ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

TITOLO DELLA TESI

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Mattia Maldini

Sessione III
Anno Accademico 2018-2019

Questa è la DEDICA*:*
ognuno può scrivere quello che vuole,
anche nulla . . .

Abstract

The course of Operative 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 operative 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 thorugh 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 find a proper machine architecture to run the project on.

This work is heavily inspired by uMPS2 (and uARM), 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), uMPS2 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 not prepared 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 μMPS) emulator, all the code was written trying to remain as simple and clear as possible to encourage a deeper study as example.

Sommario

TODO: traduzione dell'abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

An operative 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 previously mentioned, 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 μMPS [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. 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 to 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 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].

It is argued that the last approach is the most interesting and valuable for the students. If they are to study an existing Operative 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 int 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.

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.

1.1.2 ARMv8 and Raspberry Pi

The passage from MIPSEL to ARM is not new to μMPS; the previous work of μARM was already pointed in this direction. μARM had the goal to modernize the μMPS experience, thus maintaing 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). ¡mipsel is outdated¿ 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 make 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. 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-bit 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 devices. Having an - albeit small - experience 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 μMPS. Recently however an official patch has been added to qemu that allows to emulate a Raspberry Pi 3 board and debug the software with GDB. Working with Qemu and GDB brings, in the author's perspective, the added 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 to be linked with the student's work, which provides initialization and a partially virtualized peripheral interface. It was developed around the Kaya Operating System Project, with the main influence being the implementation of a virtual interface for emulated peripherals that are 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 other pedagodically meaninful peripherals (the only exception being two UART serial interfaces).

1.1.4 Existing Work

Surprisingly, there is not much existing work on OS development for Rasbperry Pi boards and the Broadcom SoC used 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 works 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. 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++. 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.

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 work 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 produced by its users. Even if the focus of that community is more shifted on embedded programming than Operating Systems development, their work in hacking and reverse engineering the harware 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 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.

Chapter 5 dwells on the implementation of mechanisms commonly used by an Operative System like context switch, scheduling, interrupt management and memory virtualization, which should be most interesting for students approaching this project.

Chapter 6 covers the "emulated peripherals"; those are the devices available in μMPS and μARM, absent in a real system such as the Raspberry Pi. The hardware abstraction layer uses the existing mailbox interface to seamlessly emulate said devices on top of other resources. For the end user, the illusion to use a real peripheral is perfect.

Chapter 7 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, chapter 8 a recap is made about the success of this work and directions for future works are listed.

Chapter 2

Discarded Options

Before settling for 64-bit 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* 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-bit 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 μUMPS, 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 μUMPS 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 the 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 the previously considered model 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.

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

2.3 Rasbperry Pi 3 (ARM64)

The final choice was the Raspberry Pi 3 (any model, in theory). Though sharing 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, Qemu [1]. The support of the raspi3 machine on Qemu came only recently (version 2.12.1, August 2018) and the opportunity was seized immediatly. Qemu support means kernels meant for the board can be more easily tested on the emulator and debugged with GDB. This permits to keep the advantages of a virtual environment like in μMPS 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, the Mini UART or UART1 and the USB controller (that manages Network peripherals as well). Of those three limitations only the USB controller cannot be overcome: the System Timer can be replaced by the internal ARM Timer, and the UART1 is not the only serial interface built in on the board (Qemu emulates UART0). 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 bit support while still keeping backward compatibility for 32 bit applications. In theory, the student developed kernel could still use a 32-bit 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 [12] and the Arm Reference Manual [13].

ARMv8 is the latest generation of ARM architectures, following ARMv7. It brings an enourmous list of changes from its predecessors, finally adding a 64 bit option to the family; it does so while still keeping backward compatibility towards 32 bit code and applications. The execution state in which an ARMv8 processor runs 64 bit code is called *AArch64*, while *AArch32* identifies the compatibility state for 32 bit 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 bit 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 bit 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

