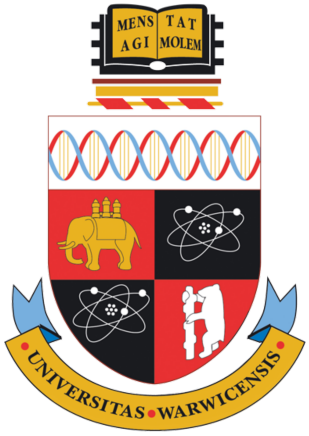

An Educational Kernel for the Raspberry Pi

Thomas Archbold
1602581
Department of Computer Science
University of Warwick

April 29, 2019

CS310 Third-year Project
supervised by Adam Chester

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Index terms – Operating systems, kernel, Raspberry Pi

Contents

1 Introduction

Operating systems are some of the most pervasive pieces of software around, but due to their inherent complexity, their inner workings are often impenetrable to understand without specialist knowledge. While widespread access to a personal computer is nothing new, the introduction of the Raspberry Pi in recent years has rendered experimentation with computers much more affordable and hence readily available, inviting tinkering at all levels with less concern of economic loss. The Pi therefore provides an ideal platform for operating systems education – novice developers looking to get involved in writing such systems have access to a standardised set of hardware that is inexpensive both to maintain and replace, if and when things go wrong. Now seven years since its initial release, the Raspberry Pi has several official operating systems to offer, each addressing its own issue such as ease-of-installation, Internet of Things integration, or classroom management [1], with many more unofficial ones. However, there is less in the way of those written to teach concepts of the operating system itself – this project attempts to fill this gap by providing a configurable, educational operating system for the Raspberry Pi 1 Model B+, with a focus on presenting code which is simple to understand and providing clear interfaces to encourage ease of extensibility, and hence a practical, software-driven approach to learning about operating systems.

1.1 Motivation

An operating system draws together aspects from all over the field of computer science, whose development requires intimate knowledge of low-level concepts such as the computer's organisation and architecture, up to an understanding for the more abstract in designing how processes communicate or implementing filesystems. The opportunity to write one as part of this project was therefore appealing as it served not only to provide the author with experience in an area of computer science of great interest, but also the chance to unite and put into practice many of the topics learned throughout the undergraduate computer science course. One of the key motivations of this project was therefore to gain experience in low-level systems programming and interacting with real-world hardware, all the while creating a useful and entirely self-contained piece of software.

The main goal of this project is to provide an operating system for the Raspberry Pi that is capable of booting on real hardware, for both educational and hobbyist use. An important aspect of this is gaining practical experience in this unique area of software development, and so in addition to being configurable at compile-time, offering multiple approaches to process scheduling and inter-process communication, it also aims to easily-understandable and open to extension. In order to achieve this it must also implement a basic interface to manage this compile-time configuration. Also key to the project's success is the ability to boot on real hardware - while it would largely be possible to implement in a solely emulated environment with the help of a virtual machine, this project attempts to provide the additional valuable experience of taking real-world design considerations and observing their effects on live hardware, the latter of which is easily ignored if working only through an emulator.

2 Background

2.1 Relevant Material

To this end, there are a handful of modern resources for getting involved with operating systems development – a particularly useful one at the time of first carrying out research for the project's proposal was `wiki.osdev.org`, which contains information about the creation of operating systems and acts as a community for hobbyist operating system developers. However, much of the focus is on the x86 platform and past providing a brief overview of the idiosyncrasies of the Raspberry Pi as well as the code to get a barebones kernel to boot, there is little material on the specifics required to get core systems working on the platform. Cambridge's *Baking Pi* [2] provides more help in this regard, with Alex Chadwick's comprehensive tutorials proving an invaluable resource for information such as accessing registers and peripherals specific to the Raspberry Pi. The project can, however, be much further extended to guide through the implementation of core operating system concepts such as memory management, the process model, inter-process communication, and filesystems. Another aspect in which this series of tutorials diverges with the goal of this project is the language in which it has been implemented – while assembly is an undeniably language in which to be competent, it is not the most easily-understandable, in stark contrast to what this project hopes to achieve. The resource which aligns most tightly with the aim of

this project is [3], whose tutorials have served as an outline to how many key features of the project have been implemented.

