University of Beira Interior

Department of Computer Science

Nº 52 - 2014: ***RTEMS - Real-Time Executive for Multiprocessor Systems***

Authored by:

André Marques

Supervisor:

Dr. Paul Andrew Crocker, Ph.D.

July 2014

Acknowledgments

I would like to express my deep gratitude to Dr. Paul Crocker, my supervisor, for his patience, enthusiastic encouragement and support whenever requested. His guidance and incentive were truly invaluable and I am honored to have had the opportunity of working with him.

I would also like to extend my thanks to the SEGAL laboratory for lending a Raspberry Pi during the development of this project, without which this project would not be possible.

Finally, I want to thank my parents and closest family for their support and encouragement throughout my studies.

Contents

List of Figures

Listings

Acronyms

RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real Time Operating System
OAR	On-Line Applications Research
API	Application Programming Interface
NASA	National Aeronautics and Space Administration
ESA	European Space Agency
POSIX	Portable Operating System Interface
SMP	Symmetric MultiProcessing
I/O	Input/Output
IOCTL	Input/Output Control
DMA	Direct Memory Access
IMFS	In Memory File System
CPU	Central Processing Unit
USB	Universal Serial Bus
SOC	System on a Chip
GPU	Graphics Processing Unit
RAM	Random-access memory
UART	Universal Asynchronous Receiver/Transmitter
EMMC	External Mass Media Controller

FAT	File Allocation Table
MMU	Memory Management Unit
DSB	Data Synchronization Barrier
DMB	Data Memory Barrier
ISB	Instruction Synchronization Barrier
BSP	Board Support Package
SD	Secure Digital
SPI	Serial Peripheral Interface
SDSC	Secure Digital Standard Capacity
SDHC	Secure Digital High Capacity
SDXC	Secure Digital Extended Capacity
SDIO	Secure Digital Input Output
MMC	Multi Media Card
CRC	Cyclic Redundancy Code
OCR	Operating Condition Register
CID	Card IDentification
CSD	Card Specific Data
RCA	Relative Card Address
SCR	SD card Configuration Register

Chapter 1

Introduction

In the last few years a number of small embedded computers have appeared on the market to become wildly popular in both the educational and scientific/engineering communities. One of those systems is the Raspberry Pi, which at this point lacks a proper real-time operating system that can be used with it.

Each embedded system is unique because they are, by nature, specialized systems that can interact with the real world. An operating system must then be able to communicate with the system's connected peripherals for it to be of any use, and this is done through a device driver.

1.1 Context

RTEMS is an open source full featured RTOS that supports a variety of open API and interface standards, targeted for embedded systems. The RTEMS project is managed by the On-Line Applications Research (OAR) corporation with the help of the RTEMS user community and serves as the base operating system for many applications with real-time needs: space, defense, medical, industry, aviation, and more. It is the main open source RTOS used by National Aeronautics and Space Administration (NASA) and European Space Agency (ESA) for their space missions, and Portugal has the only RTEMS centre outside the United States of America, which is managed by Edisoft S.A..

The RTEMS operating system is mainly used with very expensive hardware, so it can be hard to build and maintain a community around the project if only a few people can run it. In fact, most applications using RTEMS are separate and closed projects, which usually do not contribute to the main RTEMS project. For that reason, one of the RTEMS project goals is to be as user/developer friendly as possible. RTEMS has been ported to many hardware platforms and supports a multitude of processor architectures and hardware emulators, but has missed the recent popular hardware targets, such as the Raspberry Pi.

1.2 Motivation

During my undergraduate studies I feel that I have been introduced (even if briefly) to all aspects of computing from a software point of view, but my main interest within computing always was operating systems. However, during the course I did not had the chance of working with a RTOS or an embedded system, and that motivated me to work in that domain for my graduation project.

1.3 Objective

This project introduces RTEMS, a real-time operating system that is currently being ported to the Raspberry Pi, and proposes a device driver for its SD memory card interface, which allows RTEMS to store data persistently on a common SD memory card. It also describes a test case for the RTEMS POSIX API that has already been included in the official RTEMS tree.

1.4 Document Outline

In order to describe the work that was developed during the project, this document is structured as follows:

1. Chapter 1 – **Introduction** – introduces the project and its objectives;
2. Chapter 2 – **Real-Time Executive for Multiprocessor Systems** – introduces the Real-Time Executive for Multiprocessor Systems (RTEMS) operating system, RTOS and embedded systems;
3. Chapter 3 – **Rename POSIX Function Unit Test** – introduces the RTEMS testing suite, the POSIX specification and describes the unit test developed during this project for the POSIX function `rename`;
4. Chapter 4 – **Device Driver Development** – introduces the concept of device drivers, starting from what a device is to the underlying concepts of developing a device driver;
5. Chapter 5 – **Raspberry Pi** – introduces the Raspberry Pi platform;

6. Chapter 6 – **Secure Digital Protocol** – introduces and describes the SD standard and protocol;
7. Chapter 7 – **Secure Digital Device Driver** – describes the device driver developed for SD memory cards under RTEMS and the Raspberry Pi;
8. Chapter 8 – **Conclusions and Future Work** – presents the project conclusions and some ideas for future work on this project.

Chapter 2

Real-Time Executive for Multiprocessor Systems

Real-Time Executive for Multiprocessor Systems (RTEMS) is a real-time (refer to section 2.1) executive [1] targeted towards embedded systems (refer to section 2.2).

It started as an United States Army project in 1988 developed by the OAR Corporation [2] and is now a community-driven open-source project, led by a steering committee with representatives not only from OAR but also from the RTEMS community.

This chapter will introduce the RTEMS operating system, focusing on its the relevant features to this project. It will define what a RTOS and an embedded system is, describe some RTEMS applications and features as well as the RTEMS internal and application architectures. At the end of the chapter there is a brief look at the RTEMS development environment, namely at its directory structure and how the RTEMS source code is compiled.

2.1 Real Time Operating System (RTOS)

A Real Time Operating System (RTOS) is a computer system that operates within strict time requirements for the execution of certain tasks.The computing results should not only be correct, but also need to be produced within strict time constraints [22].

These systems may also be mission or safety-critical, which work under the assumption that any failure may have catastrophic consequences, such as:

[1]May also be called kernel or operating system.

[2]At the time RTEMS meant *Real-Time Executive for Missile Systems*.

- Death or damage to people;
- Property damage;
- Financial losses.

Each task on a RTOS is given a deadline by which the task should have completed, and the system may or may not permit a task deadline to be missed.

A real-time system can be classified as:

- Hard real-time - The system must not miss any deadline. An early or late answer is a wrong answer;
- Soft real-time - The system may miss one or several deadlines, but at the cost of performance loss, missed user opportunity or some other penalization.

Hard real-time systems are found in safety-critical systems, such as:

- Weapon systems;
- Pacemakers;
- Anti-lock car brake systems.

Where soft-real time systems, which only provide that a critical real-time task will receive priority over other tasks until it finishes, can be found, for example, in real-time multimedia servers.

RTOS are usually embedded in specialized systems that can interact with the real world to provide real-time capabilities.

2.2 Embedded Systems

An embedded system can be defined as any computer system that is built into a larger system [17] [22]. Depending on the system they may require different levels of user interaction and knowledge.

Embedded systems are used, for example, on:

- Cameras;
- MP3 players;

- Cellular telephones.

As for RTEMS, it runs mainly in two sub-categories of embedded systems:

- Deeply embedded systems - Systems that work with little or no operator intervention and that should, ideally, go completely invisible and unnoticed by the user (for instance, an anti-lock brake system on a car);

- Real-time embedded systems - Systems that are driven by and must respond to real world events, which at the same time must adhere to rigorous environment requirements. RTEMS acts as the provider of the real-time capabilities to the system.