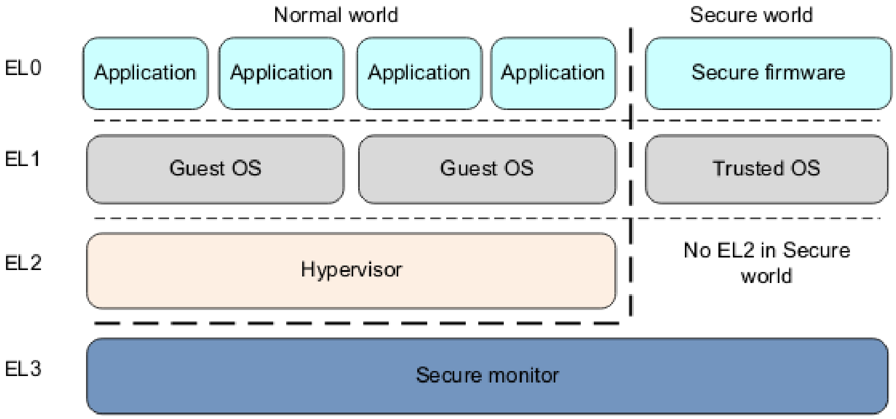


Figure 3.1: ARMv8 Exception levels and their main purpose

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. It is meant to run user application, 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 harware 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 ever handled at **EL0**, **EL0** is only reachable on `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**, 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, that lies entirely in AArch64.

3.2 Registers

3.2.1 General Purpose Registers

One of the immediate benefits of 64 bit 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.

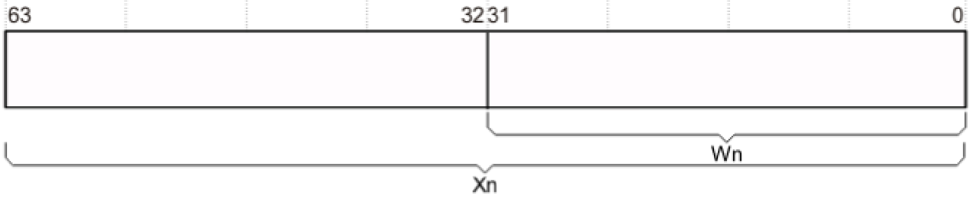


Figure 3.2: 64 bit register with "x" and "w" access

3.2.2 Special Registers

There are 5 special registers:

Zero Register: **xzr** and **wzr** provide access (as 64 and 32 bit 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 and 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.

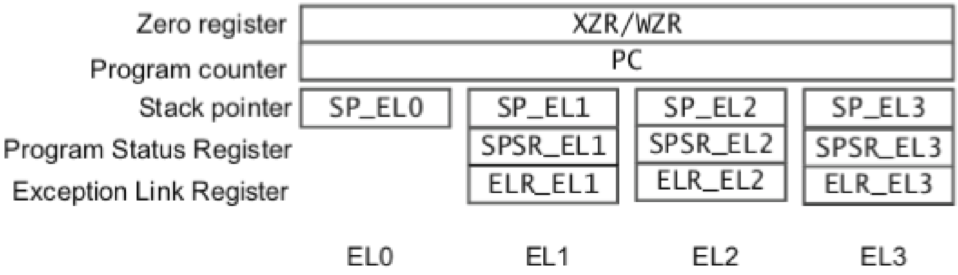


Figure 3.3: AArch64 special registers

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 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, of which one was reserved for system registers management. While coprocessors had to be controlled via specific instructions ARMv8 system registers are directly accessed in Assembly 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. There 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 [13].

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 **??** for more on exceptions). It is necessary to distinguish between exception classes and to find details specific to the exception.

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 runnint 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 **??** 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**. 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 in Assembly code.

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

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 immediatly 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 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 **VBAR_EL*n*** register, for 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 used.

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.

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

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.

Each section has four different handlers for synchronous exceptions, IRQ, FIQ and SError.

3.3.1 Interrupts

Interrupts can be fast interrupts (FIQ) or normal interrupts. Aside from the fact that FIQ have higher priority, these two types of exception are virtually identical. 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 to input pins to the core. The connection can be direct or, more commonly, pass through an external device called interrupt

