-gdb
(gdb) file output.elf
(gdb) target remote localhost:1234
```

This GDB initial configuration can be added to a `.gdbinit` file for convenience to get automatically executed every time the debugger is run.

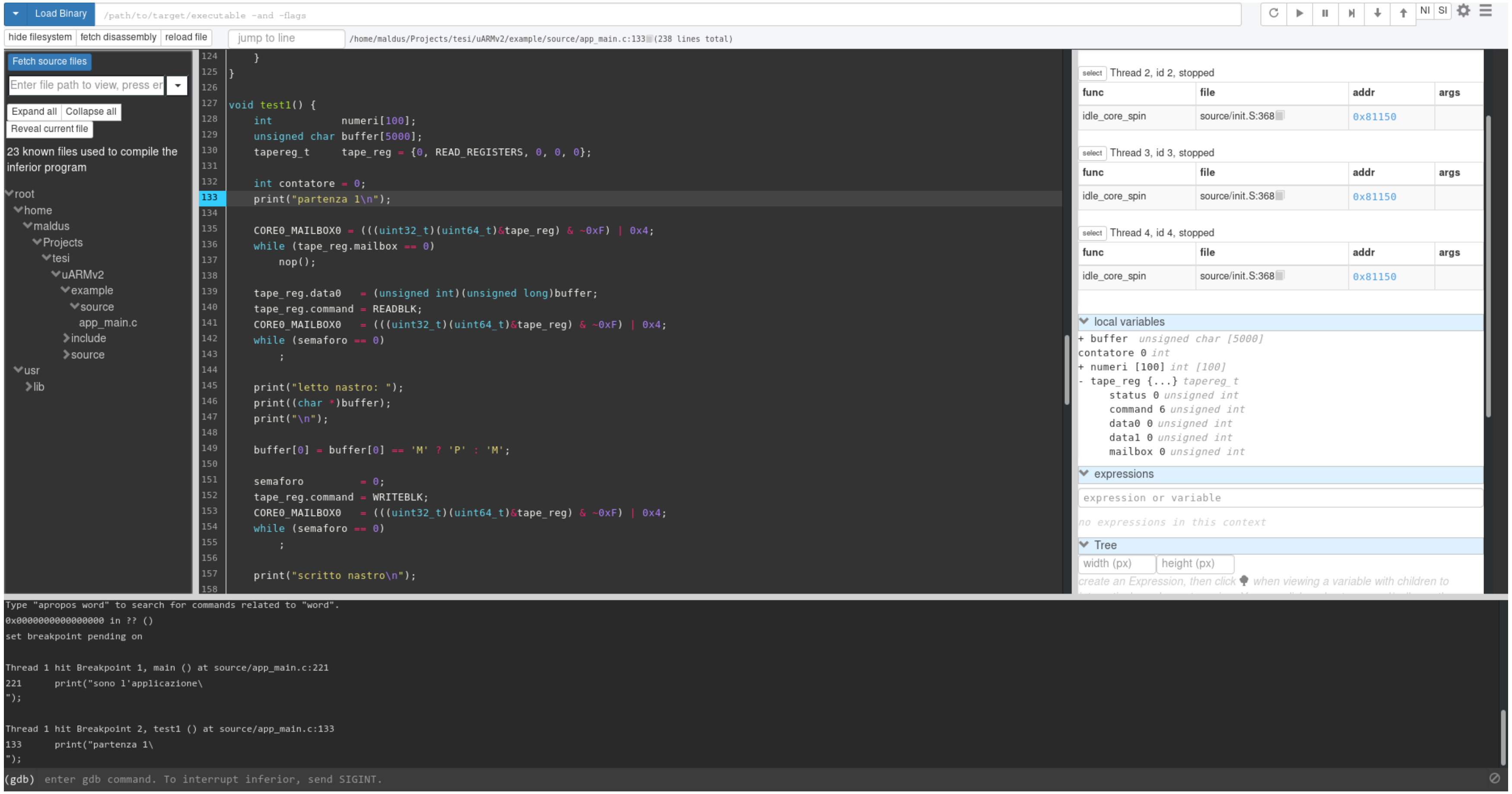
Emulators like μMPS2 have the prominent advantage of a specifically designed running and debugging interface; nonetheless, a GDB server is a complete and advanced debugging suite. The command line debugger may seem a scarce alternative, but there are plenty of richer options; the author recommends `gdbgui`, a browser-based Python GDB client. Gdbgui can be installed via `pip` or as an official package, depending on the Linux distribution. It must be launched with the `--gdb` (or `-g`) command line option to specify a proper GDB client (i.e. the one found within the ARM64 toochain); it acts as a web server reachable at the default port `5000` with any browser, and provides an intuitive interface fitted with step-by-step debugging, memory inspection, threaded view and so on. A `.gdbinit` becomes especially useful when using Gdbgui as it is read even when the debugger is executed through the Python wrapper.

9.4.1 Memory Management Unit

Special care should be taken when debugging with the MMU enabled. To accomodate virtual memory with ease of usage, the kernel image is loaded in physical address space and virtualized only later. As has already been explained, this is a somewhat uncharacteristic behaviour for a kernel, so support is not perfect. When MMU is enabled the entry point for interrupt handlers in the form of the **VBAR** register is shifted at address `0xFFFF000000000000`; the rest of the kernel only uses relative references, so everything works. However if the user is debugging the kernel he or she will become unable to break execution on kernel code; logically, having moved all addresses 16 ExbiBytes forward GDB will have a hard time finding symbols. To fix references again one must load the elf symbol table with the same offset, using the following GDB command:

```
(gdb) add-symbol-file output.elf 0xffff000000000000
```

Note that this directive too can be added to `.gdbinit`. There is no obvious drawback to loading the symbol table at two separate offsets (the first one being `0x0`); this way breakpoints will work regardless of the MMU state.


Load Binary
/path/to/target/executable -and -flags
hide filesystem
fetch disassembly
reload file
jump to line
NI
SI
Fetch source files
Enter file path to view, press er
Expand all
Collapse all
Reveal current file
23 known files used to compile the inferior program
root
home
maldus
Projects
tesi
uARMv2
example
source
app_main.c
include
source
usr
lib

/home/maldus/Projects/tesi/uARMv2/example/source/app_main.c:133 (238 lines total)

```
    }
}

void test1() {
    int            numeri[100];
    unsigned char buffer[5000];
    tapereg_t     tape_reg = {0, READ_REGISTERS, 0, 0, 0};

    int contatore = 0;
    print("partenza 1\n");

    CORE0_MAILBOX0 = (((uint32_t)(uint64_t)&tape_reg) & ~0xF) | 0x4;
    while (tape_reg.mailbox == 0)
        nop();

    tape_reg.data0   = (unsigned int)(unsigned long)buffer;
    tape_reg.command = READBLK;
    CORE0_MAILBOX0   = (((uint32_t)(uint64_t)&tape_reg) & ~0xF) | 0x4;
    while (semaforo == 0)
        ;

    print("letto nastro: ");
    print((char *)buffer);
    print("\n");

    buffer[0] = buffer[0] == 'M' ? 'P' : 'M';

    semaforo         = 0;
    tape_reg.command = WRITEBLK;
    CORE0_MAILBOX0   = (((uint32_t)(uint64_t)&tape_reg) & ~0xF) | 0x4;
    while (semaforo == 0)
        ;

    print("scritto nastro\n");
```

select Thread 2, id 2, stopped

func	file	addr	args
idle_core_spin	source/init.S:368	0x81150	

select Thread 3, id 3, stopped

func	file	addr	args
idle_core_spin	source/init.S:368	0x81150	

select Thread 4, id 4, stopped

func	file	addr	args
idle_core_spin	source/init.S:368	0x81150	

local variables

```
+ buffer  unsigned char [5000]
contatore 0 int
+ numeri [100] int [100]
- tape_reg {...} tapereg_t
    status 0 unsigned int
    command 6 unsigned int
    data0 0 unsigned int
    data1 0 unsigned int
    mailbox 0 unsigned int
```

expressions

expression or variable

no expressions in this context

Tree

width (px) height (px)

create an Expression, then click when viewing a variable with children to

```
Type "apropos word" to search for commands related to "word".
0x0000000000000000 in ?? ()
set breakpoint pending on

Thread 1 hit Breakpoint 1, main () at source/app_main.c:221
221      print("sono l'applicazione\
");