In the anti-lock brake system on a car example, each wheel has a sensor detecting how much sliding and traction are occurring, and each sensor continually sends its data to the system controller. Based on the received information, the controller orders the braking mechanism in each wheel how much braking pressure should be applied. All this process goes completely unnoticed by the car driver [22].

Deeply embedded systems are also used in space, where direct user intervention is impossible. This type of system is usually also real-time, so it comes as no surprise that one of the major applications of RTEMS is on space systems.

2.3 Applications

The RTEMS operating system has applications in a variety of domains, such as communications, medical, space and aviation, scientific, robotics, military and industrial. Some specific applications of RTEMS include:

- MITRE Centaur Robot - A modified off-road vehicle designed to follow other vehicles, using a PowerPc embedded board running RTEMS to control all the real-time input. It has many potential applications in the military, such as supply convoys;

- EPICS (Experimental Physics and Industrial Control System) is a set of software tools to provide a software infrastructure to control and operate devices such as particle accelerators, lasers and major telescopes. Here RTEMS is used, for example, to control the I/O servers of a laser beam;

- ESA Galileo - RTEMS is also running on the ESA satellites that will compose the european GPS (Global Positioning System) - Galileo.

2.4 RTEMS in Portugal

In Portugal, Edisoft is the main company using RTEMS. Edisoft works with defence and security systems as well as aeronautics and space systems, some of which using an in-house version of RTEMS.

Edisoft has also reached an agreement with OAR to create the *RTEMS Centre*, a project sponsored by ESA under the scope of the ESA-Portugal Task Force Protocol [2] [10]. The RTEMS Centre is a support and maintenance center with two main purposes:

- Design, development, maintenance and integration of tools for the RTEMS components that are of interest to ESA;
- Creation and maintenance of technical competences and support site for RTEMS in Europe.

2.5 Features

RTEMS provides a high performance environment to a wide array of applications, as shown in section 2.3, through a comprehensive set of features [18].

Some of those features include:

- Multitasking capabilities;
- Event-driven, priority-based, preemptive scheduling (with optional rate monotonic scheduling);
- Inter-task communication and synchronization;
- Priority inheritance;
- Responsive interrupt management;
- Dynamic memory allocation;
- High level of user configurability.

Over the next sections it will be described how some of these features are organized and structured, to ensure maximum portability across the many different hardware architectures supported by RTEMS.

2.6 System Architecture

This section will describe how RTEMS components are structured for use by a real-time application, as well as the application architecture.

2.6.1 Internal Architecture

RTEMS has, conceptually, three layers: hardware support, kernel and the APIs [10]. The hardware support and kernel layers compose the system itself, while the APIs are used to develop system applications by an end user. A detailed scheme of the system architecture can be seen in figure 2.1.

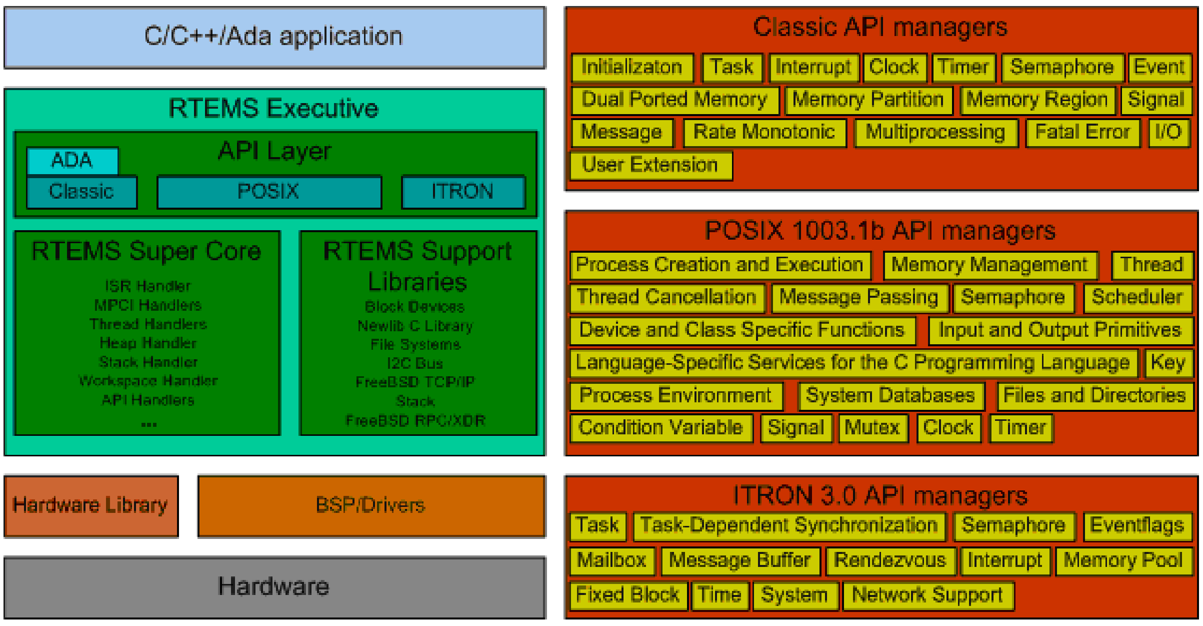


Figure 2.1: RTEMS architecture [10].

The hardware support layer contains the processor and board dependent logic, which permits to separate (at a logical level) the hardware from the software. The executive interface presented to an application is formed by grouping the systems directives into logical sets called *resource managers*. The system functions used by all resource managers are provided by the executive core (the kernel layer), which include scheduling, dispatching and object management.

Apart from the kernel directives, others are available through the available APIs:

- Classic API - Is the original RTEMS API which is based on the *Real Time Executive Interface Definition* (RTEID) RTOS standard (see figure 2.2);
- POSIX 1003.1b API - Implements the real-time extensions of the POSIX 1003.1 standard;
- Industrial The Real-time Operating system Nucleus (ITRON) API - Implements the Japanese open standard for RTOS.

In figure 2.2 the kernel layer corresponds to the *RTEMS Core*, while the surrounding resource managers build up to create the classic API.

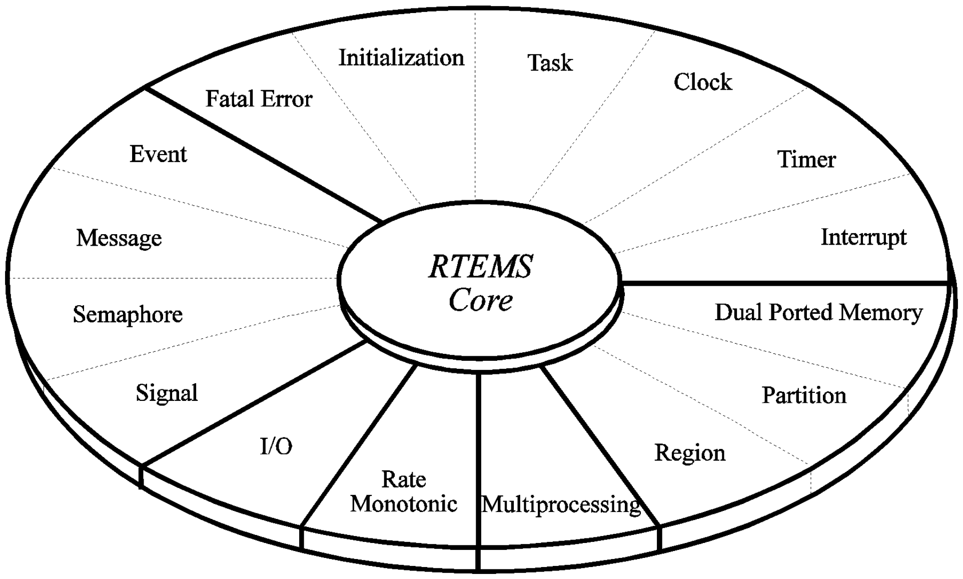


Figure 2.2: RTEMS classic API layered architecture [18].