Finally, other notable resources which are in place to teach general operating systems development are Stanford's *Pintos* [4] and Tanenbaum's MINIX operating system [5]; the former was written to accompany the university's CS140 Operating Systems course, while the latter is an illustrative operating system written alongside the book *Operating Systems: Design and Implementation*, as a means of providing concrete examples of how operating system features are implemented in practice. Helin and Renberg's *The Little Book About OS Development* [6] also serves as a guide to writing one's own operating system. The only drawback to these three is their focus on the x86 architecture, and while they are useful resources it is in concept only, given the gap which was found to quickly form from focusing on a different processor.

2.2 Why is this project worthwhile?

The project is worthwhile firstly as it provides an accessible gateway into systems programming and operating systems development. Given the relative difficulty and additional effort required to get involved with this area of software development as opposed to others, for example by reading technical reference manuals and building an intimate knowledge of the hardware with which you are working, it therefore finds its use in easing this transition and making the learning curve associated with its involvement less intimidating and more approachable. In doing this, the project is also worthwhile in that it demystifies some of the key considerations that go into operating system implementations, not only in high-level concepts such as processes, but also the low-level with notions such as memory-mapped I/O and the processor's registers. In providing this opportunity to see theory in practice, it further opens up the opportunity for experimentation and invites practical self-learning, and hopefully clarifying why existing operating systems work the way they do.

While there are similarities to be drawn between the aims of this project and those of the current background material, both looking to create a more accessible way in to operating systems development, this project addresses the gap that they leave unfilled by tackling a different architecture, as well as in approaching feature implementation in a more modular manner. Thus the project forms one more part in the ecosystem of introductory and instructional operating systems.

2.3 Useful concepts

Operating System - a program that manages a computer's hardware that acts as an intermediary between the user and said hardware [7], providing an environment in which a user can execute programs conveniently and efficiently.

Kernel - the one program running at all times throughout an operating system's execution, the kernel is often tasked with managing the most vital/recurring tasks. Other key types of programs include system programs and application programs.

Freestanding environment - typical programs are written to run in a hosted environment, meaning they has access to a C standard library and other useful runtime features. Conversely, a freestanding environment is one which uses no such pre-supplied standard, meaning a bespoke one must be supplied. Any functions we wish to use as part of the operating system we must define ourselves.
Cross-compiler - a regular compiler will generate machine-code which is specific to that on which the code has been compiled. By contrast, a cross-compiler allows us to write code on any machine and compile it for our target architecture (the architecture on which we design our code to run).

Linker - responsible for linking all object (`.o`) files generated by a compiler/assembler into a single executable (or 'library file', see [**?**]). It also defines the entry point and the location of the various sections (detailed in Section 3.2.2) in the final `.elf` file.

Exception - an event triggered when something exceptional happens during normal execution (hardware providing the CPU with data, privileged action, bad instruction, etc.).

Process - a program that has been loaded into memory and is executing.

Context Switch - the act of saving the currently executing process into memory (**state save**) followed by loading the saved state of a different process (**state restore**). It allows the CPU to switch from executing one process to executing another.

Concurrency - the ability for multiple processes to make progress seemingly simultaneously, as a result of being rapidly brought into and out of memory with respect to a policy enforced by some scheduling algorithm.
Synchronisation - the prevention of **race conditions**: when the outcome of two concurrently executing processes depends on the order in which data access took place.

Inter-process Communication - the mechanism by whic processes may exchange data and information.

Peripheral - a device with a specific memory address which it may write data to and read data from. All peripherals may be described by an offset from some base address, covered in Section **??**.

$(2^{10})^2 = 1,048,576$ bytes, respectively. This is to ensure clarity and avoid confusion with their SI-prefixed counterparts, namely kB and MB, which instead correspond to 10^3 and 10^6 bytes.