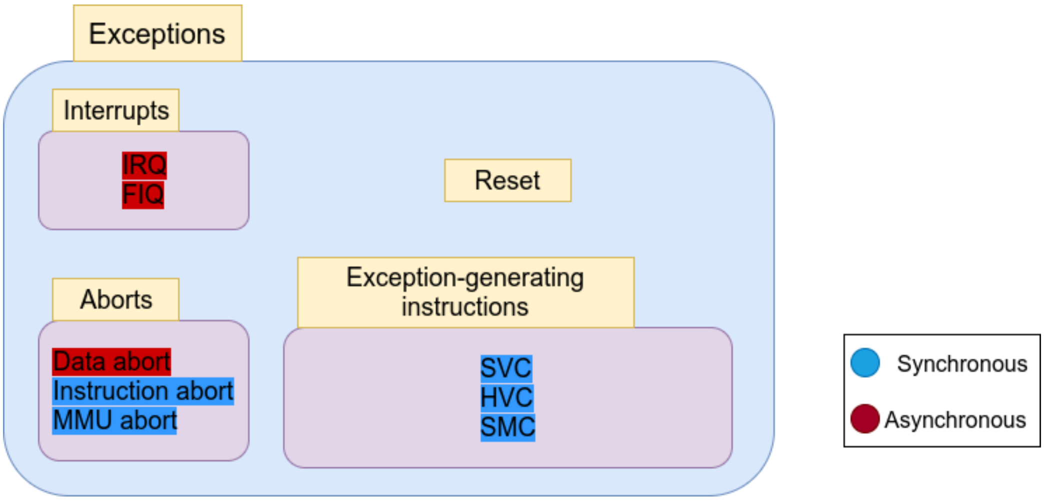


Figure 3.4: Tree of exception classes.

controller that elaborates interrupt priorities and organization (see section 4.4).

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.

3.3.2 Aborts

Abort exceptions, also called system errors (SError), occur every time some 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 Manage 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` in the case of the 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 Assembly instructions that induce an exception to a higher exception level, usually to require a service paired with an 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 instruciton 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**. Allows to require *secure world* context switch.

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

3.4 The Memory Management Unit

3.5 Memory Attributes

3.6 Multiprocessor

3.7 Security

3.8 ARM Timer

Chapter 4

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 Rasbperry Pi 2) and BCM2835 (the first used, for the Raspberry Pi 1): the scarce documentation is only available for BCM2835 [9] (and partly for BCM2836 [10]) allegedly because nothing changes from the developer perspective; the actual differences have been figured out mostly through reverse engineering from the code of the 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.
- A USB controller.
- Two UART serial interfaces.
- An external mass media controller (the microSD interface).
- Other minor peripherals (I2C, SPI,. . .).

4.1 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 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 RAM 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.