The classic API resource managers:

- Initialization - Responsible for initiating and shutting down RTEMS;
- Task - Provides a set of directives to create, delete and administer RTEMS tasks;
- Interrupt - Manages the externally (to the system) generated interrupts, allowing quick interrupt response times by providing the ability to alter task execution through preemption;
- Clock - Provides time support to the system, including date and time as well as clock ticks;

- Timer - Allows the creation of timers that may be used to schedule/fire certain tasks after a given period of time;
- Semaphore - Provides synchronization mechanisms, such as counting semaphores or mutual exclusion;
- Message - Provides message queues that can be used for inter-task communication and synchronization;
- Event - Allows a task to generate/receive a notification to/from another task when a certain condition is met;
- Signal - Provides asynchronous communication capabilities;
- Partition - Dynamically allocates memory to create fixed-size memory partitions;
- Region - Dynamically allocates memory to create variable-sized memory partitions;
- I/O - Provides a mechanism to access and manage device drivers;
- Fatal error - Processes all fatal and irrecoverable errors, so the system has a chance of recovering;
- Rate monotonic - Provides tasks that are to be executed periodically;
- Multiprocessing - Allows the creation of global objects (tasks, queues, events, signals, semaphores and memory blocks) on multi-processor systems, that can run on a processor and accessed from another. RTEMS can determine where the object is running and performs the required actions to access it, allowing the entire system (both hardware and software) to be viewed logically as a single system.

The interface between the RTEMS system and an application for a specific hardware platform is a Board Support Package (BSP), which comprises the hardware (Central Processing Unit (CPU), board or peripherals) dependent code and links it with the RTEMS executive.

2.6.2 Application Architecture

RTEMS is designed to act as a bridge between the application code and the hardware. As shown in figure 2.3, RTEMS solve most hardware dependencies through its device drivers, and provides a general mechanism (a *Standard Application*

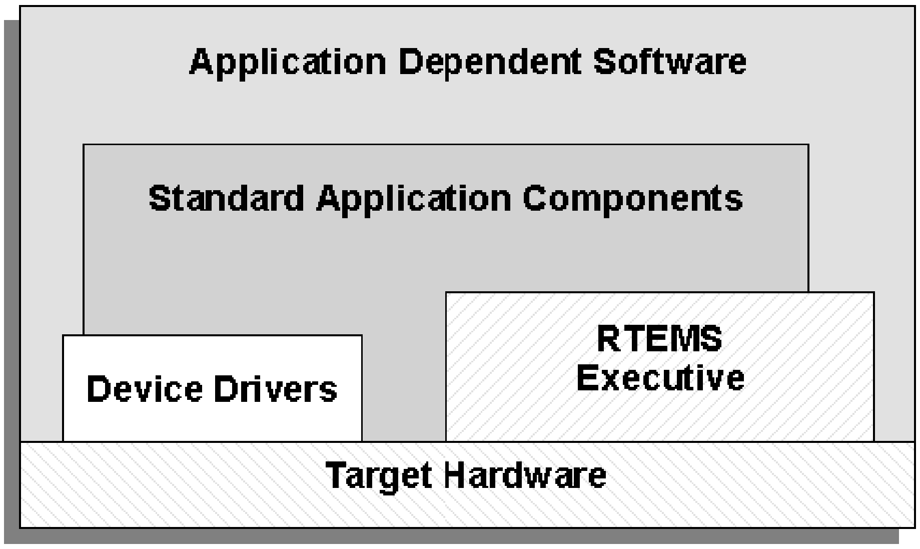

Application Dependent Software
Standard Application Components
RTEMS
Executive
Device Drivers
Target Hardware

Figure 2.3: RTEMS application architecture [18].

Component) to the application code to access the hardware. This allows code to be reused across different real-time projects.

One example of a standard application component is the RTEMS I/O interface manager, which provides to a user application all the necessary system directives and device drivers for a wide range of I/O capable hardware, leaving only the application dependent software to be developed.

2.7 Directory structure

The RTEMS system is distributed in the form of source code, however no significant support tools are provided [19], for instance cross compilers and all the associated tool chain for a particular architecture must be obtained separately. The source code is distributed and organized across several directories based upon functionality as well as hardware dependencies. This minimizes non-portable code and makes the process of adding support to a new CPU or target architecture very simple and conceptually easy to understand.

The RTEMS directory structure is designed to:

- Promote the development of modular components;
- Isolate processor and target dependent code, while at the same time allowing common code to be reused and shared across multiple processor and target boards;

- Allow localized compiling, so different users may be compiling different parts of the system for different target hardware. This is possible because RTEMS is compiled outside the source code tree, which by itself is different to most undergraduate software projects.

Taking RTEMS-ROOT as the root directory of the RTEMS source code, the following directories are of special interest to this project:

- RTEMS-ROOT/c/src/lib/libbsp/arm/raspberrypi - This directory contains the Raspberry Pi BSP code, and where the SD card driver will reside;
- RTEMS-ROOT/cpukit/libblock - Contains the support code for block devices, including I/O primitives. The libblock code is used by the Raspberry Pi SD card driver to interface with the card file system and the I/O primitives to allow the driver to be easily accessed by a RTEMS application;
- RTEMS-ROOT/testsuites/fstests - The RTEMS file system test case directory, which contains the test case created under this project and explained in detail in chapter 3.

2.8 Compiling the Source Code

RTEMS uses chains of makefiles which are compiled using the GNU automake and GNU autoconf tools. Each section of the source tree contains a *configure.ac* file that takes care of any necessary dependencies, such as header files or if a compiler is available. Each directory on the source tree has a *Makefile.am* file that is used with the section *configure.ac* by automake to generate a *Makefile.in*, the real makefile file that is used to compile all the code contained in that directory. This allows the developer to compile only the needed portions of the system at any given time.

2.9 Conclusions

This chapter introduced the RTEMS system focusing on its features and applications, giving a brief notion of the development environment, and of what a RTOS and an embedded system is. These concepts will be useful for the next chapter, which will focus on the RTEMS test framework and on a new test case for the POSIX API.

Chapter 3

Rename POSIX Function Unit Test

Just like any complex and dynamic software project, RTEMS has many outstanding development issues and ongoing projects. One such ongoing projects is the increase of the software test coverage through as many contributions as possible. Since one of this project goals was to be relevant to the RTEMS project and also to get to know and to become known amongst the RTEMS community of users and developers, a test case was chosen as the first contribution. The RTEMS POSIX API implementation of the rename directive had no previous test case to check its conformance to any POSIX specification, so a test case was made to check the conformance with POSIX.1-2008 [14], the latest POSIX specification for core services.

This chapter will introduce the RTEMS test suite and briefly the POSIX standard, focusing on the POSIX.1-2008 specification. The chapter then proceeds to detail the test case created for the rename directive, ending with a discussion of the results.

3.1 The RTEMS Test Suite

Every RTEMS test case is located in the *testsuites* directory at the root of the RTEMS source code, organized by category:

- fstests - File system test suite. The test case created during this project is located in this directory;
- libtests - Library test suite;
- mptests - Multiprocessor test suite;
- psxtests - POSIX API test suite;

- psxtmtests - POSIX timing test suite, to measure the POSIX directives execution time and other benchmark and performance tests;
- samples - Sample RTEMS applications;
- smptests - Symmetric MultiProcessing (SMP) test suite;
- sptests - Single Processor test suite;
- tmtests - Timing test suite, to measure the RTEMS directives execution time and other benchmark and performance tests;
- support - Contains support software and headers available to any test case. Each test category directory may have its own support directory, which will have the support files to that specific category of tests.