3 Design

3.1 Hardware

The project has been developed for the Raspberry Pi 1 Model B+. Some of the relevant hardware onboard includes:

- System-on-Chip: Broadcom BCM2835
- CPU: 700MHZ ARM1176JZF-S
- GPU: 250MHz Broadcom VideoCore IV
- SDRAM: 512MiB, shared with the GPU
- Video output: HDMI and DSI
- Storage: MicroSDHC slot
- 4x USB 2.0 ports
- 40 General Purpose Input/Output (GPIO) pins

Development was initially planned for the Raspberry Pi 2 Model B+ simply due to its availability, having been received as a gift some years prior. However, focus was switched to target the Raspberry Pi 1 Model B+ as a result of the difficulties encountered with interacting with the GPU via the mailbox interface, with the processing differing slightly and being more complex on the 2. The underlying architecture of the 1's BCM2835 chip is, however, identical to that of the BCM2836 and BCM2837 [8], used by the Raspberry Pi 2 and 3, respectively. They only differ in that the 1 uses the ARM1176JZF-S processor, as opposed to the quad-core Cortex-A7 and quad-core Cortex-A53 cluster used by these later boards, in addition to the 512MiB extra available to them. Therefore, the choice between specific models for which to develop would have made little difference to the outcome of the project, and indeed much of the code is transferable. As has been discussed, this standard set of hardware was desirable for the project's aims, rendering the operating system more widely accessible and for less effort.

3.1.1 The Raspberry Pi's boot process

The decision to work with the Raspberry Pi in particular, as opposed to a more open-ended PC setup, is not only due to the relative lack of material available for operating systems development on the ARM architecture, but also as a result of the much simpler boot process in contrast to other hardware, details of which were found from [9]. Booting is handled almost entirely by the Pi's system-on-chip, thus does not require the writing of a custom bootloader. Instead, it relies on closed-source proprietary firmware programmed into the SoC processor which may not be modified. The necessary files can be acquired by either downloading them from [10], or by downloading an existing operating system for the Pi and using the files that it provides (since it still requires the same firmware).

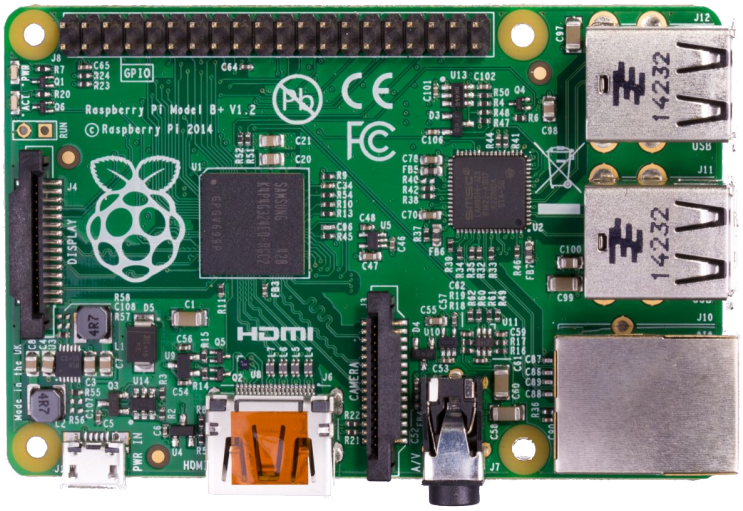

Figure 1: Raspberry Pi 1 Model B+

During system boot, the ARM CPU does not act as the main CPU, but rather as a coprocessor to the VideoCore GPU [11]. When the system is first powered on, the ARM CPU is halted and SDRAM is disabled. Control is passed to the GPU, whose responsibility it is to execute the bootloader. The bootloader itself is divided into three stages: the first stage, stored in ROM on the system-on-chip; the second stage, `bootcode.bin`, and the third stage, `start.elf`. The first mounts the FAT32 boot partition on the SD card to enable execution of the second-stage bootloader, and then loads this into the L2 cache to run it. Control is then passed to the second-stage bootloader, which enables SDRAM and loads `start.elf` for execution. This final stage allows the GPU to start up the CPU. An additional file, `fixup.dat`, is used to configure the shared SDRAM partition between the two processors. The GPU firmware reads the files `config.txt` and `cmdline.txt` to load the kernel image, then releases the CPU from reset and transfers control to it to begin executing the kernel.