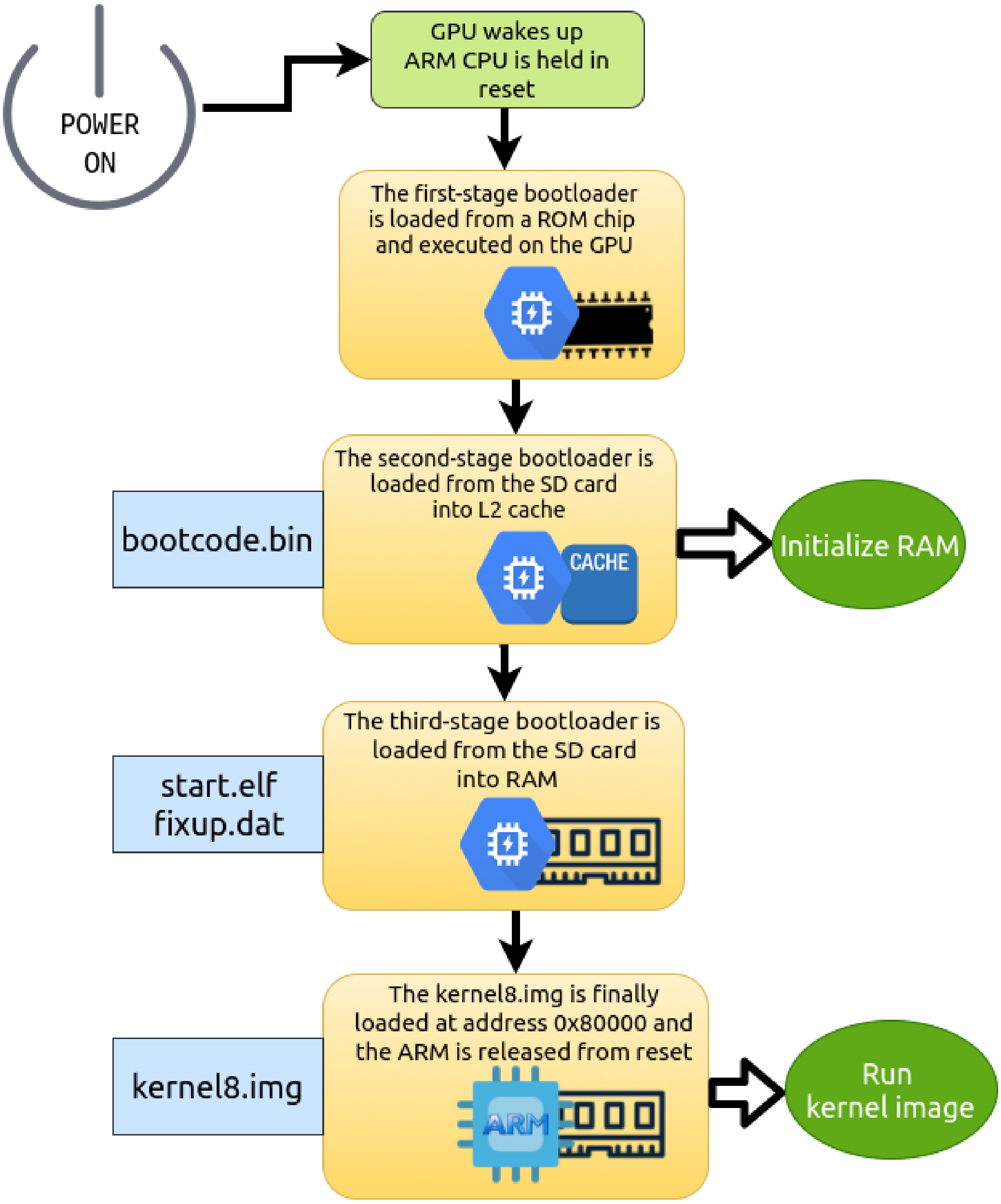

POWER ON
GPU wakes up ARM CPU is held in reset
The first-stage bootloader is loaded from a ROM chip and executed on the GPU
bootcode.bin
The second-stage bootloader is loaded from the SD card into L2 cache
CACHE
Initialize RAM
start.elf fixup.dat
The third-stage bootloader is loaded from the SD card 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 4.1: Explanatory diagram on BMC2837 boot sequence

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.

4.1.1 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 in RAM.

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 RAM.

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.

4.1.2 Configuration

It is possible to configure in different ways the boot process by combining different firmware binaries and config.txt options, but this work always uses the default with no extra steps needed; this is to ensure the usage is kept as simple as possible and since 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.

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.

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 harware 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 chapter 7).

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. Without other instruction 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.

4.2 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.

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 several 32 bit registers providing status information, read and write access. If a value is written on right the 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, containing more elaborate commands and parameters.

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.

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 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.

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. 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.

4.3 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.

4.3.1 GPIO

4.3.2 External Mass Media Controller

4.3.3 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 concurrency programming challenge for a student, as both can run successfully if properly alternated.

Those devices bear a strong similarity to μMPS' terminal devices, both having similar registers to check the current status and read or write character on the interface. For this reason, except for the initialization of the peripheral which is done entirely by the harware 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 structure, having about four registers scattered over a larger memory area instead of a compact structure; after providing a focused and

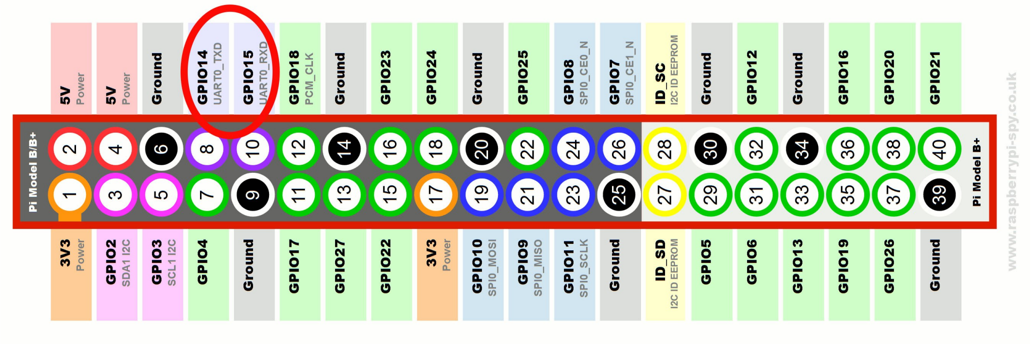


Figure 4.2: Highlight of UART reserved GPIOs

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 [11]. 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 one 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 harware, 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 yield 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 requested) 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.