To create a new test case a new directory must be created at the appropriate test category directory, and should contain:

- Makefile.am - automake makefile so the test can be compiled;
- *test_name*.doc - The test documentation;
- *test_name*.scn - The latest test output/results;
- test source files - The actual test code.

Each test case must print a message both at the start and at the end of the test case. During the course of this project RTEMS started a basic test API (rtems/test.h header file), which at this time just prints the test case start and ending message in a standardized fashion, and provides a stub function that will permit printing profiling reports in the future.

3.2 Portable Operating System Interface

Portable Operating System Interface (POSIX) is a collection of standards maintained by the *Institute of Electrical and Electronics Engineers*(IEEE), which comprise a set of specifications for different aspects of an operating system behavior, primarily UNIX-based systems [14] [22]. Standardization is important to ensure that the same system call behaves the same way on any compliant system.

For a system to be POSIX-compliant it needs to implement the POSIX core standard: the POSIX.1. RTEMS provides its POSIX implementation [20] through its POSIX API, which also implement some POSIX extensions, mainly for threads.

3.2.1 POSIX.1-2008

A POSIX specification defines how a series of system calls must behave in all thinkable situations, through a series of conditions and constraints. For each directive the specification defines:

- The header file where the directive must be defined;
- Function parameter types;
- Detailed description using an if-then approach to describe the conditions and constraints to the function behavior;
- Return values;
- Error situations, as well as the right error codes to be set.

Some directive specifications group directives with very similar purposes, such as renameat which final goal is the same as rename.

3.3 The Rename Specification Test Case

The rename directive shall change the name of a file.

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Listing 3.1: Rename directive prototype.

In the rename directive prototype (listing 3.1), the `old` argument must point to the path name of the file to be renamed, and the `new` argument must point to the new file path name. This means that rename should also be able to move files across directories. If the file passed through `new` exists, `old` shall become `new` and `old` must be deleted.

3.3.1 The Test Case

The created test case is at *testsuites/fstests* under the name *fsrename*. This test it file system independent, so it can be tested with any file system. It must, however, be executed on a file system. Only one file system has been tested with *fsrename*, the Mounted In Memory File System, or MIMFS, under the name *mimfs_fsrename*. This secondary test case just initializes and mounts the In Memory File System (IMFS) and calls the *fsrename* test case.

The structure of the created test case divides the rename POSIX specification into 8 logical sections, each tested through a test function. Each function starts by declaring any needed variable or structure, and sets a test working directory.

For each part of the test the necessary files and directories are created from scratch, so it does not depend on previously used files, thus avoiding error chains. When some functionality has been tested, all the generated files and directories are deleted, and when the function ends it deletes the working directory as well.

The following are the test functions used by the test case.

symbolic_link_test

Tests symbolic link renaming. A symbolic link is a file on its own that points (links) to another file, which may or may not exist. This function tests that rename:

- Always operates on the symbolic link itself and never touches the file it links to;
- Gives an ELOOP error code if either new or old point or contain a symbolic link loop.

same_file_test

Tests same file renaming. It also tests with hard links, which are just aliases to existing files, so there can be several files which are in fact the same file.

This function tests that rename does nothing if old and new are the same file, either because they have the same file path or because the both point to the same file on the disk (through an hard link).

directory_test

Tests directory renaming, including the rename of files within directories protected with a sticky bit. A sticky bit on a directory defines that only the file or directory owner (or the root user) can rename or delete a file on that directory. This function tests that rename:

- Gives an ENOTDIR error code if trying to rename a directory with a file;
- Gives an EISDIR error code if trying to rename a file with a directory;

- Gives an EINVAL error code if trying to rename a directory with an ancestor directory;
- Gives an EEXIST or ENOTEMPTY error code if trying to rename a directory with a non empty directory;
- Can rename a directory with an empty directory;
- Gives an EMLINK error code if trying to rename a file which would move it to an already full directory (the directory link count is already LINK_MAX);
- Gives an EPERM or EACCESS error code if trying to rename files within and to a directory with a sticky bit set.

arg_test

Tests the function behavior with certain function arguments. Rename must:

- Rename an existant old file with the name of a non existant new file;
- Give an ENOENT error code if the file/directory to be renamed (or part of its file path) does not exist;
- Give an ENAMETOOLONG error code if new is longer than NAME_MAX.

arg_format_test

Tests the function behavior with certain argument formats. Rename must:

- Give an EINVAL or EBUSY error code if any of the arguments contain a final component that is a dot (current directory) or dot-dot (ancestor directory);
- Give an ENOENT error code if any of the arguments is an empty string.

write_permission_test

This function tests the renaming of files on a directory with no write permission.

Rename must give an EACCES error code if either old or new points to a file on a write protected directory.

search_permission_test

This function tests the renaming of files on a directory with no search/execute permission.

Rename must give an EACCES error code if either old or new points to a file that is on a search/execute protected directory.

filesystem_test

Tests file renaming across different file systems.

Rename may give an EXDEV error code if old and new are on different file systems.

3.3.2 Test Case Unchecked Cases

The produced test case does not test some details of the specification, because these details are either unable to be tested with an emulator, too complex to setup or because they are already tested by existing test cases.

The test case does not check the following error cases:

- EIO - physical error on the hard disk;
- ENOSPC - no space left in the new file path;
- EROFS - renaming on a read-only file system, as it is already covered by the *testsuites/fstests/fsrofs01* test.

The test also does not check the following functionalities:

- File system lock during the renaming operation - for a file to be renamed after another, that file needs to be deleted before the first file can have its name, but in the meantime the file name must remain visible to the system;
- Free file disk space when link count reaches 0 - if rename deletes a file during its operation, and the file link count reaches 0 (no reference in any file system to that file after rename deletes it), the space occupied by the file shall be freed and no longer accessible. This would depend on the statvfs directive, which provides disk statistics, but it is not implemented for the RTEMS IMFS file system.

3.4 Results

The full test results can be seen at appendix A. They reveal that the current RTEMS implementation of the rename directive fails at:

- Renaming symbolic links;
- Giving an ELOOP error at symbolic link loops;
- Renaming a file with itself;
- Renaming a file with itself through an hard link;
- Giving an ENOTDIR error when renaming a directory with a file;
- Giving an EISDIR error when renaming a file with a directory;
- Giving an EINVAL error when renaming a directory with an ancestor directory;
- Renaming an empty directory with another empty directory;
- Giving an EPERM or EACCES error when renaming files on a directory with sticky bit set;
- Giving an ENOENT error when renaming a non existant file or file path;
- Giving an EINVAL or EBUSY error when renaming file paths which end in dot or dot-dot or empty string file paths;
- Giving an EACCES error when renaming files on a write protected directory;
- Giving an EACCES error when renaming files on a directory with no search/execute permissions with a file on a directory with search/execute permissions.

3.5 Conclusions

This chapter introduced the RTEMS test suite, the POSIX standards and described the implementation of a test case for the rename directive, under the RTEMS POSIX API. With a test case in hand the rename directive may then be made POSIX conformant by solving the components that failed the test. This test case can also be easily extended to the renameat directive. During the development of this test case it was also noted that the IMFS file system does not have an implementation for the statvfs directive. The test developed during this project and described in this section has already been merged into the official RTEMS code base [8].

Chapter 4

Device Driver Development

Drivers are an important part of an operating system, as they make it possible to connect and use hardware on a computer, in a way that hides the underlying complexity of how a specific hardware device works. A device or peripheral device can be, for instance, a keyboard, a printer, a memory card or some unique custom built device that sweeps the dust on a desk. Device driver development is and will continue to be important as devices get more diverse and complex.