Thread 1 hit Breakpoint 2, test1 () at source/app_main.c:133
133      print("partenza 1\
");
(gdb)  enter gdb command. To interrupt inferior, send SIGINT.
```

Figure 9.3: gdbgui browser interface

Chapter 10

Conclusions and Future Work

This chapter enumerates possible directions in which MaldOS could could grow. As a general note improvement is always possible; for example, the hypervisor path described in section 7.7 has not been dismissed entirely. Other than that there are also more specific branches to be followed both in the near and distant future.

10.1 Extending Qemu

The recently added Raspberry Pi machine configuration for Qemu only supports a few capabilities of the original board: the two serial interfaces, the framebuffer display and the microSD card EMMC. The biggest missing part is of course the USB controller (bringing around the Network interface as well); the base complexity of the USB protocol, however, would probably make it an unsuitable choice for learning projects anyway.

Peripherals of less practical value in an emulator would perhaps end up being most interesting in the scope of OS study. SPI and I2C are relatively easy low level serial protocols that could make an interesting addition to the learning program; same goes for the PCM audio interface and the whole GPIO header in general. Qemu is a fairly flexible emulator, and a future improvement could focus on enriching the virtual environment with more device options.

There is another consideration which has been ignored for the purposes of this work: Qemu exists to virtualize generic machines for any architecture. The Raspberry Pi is a premium choice when the objective is learning, but an entirely new and ad-hoc virtual machine could be created and run on the emulator just like μMPS2 does, only with the advantage of a widespread and general purpose tool as backend. One could implement printers, terminals,

disks, tapes and even network interfaces not just as peripherals emulated by the abstraction layer (which is somewhat redundant in a virtualized machine) but by the Qemu emulator itself.
Moving away from ad-hoc tools in favor of Qemu is still an improvement in the author's perspective, even though this obviously drops the advantage of running on real hardware as well. However, on a more ambitious and visionary note, would it really? FPGA technology has become more and more affordable in recent years, and with a stretch of imagination the near future could harbor a FPGA board popularized like the Raspberry Pi and capable of being programmed as an ARMv8 (or other, like RISC V) core. In this scenario the acts of building a new virtual Qemu machine and a real FPGA-hosted SoC would overlap, allowing for a completely controlled real **and** virtual learning environment.
This is an arguably possible but very wild speculation.

10.2 Debugging with GDB

Being a off-the-shelf software GDB is flexible enough to be extended with a specifically tailored client. GDB provides a machine interpreter that recognizes machine readable commands for the purpose of creating higher level interfaces.

If the generic approach of Gdbgui was deemed too complex for inexperienced graduate students one could implement a μMPS2-like debugging interface that connects to the Qemu GDB server. Following a more common approach for debuggers, a GDB compliant environment could be created inside a widely spread, extensible IDE like Atom or Visual Studio Code.
Indeed, by using generic GDB interface extensions it is already possible to debug Qemu from VS Code. To improve on this it would suffice to add debugging widgets with specific register and device views.

10.3 Other ARM64 SoC

Although it is now firmly seated in the Olympus of open source educational boards, the Raspberry Pi family is build on awfully obscured and undocumented hardware. Broadcom follows the market trend of not releasing any information on its products like other manufacturers.

There are many Raspberry Pi-like boards that base themselves on similar hardware (namely, a powerful ARM CPU assisted by a graphical processing unit). In principle, the work that has been done for the British board could be easily ported to a wide number of similar devices. For example Pine64, an open source family of products taking heavy inspiration from the Raspberry Pi inheritance, has recently marketed a laptop powered by one of their compute modules and is planning on a smartphone and tablet with the same characteristics. Running a toy OS on a real-world mobile device could be both a more academically interesting exploit and an higher highlight for an undergraduate (or even graduate) student.

10.4 Other Programming Languages

Traditionally, Operating System kernels are written in C with critical parts coded in Assembler, with some alternative higher level components using C-derived languages like C++ (Windows and Android) and Objective-C (IOS). In the introduction some notable exceptions were mentioned: the Ultibo project [8] is a Raspberry Pi hardware abstraction layer written entirely in Pascal; Circle64 [9] is a similar library where C++ is used instead; when performance and resources are limited, one can find kernels written entirely in Assembler (like KolibriOS).

Different programming languages obviously bring advantages and disadvantages, but in the scope of educational work the choice falls unambigously on pure C. It is the standard in Operating System development, and for good reasons: simplicity, ease of carrying on low level memory operations, low portability dependencies, huge community and learning reference, perfect balance between abstraction and hardware fidelity.

Even beyond an academic objective, C is by far the dominant choice when it comes to kernel development and has been for decades. There is, however, a contender that is recently emerging and gaining purchase thanks to its qualities like performance, reliability and security: Rust.
Unlike the multitude of newly created programming languages of our time, Rust has quickly managed to carve a spot in embedded development. As of now, it is already possible (and fairly easy) to run Rust code in various bare metal ARM environment, Raspberry Pi included.