4.4 Interrupt Controller

The BCM2837 SoC has at least two devices acting as interrupt controllers. One of them is clearly defined in the peripheral datasheet [9], while the other is not clearly named but hinted at thorugh register definition in a later revision [10]. Those are here arbitrarely 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.

From a practical standpoint there are often serveral registers indicating which interrupt line is being asserted at any 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. 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,

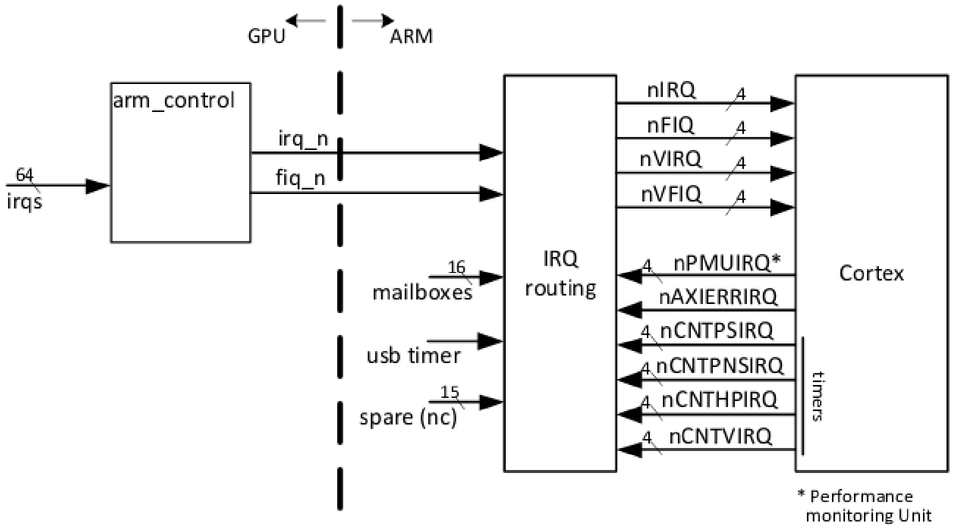


Figure 4.3: BCM2837 interrupt controllers configuration

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 4.3).

4.4.1 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 for each core and for each mailbox the GIC exposes three types 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 the same content that was read from the register must be written back on it.

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

Chapter 5

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 instance 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 phylosophy of this work.
- The USB controller suffers a even worse degree of complexity, to the point that 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 inter-

sting 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.

5.1 Emulated Device Interface

Initially emulated devices were made accessible via fake registers: simple pre-established memory locations that were frequently polled 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 lead 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.

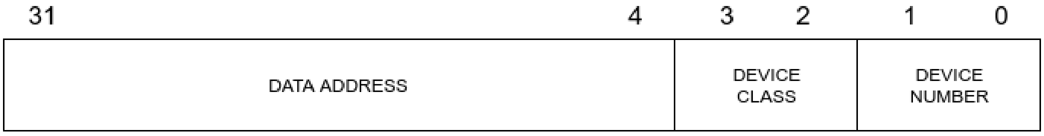


Figure 5.1: mailbox structure

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 four least significant bytes. Since it is a mechanism already present in the system it fits naturally in the development process.

The "register" structure that should be pointer 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 5.1: device register 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, as real world devices take time to operate; as such, there are fabricated delays between commands and execution.

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 5.2: emulated interrupt lines

5.2 Tapes

Four instance of the tape device are supported. They are read only and work as if queried through a DMA system. The tape can be viewed as a sequential list of 4KiB blocks. each block is marked with a 4 bytes delimiter denoting the content of the underlying block.

5.3 Disks

5.4 Printers

Chapter 6

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 of modifying the work, or to study ARM bare metal development.

The size of the project is comparatively small, only reaching about 4000 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.

6.1 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 exaplain 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. Third are the modules of the emulated devices like tapes and printers, leaning on the previous utilities to create the illusion of physical peripherals.

6.1.1 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 two Assembly source files: `init.S` is the absolute first entry point and provides initialization for system registers, interrupt vectors, 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.