This chapter will introduce device drivers and the important concepts that are needed to develop one, such as the notion of execution spaces, concurrency, driver operations, time, memory, device communication and file systems while focusing on a specific type of device made of memory blocks: block devices.

4.1 Device Drivers

Computing needs a physical medium to happen - the hardware - or at the very least, it must have a physical interface for it to be of any value. The hardware can be viewed as a body, and just like us humans, the hardware does not work without a brain to control it. That is the role of the operating system: to operate the hardware. But what happens if we attach a new member to the body? Just like us, the operating system would need to be rewired to operate the new member. As far as this short analogy goes, the member we just attached to our body can be a keyboard, and the operating system (the brain) would need to be instructed on how to communicate and perceive the keyboard. This keyboard would be peripheral to our system, in the sense that the computer can still work without one, so it just extends the system capabilities.

A device, or peripheral device, is then some type of hardware that is connected to a system to provide some extra feature. However, this requires that the operating system running on that system can acknowledge and interact with the device, and for that it needs the right device driver. A device driver is a small part of the operating system that knows how to interact with a type of hardware, and device drivers can always be added to support new hardware: just as learning a new language.

A good device driver *should* provide a mechanism - access to the hardware capabilities and features - and not a policy that defines how those capabilities can be used [7]. A memory card driver, for instance, should show the card to the system as a contiguous array of data blocks. Its the role of a system application to provide policies, such as checking if a user has permissions to access a certain file, or if the file can be accessed directly or via a file system. By not enforcing any policy the driver can be used by any application, thus promoting code reuse.

A badly designed driver can also lead to security problems, as drivers usually run in a privileged environment. Care should be taken with buffer overflows, user/application input or user commands with potentially harmful results, such as formatting a memory card.

4.2 Kernel and User Space Drivers

The point of having an operating system is to provide programs with a consistent view of the computer's hardware, while keeping the programs operations independent and protected against unauthorized access to the system's resources [7]. A program may then run in two main execution environments:

- The kernel space, where a program can do anything it wants with the system resources;
- The user space, where resource access is constrained.

This is enforced through the processor, which can provide different operating levels, such as a *supervisor mode* for kernel space and a *user mode* for user space programs. Because a driver is also a program, this means that it can run from either execution environment, which may not seem intuitive at first since a driver needs to directly operate the kernel controlled hardware resources. This leads to the notion of an high and low level driver, where a low level driver interacts directly with the hardware in kernel space and provides a set of directives available to the user space, where an high level driver can use those directives to operate the hardware - to some degree - from user space.

4.3 Concurrency and Compiler Optimization

The problem of program concurrency appears when the processor switches programs in the middle of their execution and they rely on data that may be modified by other programs while they are waiting to execute again, leading to unexpected results. The compiler and even the CPU may also change the instruction execution order for the sake of performance optimization [7]. This problem is obvious with multi-core systems, where several programs can execute at the same time, but may also happen on a single processor system if a program or other entity can preempt the processor, such as an hardware interrupt from a device.

Each program has a context: the CPU register values, the process state and the program memory configuration [22]. When the CPU is preempted or scheduled to execute another program, it saves the current process context in memory and switches to the new program context. If both programs rely on a shared or global variable, context switching can introduce unexpected results. The compiler may also optimize the execution order of the program, such as memory access instruction or omission of seemingly unnecessary instructions, such as two consecutive reads. Because hardware is usually inflexible in the way it works, doing a read before a write or omitting a consecutive read will most likely lead to unexpected results. Context switching may also create changes in the execution order of instructions in the hardware, if more than one device is using the same memory registers at the same time.

To solve the concurrency problem the developer needs to ensure that their code is *reentrant*, meaning that it can be run in more than one context at a time.

4.4 Major and Minor Numbers

To use a certain device the operating system must have a driver that enables it to interact with the device hardware. However, multiple devices can be connected to a system that uses the same device driver. One example is an Universal Serial Bus (USB) controller driver, which may control a number of USB ports: the driver is the same to all ports, but each port is an independent device.

To manage this situation, most systems use a major and minor number approach, where each driver is given a major number, and every device is given a minor number that is specific for the driver it uses. This must be the first thing any driver code does at the start: register the driver major number and assigning a minor number to the device. While the major number must be unique in the entire system, minor numbers are driver specific. This allows a device to be uniquely identified, while at the same time associating it with a device driver.

To ensure that a driver major number is unique, RTEMS and most systems allow dynamic allocation of major numbers, where the system assigns the next greater major number that is not yet registered. Minor numbers, however, are managed by the driver itself, so it is up to the driver developer to manage the many devices that may use the driver at the same time and to assign minor numbers correctly.

4.5 Driver Operations

As seen previously in this chapter, a driver provides an interface to the system so it can communicate with a device. That interface is just a list of directives, or operations, that can be used to access the device hardware. Each type of device will have a set of specific operations, which may not make sense to use in others. For instance, a keyboard device driver probably will not need to provide a write operation to an output device such as a screen. Also, unique devices will also need to provide unique operations, so the operations that a device driver may provide can be as diverse as there are devices to connect to a computer, however some operations are usually provided:

- Initialize - This would be the first operation called when using a driver. It prepares the operating system for any needed resource (memory and disk space, for instance) and performs any required hardware initialization sequence or hardware checks;
- Open/Close - Depending on the device, this function could be used to open or close an hardware port (such as a CD tray), or be used as the initialize call (for open) and as an unmount call to unmount an hard disk from the system;
- Read/Write - As the names suggest, the driver would provide these operations so the system can do I/O with the device;
- Input/Output Control (IOCTL) - This would be the preferred way to implement any unique operation while using a common driver framework. The IOCTL operation allows the system to send requests to the driver, which can be defined by the driver itself, avoiding the creation of a new system call [22] [7].

The device driver created for this project is IOCTL based, which receives read and write requests to perform I/O operations on the device.

4.6 Time

Every action in the universe takes time, and hardware devices are no exception. Every device driver must take into account the time that the hardware takes to complete a certain task, for instance, the time it takes to write to an I/O register. If the CPU can execute thousands of instructions per second, but the device hardware takes 150 milliseconds to process a command, the device driver must wait at least 150 milliseconds before allowing the next command to be sent, otherwise the device may not be ready to execute the following commands if they depend on the first.

To create a delay on a device driver it is important to consider how a CPU clock tick compares with the speed of the device, because a small delay on the device can mean a long delay on the CPU [7]. Delays may be introduced through busy waits, where the CPU is occupied waiting and does not attend other processes, or through a timeout timer, which sleeps the process (yielding the processor to perform other tasks) until the timer ends.

4.7 Hardware Communication

A driver sits between the software programs and the hardware circuitry, and for that reason it must be able to communicate with both of them [7]. To control any peripheral device the system reads and writes to the device registers, which is done through I/O ports. These I/O ports can range from simple digital I/O hardware pins, to complex data buses such as the ones used on the USB protocol.

Each device has several memory registers, that can be accessed at consecutive memory addresses either in the memory address space or in the I/O address space depending on the device type. The difference between I/O registers and Random-access memory (RAM) is that I/O operations will have side effects on the device hardware, while RAM operations just store and retrieve values. As mentioned in section 4.3, both the compiler and the CPU may optimize memory accesses by using caches, which will not affect either the device or system's memory, or may reorder the sequence of instructions so that the memory access is faster. The problem is that while for RAM memory accesses this is a desirable feature, it does not work for hardware memory registers and can lead to some very hard to debug errors. Peripheral devices have strict execution orders for I/O operations, and these operations must happen on the physical memory registers, so no caching is allowed.