The main drawback lies in its inherent complexity: concepts like ownership and parallelism take a great deal of effort before mastery, and that would contrast with the focus on OS development of this work. Nevertheless, Rust is considered to be the closest second to C when it comes to the choice of programming language and if it was included as an additional topic of study in future Computer Science courses the opportunity should be considered seriously.

10.5 Emulation Layer on Videocore IV

Through the mailbox interface it is possible to ask the Broadcom GPU to directly execute code: this is what happens with graphical firmware routines in fully fledged Operating Systems like Raspbian.
In the early years code for the GPU was released only in binary form as proprietary software; only in 2012 Broadcom gave in and documented open source firmware for graphical processing. To this day a small component of proprietary binaries is still necessary to boot the Raspberry Pi (for example, `bootcode.bin`), although there are timid attempts at reverse engineering that as well [16].

If the Videocore IV was freely available for the developer like the ARM CPU, part (if not the entirety) of the abstraction layer that is implemented as some kind of *interrupt-in-the-middle* could be moved there. The interface to access abstraction capabilities would not be much more convenient for the user, as the communication media would still be mailboxes; it would however provide a more detailed emulation of hardware devices, for examble by acting as a real DMA peripheral when reading and writing to emulated disks and tapes. Students could write completely from scratch their Operating System, relying on GPU firmware to assist them in more complex tasks like device communication and boot process.
It should be noted that this approach would probably break Qemu emulation, because the general purpose tool virtualizes the architecture, not the hardware. There is no Videocore in the `raspi3` machine: the kernel image is loading by Qemu and it is unclear to what degree mailboxes to the GPU respond correctly.

10.6 Course Organization

MaldOS was developed to be of assistance for students. The focus is on an Operating Systems course, so effort was put into shaving off low level hardware interfacing and initialization: the objective is to learn how an OS works, not the AArch64 Assembler instruction set and registers, or how to operate the Arasan EMMC controller.
This approach however could be radically different depending on the subject under scrutiny. Low level Assembler programming and the ARM64 architecture are interesting topics for a course in System Architectures. Given enough time, will and coordination, the Kaya OS project could be build bottom-up by students in its entirety over four semesters and two different

courses: System Architectures for initialization and peripheral interfacing and Operating Systems for the kernel and process management.

In that scenario, MaldOS is not a support for students anymore, but a mere example and study on how to develop such an abstraction layer. Some sections would probably end up identical even when created by other developers (there are few alterations that can be made to the boot code, for example), but the interrupt handling in particular would become much less arbitrarily complicated.

Overall, it would certainly be an extremely formative experience.

Bibliography

[1] Andrew S. Woodhull, Andrew S. Tanenbaum, Operating System Design and Implementation, 1997.

[2] University of Cambridge, Department of Computer Science and Technology, Baking Pi - Operating Systems Development, `https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/`

[3] M. Goldweber, R. Davoli, and M. Morsiani, T̈he Kaya OS project and the μMPS hardware emulator,̧ SIGCSE Bull., vol. 37, pp. 49-53, June 2005.

[4] T. Jonjic, D̈esign and Implementation of the μMPS2 Educational Emulator,̧ Alma Mater Studiorum, 2012.

[5] M. Melletti, S̈tudio e Realizzazione dell'emulatore μARM e del progetto JaeOS per la Didattica dei Sistemi Operativi,̧ Alma Mater Studiorum, 2016.

[6] M. Goldweber, R. Davoli, μMPS Principles of Operation, Lulu Books, 2011

[7] Raspberry Pi OS, `https://github.com/s-matyukevich/raspberry-pi-os`

[8] The Ultibo Project, `https://ultibo.org/`

[9] The Circle C++ environment, `https://github.com/rsta2/circle`

[10] BCM2835 ARM Peripherals, Broadcom.

[11] ARM Quad A7 Core, Broadcom.

[12] PrimeCell UART (PL011) Technical Reference Manual, ARM.

[13] ARM Cortex-A Series Programmer's Guide for ARMv8-A, ARM, 2015.

[14] ARM Architectural Reference Manual ARMv8, for ARMv8-A Architecture Profile, ARM, 2017.

[15] ARM Cortex-A53 MPCore Processor, ARM, 2016.

[16] Raspberry Pi Open Source VPU-side Bootloader, `https://github.com/christinaa/rpi-open-firmware`

Acknowledgements

My thanks go to the many people that assisted and sustained me while preparing my work.

My friends, colleagues and parents who had to endure rambled concerns about my work for more than a year and that never failed to remind me how unfounded and childish my anxiety was.

My brother, who went through more than a hundred pages of mostly incomprehensible Computer Science jargon to proofread my dissertation.

My grandparents, who attentively listened every time I was describing the current obstacles I was facing, always trying their best to follow a topic so obscure to them.

My employers, for giving me the opportunity to organize work and study at my own pace and with no limitations.

All the ever-helpful users of the Raspberry Pi bare metal forum for promptly answering all of my doubts and uncertainties about ARM development and the BCM2837 board.

Last but most certainly not least important I am grateful to my advisor, professor Renzo Davoli, for granting me trust and guidance in developing this project, with the intent of keeping it alive if it turned out to be academically sound and useful.