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.

[Mention JaeOS, KayaOS, uARM, uMPS articles] One option is to use an already existing OS as reference while studying its operating principles. ...

1.1.1 Shortcomings of μMPS

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 [1], 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

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 [7] (and partly for BCM2836 [8]) 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.

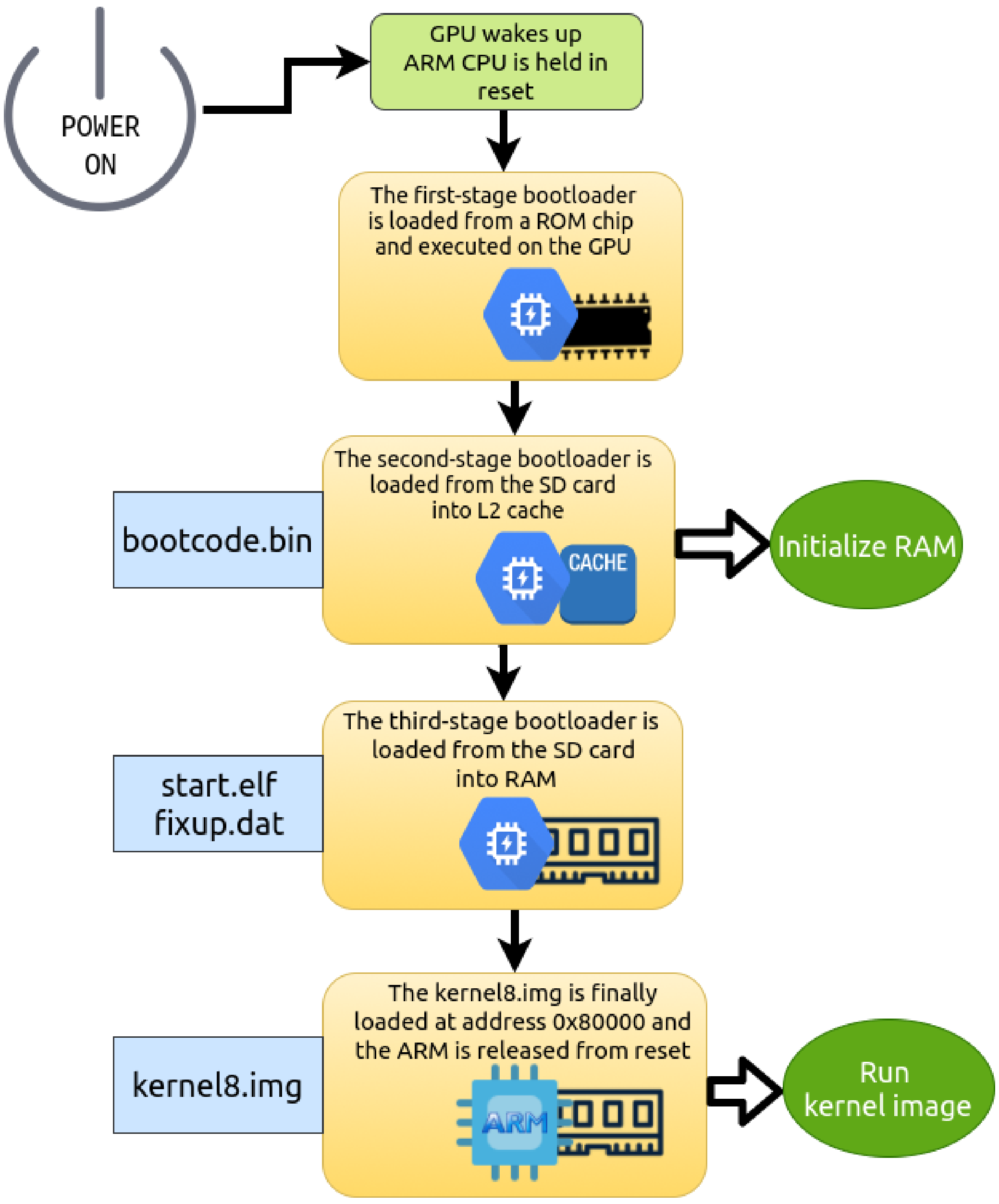
The BCM2837 contains the following peripherals, accessible by the onboard ARM CPU:

- 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

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).

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.

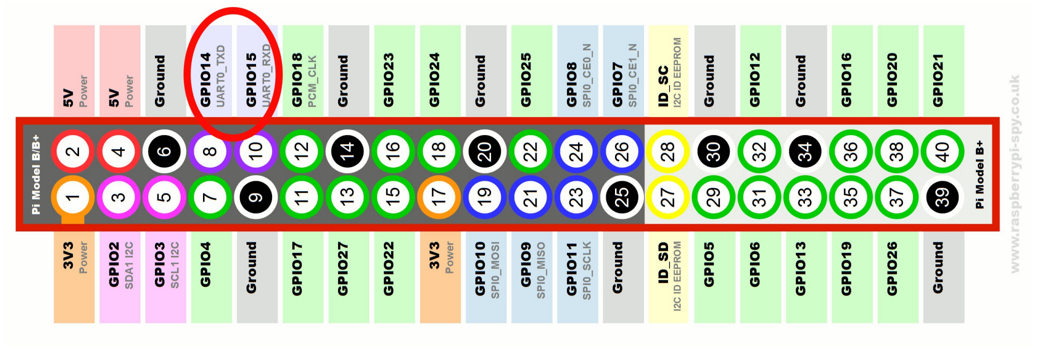
4.2 Peripherals

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.

4.2.1 GPIO

4.2.2 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.

UART0

The UART0 is a fully fledged asynchronous serial interface, abiding to the PL011 ARM specification [9]. Its register are located starting at the address 0x3F201000, each of them is 32 bits wide and they are organized as follows:

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 corrisponding 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 with no parity bit and with a baudrate of 115200.

Chapter 5

Operating System Facilities

Chapter 6

Emulated peripherals

Chapter 7

Usage and Debugging

Chapter 8

Conclusions and Future Work

8.1 Debugging with GDB

8.2 Other arm64 SoC

Bibliography

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

[2] 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.

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

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

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

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

[7] BCM2835 ARM Peripherals, Broadcom.

[8] ARM Quad A7 Core, Broadcom.

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

Ringraziamenti

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