6.1.2 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 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 is 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 user control over the process.

6.1.3 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.

6.2 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` is called.

6.3 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] The Rasbperry Pi 3 starts in **EL2**, while Qemu initially runs at **EL3**

1. ARM timer
2. UART0
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 queue of virtual timers. Once the time slice is over the internal interrupt handler is called. It is responsible for operating emulated devices, but other than that immediatly passes control to the user defined interrupt handler.

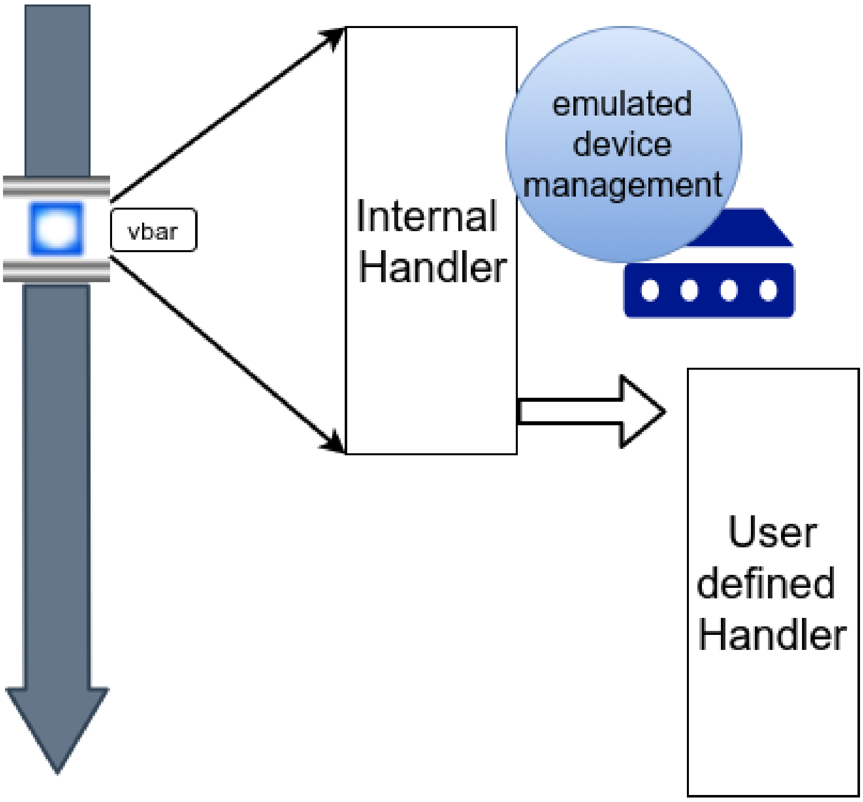


Figure 6.1: Interrupt handling schematic

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

6.4 Emulated Devices

The idea behind emulated peripherals and their fabricated interface have already beed described in Chapter 5. Here we give a more detailed presentation about the principle 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 obscured 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 timer is set to be executed by a normal interrupt after a certain amount of time.

6.4.1 Timer Queue

The second step of some device commands is scheduled for execution after a while; since there is only one timer in the system to fire scheduled interrupts, this would eventually overwrite other pending timers (namely the process time slice). To prevent this internal timers uses a set of queue managing functions to schedule multiple timers at once. The ROM function `setTIMER()` itself just pushes a new timer onto the queue.

The queue is kept ordered from the first timer that will occur to the last. When a 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.

Interestingly, the implementation of this module is heavily inspired by the solution to phase 1 of the KayaOS project, covering process and semaphore queues.

6.5 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.

Chapter 7

Student's Perspective

Chapter 8

Usage and Debugging

8.1 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. 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.

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

8.2 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.
- 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.

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).