After an operating system is loaded, the code on the GPU is not unloaded, but rather runs its own simple operating system, the VideoCore Operating System (VCOS) [12]. This can be used to communicate with the services provided by the GPU (for example, providing a framebuffer), using the mailbox peripheral and ARM CPU interrupts, which the GPU is capable of producing. The GPU is not only in charge of graphical functions as, for example, it also controls the system timer and audio – in this way it is therefore more akin to a regular PC's BIOS.

3.2 Development environment

The project was developed on various x86 machines running the Linux kernel version 4.16 onwards. Since the target environment, the Raspberry Pi 1 Model B+, runs on an ARM CPU, the target architecture is therefore different to that of the development machines. Therefore, a cross-compiler is required in order to compile the code for the target machine. Available on the ARM developer website is the GNU Embedded Toolchain [13]. Conveniently this suite of tools is available from Arch Linux's package manager, Pacman [14], and this is the version that has been used to develop the project.

3.2.1 Technical Documentation

With no prior experience working in assembly and, in particular, how to use ARM assembly in conjunction with the C programming language, it was necessary to become better acquainted with this environment, including becoming familiar with the instructions available and the calling conventions that must be followed (for example, returning the result of an assembly procedure call in register 0). This information was gathered by reading through both official and unofficial documentation on the

ARM environment in general and the specifics of working with the Pi. In particular, the following were the main resources of technical documentation used throughout the project:

- *ARM1176JZF-S Technical Reference Manual* [15]
- *ARM Developer Suite Assembler Guide* [16]
- *ARM Cortex-A Series: Programmer's Guide* [17]
- *Broadcom BCM2835 ARM Peripherals Manual* [18]

The technical reference manual provides important, in-depth architectural information upon which the processor operates, and, for example, details concepts such as the programmer's model or interaction with the system control coprocessor and the memory management unit. The online online guide at [16] provides comprehensive coverage of the instructions to use when using assembly to program on the ARM CPU, and used in conjunction with the programmer's guide provides an understanding of when and how to use certain instructions. Although not written for the processor used by the Raspberry Pi 1, instead written for the CPU used by the 2 and 3, the programmer's guide provided further examples of various instructions' use-cases, and its utility continued even when focus shifted from the Raspberry Pi 2 to the Raspberry Pi 1.

On the Raspberry Pi are various peripherals, such as the Universal Asynchronous Receiver/Transmitter (UART), system/ARM timers, or the interrupt controller. Information regarding the layout of their registers and the memory addresses from and to which to read and write was obtained from the peripherals manual. Since the underlying architecture behind the BCM2835 and BCM2836/7 is largely the same, the manual was helpful both in development for the Raspberry 1 and 2, with the BCM2836 [19] providing some extra processor specifics in the case of the latter.

3.2.2 System V ABI

Of particular use within GNU's suite of tools is the cross-compiler for `arm-non-eabi`, which provides a toolchain to target the System V ABI (Application Binary Interface). This is a set of specifications that detail the calling conventions, object and executable file formats, dynamic linking semantics, and more, for systems complying with the System V Interface Definition, of which the Raspberry Pi is one. For example, it defines the Executable and Linkable Format, or ELF, which is a format for storing programs or fragments of programs on disk that are generated as a result of compiling and linking. Each ELF file is divided into sections, specifically:

- `.text` - executable code
- `.data` - global variables which are uninitialised at compile-time
- `.rodata` - read-only data i.e. global constants
- `.bss` - uninitialised global variables

These are covered in more depth in the discussion of the linking process in section 4.1.1. Additionally the SysV ABI defines the `.comment`, `.note`, `.stab`, and `.stabstr` sections for compiler and linker toolchain comments and debugging information.

3.2.3 ARM environment

The ARM1176JZF-S processor implements the ARM11 ARMv6 architecture [15], supporting both the ARM and Thumb instruction sets[1]. The processor contains 33 general-purpose 32-bit registers and 7 dedicated 32-bit registers, 16 of which are accessible for general use at any one time in the ARM state. These are as follows:

- `r15` - Program Counter
- `r14` - Link Register
- `r13` - Stack Pointer
- `r12` - Intra-Procedure-Call Scratch Register

[1] A subset of the most commonly-used 32-bit ARM instructions. Each Thumb instruction is 16 bits long, and has a corresponding 32-bit ARM instruction with an equivalent effect on the processor model. While not useful and out-of-scope for this project, the instruction sets can be easily switched between to enable the programmer to optimize both code density and performance as they see fit.