To solve the problem with compiler optimization and hardware reordering the usual solution is to use a memory barrier between the operations that must be visible to the processor in a particular order. Memory barriers, however, affect the driver performance, so they must only be used where needed. These operations, as any I/O operation, are highly processor dependent, because each CPU handles data differently, making the related source code platform dependent.

Another aspect of hardware communication is that while the CPU is aware of changes in the RAM memory, changes in the peripheral hardware memory registers are outside the CPU reach, so it must have a way of keeping with the hardware memory register updates [22]:

- Polling - the CPU checks the status of a memory register in the peripheral device periodically (every 100 milliseconds, for example);
- Interrupts - the hardware connects with the CPU through an interrupt-request line, which can be used by the hardware to signal the CPU of some change in the hardware memory registers or I/O operation.

Polling can be very inefficient, because each time the CPU polls the hardware and there is no change in the memory registers is time that could be used to do something else. The ideal is to use hardware interrupts, which will be introduced on the next section (4.8).

4.8 Interrupt Handling

Devices deal with the real world, and the notion of time can be very different between the system and the peripheral device, making it undesirable to have the system waiting for external events (read about polling on the previous section (4.7)).

An interrupt is just a signal sent by the hardware to get the CPU's attention [22] [7]. This is done through *interrupt-request lines* in the CPU, which are sensed after the execution of every instruction. This sensing can be seen as a super fast polling, since it happens within the CPU itself, and when it detects a signal it catches it and dispatches it to the corresponding interrupt handler.

Each processor has a limited number of interrupt lines, and for that reason two or more drivers may use the same interrupt line to connect with their device. Also, interrupt handlers run concurrently with other code, so interrupt handlers need to be designed with concurrency in mind. A driver must require an interrupt channel to the operating system before using it and release it when finished, and must register an interrupt handler for the assigned interrupt channel during driver initialization or opening (refer to section 4.5), before instructing the hardware to generate interrupts.

Interrupt handlers can not transfer data to or from user space (refer to section 4.2) as they do not execute in the context of a process. They also can not call any directive that would cause the handler to sleep, such as allocating memory, or locking a semaphore. The role of the interrupt handler is, then, to acknowledge the reception of the interrupt (by clearing the interrupt bit in the device memory register, although this step can be device specific) and perform read or write operations depending on the nature and purpose of the generated interrupt, or wake a sleeping process on the device (that may be waiting to receive some data).

An interrupt handler should also execute in the minimum time possible, so I/O operations with the device may be interrupt-driven. This type of I/O uses a buffer that is used by the interrupt handler to store data, that is then used by a process as if it was reading data from the device, and vice-versa.

4.9 Memory

Memory is central to a system's operation, be it RAM, caches or memory registers [22]. Any form of memory provides an array of words (a fixed-size chunk of memory that the processor can handle as a unit, which is specific to a processor hardware architecture), and each word has its own memory address. This memory addresses can be physical addresses in the real memory chip, or generated by the CPU. Therefore memory addresses can be organized in a physical address space, or in a logical address space, if CPU generated.

4.9.1 Memory Mapping and Direct Memory Access

Memory mapping and Direct Memory Access (DMA) are two ways through which an application can access a device's memory, and it is based on the concept of memory address spaces. Memory mapping a device means to associate a certain range of logical or physical (RAM) addresses to a device hardware memory registers, so that when an application accesses one of those CPU generated memory addresses it will be actually accessing the corresponding memory register on the device [7]. However, this mapping is done by the CPU, so each time an application writes to a memory address it must wait for the CPU to transfer that data to the corresponding address on the device. Direct Memory Access (DMA), as the name implies, refers to direct access to a device's memory. This requires an additional hardware mechanism on the system that can transfer I/O data directly from system to device memory, without having to wait for the CPU, removing computational overhead. A DMA transfer requires a buffer: a range of memory addresses that contain data to be sent to the device or to store data from the device. Because these transfers are done by the DMA hardware, the CPU does not know when a transfer has finished, so interrupts (refer to section 4.8) are used to signal it.

4.10 Block Devices

A block device is any device that permits access to independent, fixed-sized blocks of data, such as hard disks, memory sticks, CD-roms, and so on [22] [7]. A block has usually a length of 512, or any power of two number of bytes (depending on the architecture and file system used), which makes the smallest number of bytes that can be transferred.

These devices usually store file systems (refer to the section 4.11) which are used to access the device data, but direct access to the device is also possible, such as to install or repair a file system.

4.11 File Systems

A file system lives on a physical storage device where it defines how data is organized, keeping records of its tree of files and directories [22] [7]. The role of a file system is to define how the information is stored in the device, and this requires not a device driver but a different type of driver: a *software driver* which maps the low-level data structures in the device to high-level data structures usable by the operating system and other applications.

It is the file system that defines which information to store about each file on a directory, and the file system driver provides the system with the necessary directives to perform I/O operations on the file system. This file system driver, however, can not interact with the block device itself, as it only works with files, so it needs a device driver underneath to provide the data blocks that make the said file. For instance, if an application asks the operating system for the file “example.txt”, it sends the request to the file system driver, which will check the file records to get the block indexes that make up the file and finally it will request the block device driver to send the data on those blocks. Hence a device driver for a block device only does block I/O, and does not work with higher level data structures such as files or directories.

4.12 Block Device Drivers

A block device driver provides access to block devices (refer to section 4.10) through block I/O directives. Because it is responsible by the transfer of data between the device and the system, block device drivers are critical for performance [7]. A block device driver registers its device on the system and may be called in several situations, such as to partition the disk, install a file system, check a file system integrity or simply using a file system. All these operations look the same to the driver as it just performs block I/O, which directives it must make available to the operating system (refer to section 4.5).

The system may also have block request queues to keep track of which blocks the system needs, and may reorder the requests to improve the performance in non random accessible disks, through an I/O scheduler. An example of this type of optimization is when two contiguous physical disk blocks are scheduled to be fetched together. These optimizations may not be relevant at all if the access is really random, and the computing cost of getting any block is the same, such as with flash disks.

4.13 Conclusions

This chapter introduced device driver development targeted to block devices. It briefly describes what a device driver is and how it works, the concurrency issues that are raised, how to communicate with a device, memory management, file systems, time and, of course, block device drivers. These concepts will be needed for the following chapters, which will describe the device driver developed for this project.

Chapter 5

Raspberry Pi

The Raspberry Pi is a low cost, credit-card sized computer that can be used as a general purpose computer which also interacts with the outside world [12]. The goal is to provide a cheap and simple system that can be used not only as a learning platform but as a full featured portable system that can also be embedded in other systems.

This chapter will briefly describe the Raspberry Pi hardware features, why it is an interesting/relevant platform for the RTEMS project, how the Raspberry Pi boots, how to communicate with it and how its memory is organized.

5.1 Hardware Features

The Raspberry Pi is available in two models - A and B - which share most of the same hardware features. This project used a Raspberry Pi model B for its development and this section will describe some of the Raspberry Pi relevant hardware features to this project.

5.1.1 System on a Chip

A System on a Chip (SOC) is an integrated circuit that contains all the necessary components to run a computer. The Raspberry Pi SOC comprises on a single chip the Central Processing Unit (CPU), Graphics Processing Unit (GPU), Universal Serial Bus (USB) controller and Random-access memory (RAM) [12]. The specific SOC used on the Raspberry Pi is a *Broadcom BCM2835*.

CPU

The Broadcom BCM2835 uses an *ARM1176JZF-S* (ARM11 with ARMv6 instruction set) CPU with a clock speed of 700 MHz.