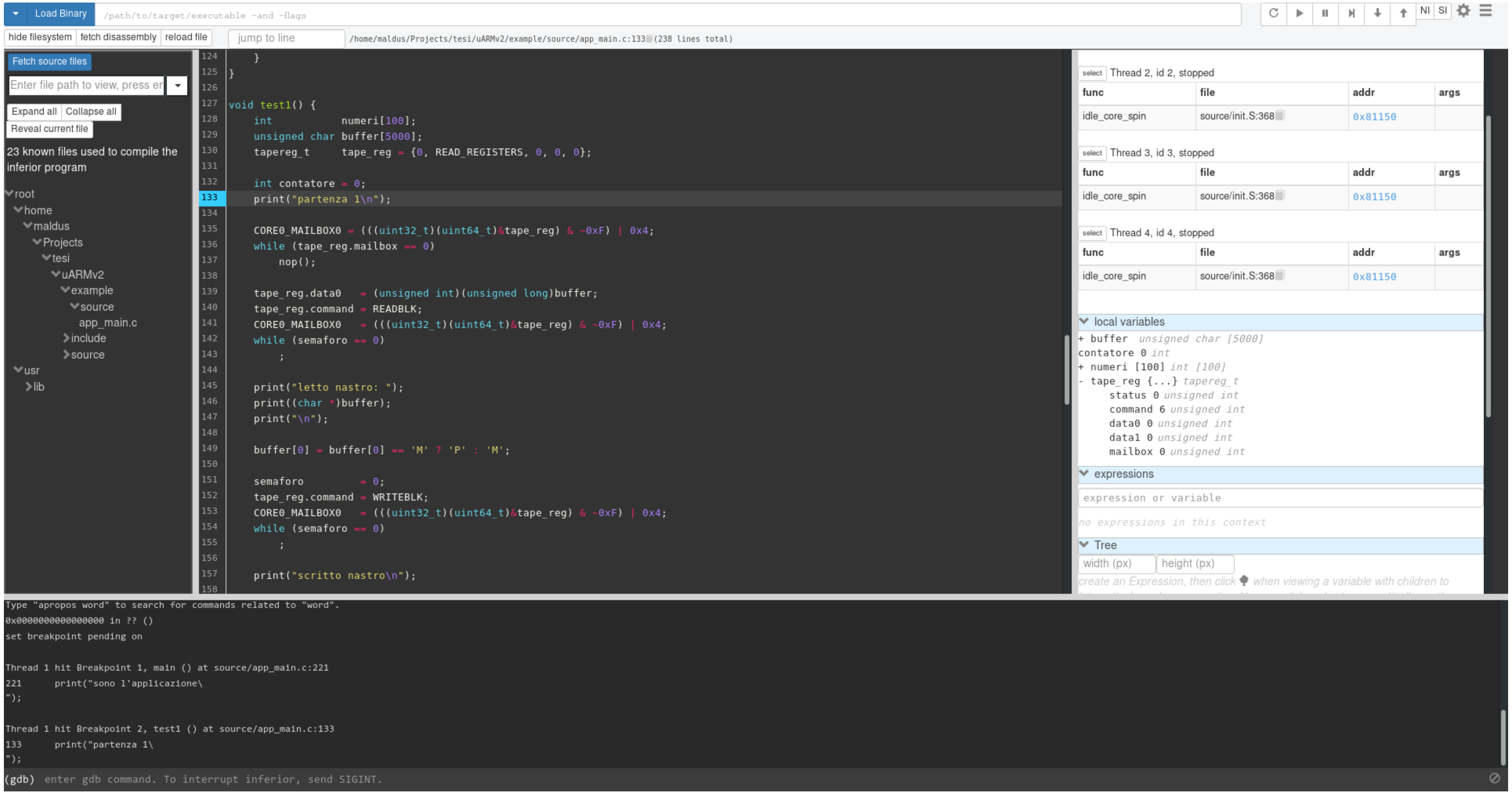
8.3 Debugging

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
file output.elf
target remote localhost:1234
```

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. 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.



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

23 known files used to compile the inferior program

```
    }
}

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	

```
+ 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
```

no expressions in this context

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 8.1: gdbgui browser interface

Chapter 9

Conclusions and Future Work

9.1 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.

9.2 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. Using the same interface but on a different note a debugging environment could be created inside a commonly used IDE, like Atom or Visual Studio Code.

9.3 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 potent 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. The Pine64 family, for example, has recently marketed a laptop powered by one of their compute modules. Running a toy OS on a real laptop could be a even higher highlight for an undergraduate or even graduate student.

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

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

[9] BCM2835 ARM Peripherals, Broadcom.

[10] ARM Quad A7 Core, Broadcom.

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

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

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

Ringraziamenti

Qui possiamo ringraziare il mondo intero!!!!!!!!!!
Ovviamente solo se uno vuole, non è obbligatorio.