- `r4-r11` - local variables to a function
- `r0-r3` - arguments passed to a function, and the returned result

The Current Program Status Register (CPSR) contains code flags, status bits, and current mode bits. Another register, the Saved Program Status Register (SPSR), is similar to the CPSR but is only available in privileged modes. This contains the condition code flags, status bits, and current mode bits saved as a result of the exception which prompted the processor to enter the current mode.

The architecture asserts a full descending stack: full, meaning the stack pointer points to the topmost entry in the stack[2]; and descending, meaning the stack grows downwards, starting from a high memory address and progressing to lower addresses as items are pushed.

As a final note on correctly interacting with the ARM environment, any procedure calls must preserve the contents of registers 4 to 11 and the stack pointer. Further, subroutines calling other subroutines must save the return address (found in the link register) to the stack before calling that subroutine.

3.2.4 Tools

The following is a summary of the tools used to develop the project:

Languages	C, ARM assembly
Cross-compiler toolchain	GNU Embedded Toolchain (`arm-none-eabi-*`)
Build automation	GNU Make
Version control	Git, hosted remotely on Github
Emulation	QEMU

Table 1: Tools used by the project

The project uses the C language both due to its familiarity as well as its ease-of-setup on the embedded environment, however there are times in such an environment, and especially for operating systems development, that assembly is more suitable [20], and was thus opted for instead. As already discussed, the GNU Embedded Toolchain has been used to target the ARM architecture for which the project is built.

The GNU Embedded Toolchain consists of several utilities, all of which are used at various stages in the project's development: `arm-none-eabi-gcc` is responsible for compiling the C and ARM assembly files into object files; `arm-none-eabi-ld` is an embedded platform-independent linker, used to link multiple object files into a single `.elf` file; `arm-none-eabi-objcopy` converts the resultant `.elf` file into a binary (system executable, or `.img`) file; `arm-none-eabi-objdump` provides helpful debugging information about the final executable file (for example, used as a disassembler it presents the kernel image in assembly form); and `arm-none-eabi-gdb` provides the familiar interface of the GNU debugger.

As the project increased in complexity, especially with the addition of multiple source files, GNU's Make utility was used to automate the build process. This involved becoming familiar with concepts regarding rules, with GNU's online manual page [21] and [22] being of particular help.

Throughout its early stages, the project operated solely in an emulated environment, simply as it provided quicker feedback with respect to testing, which is already limited and slow in such an environment. For this purpose QEMU was consequently used. However, due to its lack of simulation of a system timer, an important aspect when programming interrupts and process scheduling, focus had to eventually be shifted towards operating on real hardware, limiting QEMU's influence on the project as a whole.

3.3 System Overview

The project presents an operating which is capable of booting on a Raspberry Pi 1 Model B+ and running (with reduced functionality) on an environment emulating the Raspberry Pi 2 Model B, and initialises a C runtime environment. It provides a mechanism for memory management, first in the form of determining the total memory available to the system, and then in dividing this up into 4kiB pages. A 1MiB section of "heap" memory is also set up, and an interface similar to the C standard library's `malloc()` and `free()` is implemented to manage this.

[2]Contrasted to empty, which points to the next free location, i.e. the address at which the next item will be stored

Visual feedback is provided in the form of output to HDMI and, in the virtual environment, output to serial via the UART peripheral. The mechanism for communication with the GPU required to do so is set up and managed via the mailbox peripheral, and output via HDMI is achieved by requesting and providing relevant data (dimensions, bits per pixel, etc.) to the framebuffer. User interaction is so far only achieved in the virtual environment, again doing so via UART, with real-world interaction via USB proving to be a significant source of challenge throughout development.

The project sets up interrupts and exceptions by providing various exception handlers. This is with the main aim of interacting with the system timer, a peripheral capable of generating interrupts every set amount of time, for use in process scheduling. It also provides a somewhat intuitive interface for registering new "types" of interrupts (further covered in Section **??**). Processes are implemented and stored using the standard job- and ready-queues, and a simple interface for creating new threads of execution is provided.