GPU

The Broadcom BCM2835 SOC provides a *Broadcom VideoCore IV* GPU, providing Open GL ES 2.0, hardware-accelerated OpenVG, and 1080p30 H.264 high-profile encode and decode. As a general purpose computer it can be connected to an High-Definition Multimedia Interface (HDMI) monitor or television screen, with a standard keyboard and mouse, and used to browse the Internet and play high-definition video. This also makes it a very interesting device to embed in other systems to provide graphics support.

Because the GPU documentation is proprietary, interaction with the GPU is done through a *mailbox* interface [11]. The mailbox interface is a register that has several channels ("mail accounts") for different resources on the board (to do power management or control the frame buffer graphics) , so a driver sends a buffer to the CPU with a request (an "email") to one of them, which the GPU fills with the requested data.

Memory

The Raspberry Pi SOC comes with 256 mega bytes of RAM for the model A, and 512 for the model B.

5.1.2 External Mass Media Controller Interface

The Raspberry Pi board provides an External Mass Media Controller (EMMC) interface, which is an embedded MultiMedia and SD card reader interface, compliant with the Secure Digital High Capacity (SDHC) specification [6] and the SD memory card specification [21] versions 3.00 [5]. The next chapter will detail both these specifications and the SD protocol.

5.2 Raspberry Pi and RTEMS

The RTEMS project is always looking for new target platforms to run on, but readily available boards are of special interest. A readily available board is a ready to use piece of hardware that just needs some software to run on top. These boards are interesting because they are built in a standardized fashion, thoroughly tested and commercially available.

The Raspberry Pi is one such board, with the added advantage of being extremely cheap. Porting RTEMS to the Raspberry Pi has benefits both to RTEMS, which usually runs on very expensive hardware, and to the Raspberry Pi, by taking a proven RTOS to a popular piece of hardware. Currently the RTEMS support for the Raspberry Pi is very limited and only provides an Universal Asynchronous Receiver/Transmitter (UART) console driver on polled mode (refer to section 4.7), which also accesses the CPU internal timer to control the data transfer operations. To help in the effort of porting RTEMS to the Raspberry Pi this project provides a driver for the card reader (EMMC) interface, which is described in section 5.1.2.

5.3 Booting the Raspberry Pi

The Raspberry Pi boots exclusively from a SD memory card, inserted on the onboard SD card reader on the Raspberry Pi board: the EMMC interface (refer to section 5.1.2). The SD card may, however, be used to boot the Raspberry Pi from another memory device, such as an USB disk or another SD card.

The bootable SD card needs to be formatted with a File Allocation Table (FAT)32 boot partition, which must contain the GPU firmware, a configuration file, the operating system or kernel to be loaded, and some other necessary files.

Most of the boot process is done by the GPU [1]:

1. First stage bootloader - Mounts the FAT32 boot partition on the SD card so the second stage bootloader can be accessed;
2. Second stage bootloader - Loads the GPU firmware from the SD card to start the GPU;
3. GPU firmware - Once loaded, allows the GPU to start up the CPU;
4. User code - The user defined kernel.

5.4 Communication with the Raspberry Pi

To interact with any type of computer a user needs to be able to send and receive data, which is usually done through a keyboard (to send commands) and a screen (to see the command results). However, as explained in section 5.2, the

RTEMS support for the Raspberry Pi only covers the most basic of communications through an UART interface. This interface uses a serial protocol where data packages are sent through a transmitter and received through a receiver, both controlled by a timer. This requires a terminal emulation software running on a host computer, such as the *Minicom* program on Linux, to be able to see the terminal running on the Raspberry Pi. Nevertheless, this only permits sending commands and receiving their output to an already running program on the Raspberry, which must be loaded on the SD card (refer to section 5.3). In practice this means that whilst developing an RTEMS application for the Raspberry Pi, each time the developer wants to test the application on the hardware he must remove the SD card from the Raspberry Pi to an host system with a SD card reader, transfer the newest version of the application and put the SD back in the Raspberry Pi to test it.

This process can create problems on the Raspberry Pi hardware overtime due to the constant SD card movement, so a solution was found to send applications through the UART interface using the *xmodem* protocol. This protocol enables sending files over a serial connection (as the UART), but it requires an application on both sides of the communication channel which use and implement the protocol. On a Linux host system minicom can provide xmodem transfers, but the Raspberry Pi also needs an application that implements the protocol. A bootloader that implements this protocol was found [23], and it was copied to the SD card so the Raspberry Pi could use it. The way the bootloader works is simple: it waits for a xmodem transfer, stores the sent program in memory (RAM) and then boots the Raspberry Pi with that program. This means that when power is off the sent program is lost, so the SD card contents do not change at any stage of the process. When the developer wants to send another program he just needs to disconnect and reconnect the power cable on the Raspberry Pi or the UART cable so that the bootloader restarts and waits for another xmodem transfer. This process takes approximately 2 minutes to send 1 mega byte of data due to the implementation of the xmodem protocol on the bootloader, which uses the default transfer block size of 128 bits instead of the maximum size of 1024 bits.

5.5 Memory Registers

As referred in section 4.9, the processor can generate memory addresses to create a logical address space. This can be done either by the CPU or GPU Memory Management Unit (MMU). On the Raspberry Pi the peripherals memory registers are mapped to the physical memory in the RAM by the VideoCore or GPU, which translate the addresses in the range of `0x20000000` to `0x20FFFFFF`.

Access to the peripheral memory register address space must be protected with memory barriers (refer to section 4.7) when more than one peripheral is being used, as the CPU may change peripherals at any time. This would make the data arrive out-of-order in the memory registers, so a memory barrier should be used before the first write to a peripheral and after the last read to a peripheral [5]. A memory barrier ensures that any explicit memory access done before the memory barrier is completed before executing the following memory accesses. The Raspberry Pi CPU provides some memory barrier instructions, in the for of co-processor assemblies [15] such as:

- Data Synchronization Barrier (DSB) - does not complete until all the previous instructions complete;
- Data Memory Barrier (DMB) - ensures that any explicit memory access after the DMB instruction only start when all explicit memory accesses before the DMB instruction complete;
- Instruction Synchronization Barrier (ISB) - flushes the pipeline in the processor, so all the following instructions are fetched from cache or memory once the ISB completes.

As for the register memory size, each register in the Raspberry Pi SOC is 32 bits wide. These are used to control the peripherals through bitwise operations in accord with the Broadcom BCM2835 ARM Peripherals datasheet [5].

5.6 Conclusions

This chapter introduced the Raspberry Pi and its features, how it boots, the current limitations of RTEMS for the Raspberry and and briefly describes how the memory is structured and accessed. It also describes the current RTEMS development process for the Raspberry Pi, and introduces the EMMC interface, which will be detailed in the next chapter.

Chapter 6

Secure Digital Protocol

Secure Digital (SD) cards are used in every type of mobile device today that requires portable flash memory. This popularity is due to their portability across many different systems and hardware, where a SD card may be used with any SD host controller (or card reader) on any operating system that provides a SD device driver.

This chapter will introduce the SD standard that makes this possible, explain what exactly is a SD card, how a device driver can access the card and how the communication with the card unfolds.

6.1 The Secure Digital Standard

The Secure Digital (SD) standard defines a memory or I/O card that can be used with a wide array of devices [3], including mobile phones, digital cameras, *Global Positioning System* (GPS) and embedded devices.

This section will introduce the SD memory card standard with a short reference to the Secure Digital Input Output (SDIO) card standard, which is outside the scope of this project.

6.1.1 The Secure Digital Memory Card

The Secure Digital (SD) memory card standard defines a card which purpose is to store data permanently, so every SD memory card is categorized by their storage capacity:

- Secure Digital Standard Capacity (SDSC) - Up to and including 2 *Giga Bytes* (GB) of storage;

- Secure Digital High Capacity (SDHC) - More than 2 GB and up to and including 32 GB;
- Secure Digital Extended Capacity (SDXC) - More than 32 GB and up to and including 2 *Tera Bytes* (TB).