With process scheduling being one of the main systems designed for ease of understanding and extensibility, the project provides a simple interface for using and programming custom process schedulers, with all code required contained entirely in one file dedicated to such a task. Taking into account the added complexity brought on by concurrency, synchronisation is achieved by implementing both spin-locks and mutual exclusion locks. Finally, a simple and somewhat rudimentary form of inter-process communication is provided, making use of the "shared memory" model. User interaction via USB keyboard is attempted and provided as a compile-time option, and makes use of a statically-linked library provided by Rene Stange's project `USPi` [23].

Future work will be concentrated on achieving user-interaction via USB keyboard and using this to implement a shell/command interpreter. A final important system to implement is filesystems and permanent storage, for which interaction with the Embedded

References

[1] "Downloads." `https://www.raspberrypi.org/downloads/`. Page accessed: 2018-11-22.

[2] "Baking pi - operating systems development." `https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/index.html`. Page accessed: 2018-11-22.

[3] "Building an operating system for the raspberry pi." `https://jsandler18.github.io/`. Page accessed: 2018-05-17.

[4] B. Pfaff, "Pintos." `https://web.stanford.edu/class/cs140/projects/pintos/pintos.html#SEC_Top`, December 2009. Document accessed: 2018-08-24.

[5] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Pearson, 3 ed.

[6] E. Helin and A. Renberg, "The little book about os development." `https://littleosbook.github.io/`, January 2015. Document accessed: 2018-11-22.

[7] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, ch. 1: Introduction. Wiley, 9 ed., 2014.

[8] "Raspberry pi documentation: Bcm2836." https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/README.md. Page accessed: 2018-10-08.

[9] "Understanding the raspberry pi boot process." https://wiki.beyondlogic.org/index.php?title=Understanding_RaspberryPi_Boot_Process. Page accessed: 2019-04-13.

[10] "Github: raspbrerypi/firmware." https://github.com/raspberrypi/firmware. Page accessed: 2019-04-13.

[11] "Raspberry pi: Level of hackability of raspberry pi." https://raspberrypi.stackexchange.com/questions/7122/level-of-hackability-of-raspberry-pi/7126#7126. Page accessed: 2018-11-15.

[12] "Raspberry pi: What bios does raspberry pi use?." https://raspberrypi.stackexchange.com/questions/8475/what-bios-does-raspberry-pi-use. Page accessed: 2018-11-15.

[13] "Gnu-rm downloads." https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads. Page accessed: 2018-11-18.

[14] "arm-none-eabi-gcc 8.2.0-1." https://www.archlinux.org/packages/community/x86_64/arm-none-eabi-gcc/. Page accessed: 2018-11-18.

[15] ARM, *ARM1176JZF-S Technical Reference Manual*, 2009. Revision r0p7.

[16] "Arm developer suite." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.ads/index.html. Page accessed: 2019-04-20.

[17] ARM, *ARM Cortex-A Series: Programmer's Guide*, 2013.

[18] Broadcom Corporation, Broadcom Europe Ltd., 406 Science Park, Milton Road, Cambridge, *Broadcom BCM 2835 ARM Peripherals*, 2012.

[19] G. van Loo, *Broadcom BCM 2836 ARM Peripherals*. Broadcom Corporation, Broadcom Europe Ltd., 406 Science Park, Milton Road, Cambridge, 2014.

[20] "C: Things c can't do." https://wiki.osdev.org/C#Things_C_can.27t_do. Page accessed: 2018-08-30.

[21] "Gnu make: Writing recipes in rules." https://www.gnu.org/software/make/manual/make.html#toc-Writing-Recipes-in-Rules. Page accessed: 2019-04-20.

[22] "Make: How to use variables." https://ftp.gnu.org/old-gnu/Manuals/make-3.79.1/html_chapter/make_6.html. Page accessed: 2019-04-20.

[23] "rsta2: A bare metal usb driver for raspberry pi written in c." https://github.com/rsta2/uspi. Page accessed: 2019-04-22.

Appendix A Appendix A: source code