Apart from the storage capacity, each SD card is also defined by their speed class and form factor, such as mini, micro or full size. Any SD card can be used with a host controller (card reader) compatible with its form factor, but the same card may report different performance levels (data transfer speeds) across different host systems. This happens because there are two stages though which the system's data must pass through before reaching the memory card:

- The device driver - Where the performance will depend on which versions of the SD standard the driver supports, and how they are implemented;
- The host controller - Each card reader only supports up to a certain version of the SD standard. This means that a recent card running on an older host controller will be limited to the capabilities supported by the host controller.

To the driver developer this poses the challenge of knowing exactly what the hardware (the card and host controller) supports, and then adjusting the data transfer operations accordingly. This is the only part of the data transfer process that a user/developer can control, as the SD card and host controllers have their own implementations of the standard and therefore operate on the data in a predefined and non configurable way.

Each SD card has a tiny microprocessor that is the only one that can directly operate on the card's flash memory, so when a system is interacting with an SD card it is in fact interacting with this microprocessor and not the flash memory itself. The reason behind this is the increasingly small size of the flash memory units, which makes direct access to the flash memory very card specific and therefore impractical to be done at the host controller or operating system level, as it would imply different device drivers on a system for every SD card model from any given manufacturer [21] [4].

As for the host controller hardware, it provides the operating system with a register set that can be used to interface with an inserted SD, SDIO or even Multi Media Card (MMC) card, as in the case of the EMMC interface used in the Raspberry Pi (refer to section 5.1.2). Each type of card requires a host controller standard to be supported/implemented on the physical host controller hardware, and in the case of both SD and SDIO cards this is handled by the SD host controller standard [6].

6.1.2 The Secure Digital Input Output Card

The Secure Digital Input Output (SDIO) card is defined in the SD standard to include the ability to use a SD card slot for more than memory cards. The concept is similar to the USB protocol, where an USB port may be used to access memory devices (such as USB memory sticks) but also other devices such as wireless receivers, fingerprint readers, and so on. A SDIO card is then a device that uses any standard SD card slot to extend the functionality of a device [3].

As stated in section 6.1.1, the SD host controller has the ability to control both SD memory cards and SDIO cards, but because it was not the objective of this project to support SDIO cards the provided driver ignores any inserted SDIO card.

6.1.3 Operating Modes

The SD standard defines that each SD card must implement two operating modes: the SD or native mode and the Serial Peripheral Interface (SPI) mode. The native mode uses a communication protocol defined by the SD Association, which also owns the protocol documentation. For that reason, a developer who wants to write a driver using the SD native mode must either pay a license to the SD Association to gain access to the full documentation of the standard, or use the simplified specifications which can be downloaded for free on the SD Association website [3]. As for the SPI mode, it is a secondary communication protocol designed to work with a SPI channel, however, the SPI standard only defines a physical link and not a complete data transfer protocol [21], so it must still use a subset of the native SD communication protocol and commands to work.

The advantage of the SPI mode is that its standard is completely open, so anyone can see its full documentation. It also allows to add a SD host controller to any device with a SPI interface, so no native SD card reader would be required. Nevertheless, using a SD card through a SPI bus implies a performance loss [21]. In the case of the Raspberry Pi it offers a native SD card reader, the EMMC interface (refer to section 5.1.2), and also offers a SPI interface but because RTEMS does not provide a driver to the SPI interface on the Raspberry Pi yet, this project implemented a SD card driver for the native protocol instead.

6.2 The SD Bus Protocol

To communicate with a SD card using the native mode (refer to section 6.1.3) the SD bus is used. This type of communication is based on command and data bit streams which start with a start bit and end with a stop bit [21]. To start an operation on a SD card a command is sent from the host to either a specific card (using the card address) or to all connected cards (a broadcast command). The card's response to the a command is stored on the host controller response registers, from where the host system can then retrieve and interpret this data.

Every card operation, such as sending commands, receiving responses or transferring data, is done using memory blocks. Each block ends with a Cyclic Redundancy Code (CRC) checksum code to protect it against transmission errors.

6.2.1 SD and Application Specific Commands

Both SD and application specific commands are identified by an index, and may be sent to all cards connected to a host controller, or to a specific card (each card is given an individual address by the host controller). Each command may also accept arguments (to tailor the command behavior as needed) and may lead to a card response (in the case of a request).

A SD command and its response compose a 48 bit block each which contains:

- One start bit;
- One direction bit (if the command is coming or going to the host controller);
- The command index;
- The optional command argument (if a command) or the card status (if a response to a command);
- The Cyclic Redundancy Code (CRC) code to protect against transmission errors;
- One end bit.

The SD protocol specifies for each command if it must be accepted by all SD cards (mandatory command), is optional or is reserved to the card vendor, as well as the acceptable arguments and which host controller register the card uses to store the response to the command. The application specific commands work as an extension to the SD command set that focus more in application related function of the card, such as access to the card's storage.

Sending commands to a SD card must, however, take into account the current card state and the command transition table defined by the protocol, which specifies when each command may be sent to the card and which state the card will take after the operation, such as idle or ready for instance.

6.2.2 SD Card Registers

Each SD card contains a register set which carries the card specific information:

- Operating Condition Register (OCR) - stores the card voltage support;
- Card IDentification (CID) register - contains card identification data, for instance the manufacturer, product name, serial number and manufacturing date;
- Card Specific Data (CSD) register - provides information about how to access the card contents. This register has different versions for SDSC and for higher storage capacity card such as SDHC and SDXC cards, so a device driver should take the register version into account when reading the register as the structure is different for each version;
- Relative Card Address (RCA) register - stores the relative address of the card which is assigned by every host controller. This allows commands to be sent to a specific card;
- SD card Configuration Register (SCR) - this is a vendor specific register which is populated in the factory with vendor specific configuration options for special features that are not mandatory by the SD standard.

Each register is accessed by a corresponding SD command.

6.3 The SD Host Controller Protocol

As referred in section 6.1.1 the host controller hardware provides the host system with a register set that can be used to interface with an inserted card. Some of these registers include:

- SD command generation - Generates commands to be sent to a card, including their arguments;

- Response registers - Hold a card's response to a sent command;
- Host control registers - Can be used to operate the host controller hardware, such as getting the present status, controlling the SD bus or resetting the hardware;
- Interrupt control - To manage host controller interrupts, such as when a card is inserted;
- Capabilities - Informative register with vendor specific support information about a specific host controller hardware;
- Advanced DMA registers - Allows DMA access to the card, without interrupting the CPU execution (refer to section 4.9.1);
- Transfer Mode - Defines the data blocks transfer mode, such as single or multi-block transfer, the transfer direction (to or from the SD card) and if the transfer uses DMA or not;
- Preset value registers - Contains the default values used by the host controller for each card.

To access these registers, and therefore to use the host controller and access a connected SD card, a device driver must take a specific sequence of actions which together compose the *SD Protocol*. The following subsections will introduce the protocol sequences that are implemented on the proposed device driver.

6.3.1 SD Card Detection

The flow chart shown in figure 6.1 represents the full sequence of steps required to detect the presence of an inserted card.

(1) Enable a interrupt to be generated by the host controller when card movement is detected (insertion or removal);

(2) Clear the interrupt on the host controller register, before using or finishing to use the card;

(3) Confirm that the card is inserted (or not) on the host controller registers before any further action.

The host controller records the card presence on its memory registers, and the only purpose of the generated interrupt is to notify the host system that a card has been inserted and can be used. Because the Raspberry Pi must have at all times a SD card inserted (refer to section 5.3), the proposed device driver only performs step (3) as reassurance that the host controller recognizes the card.

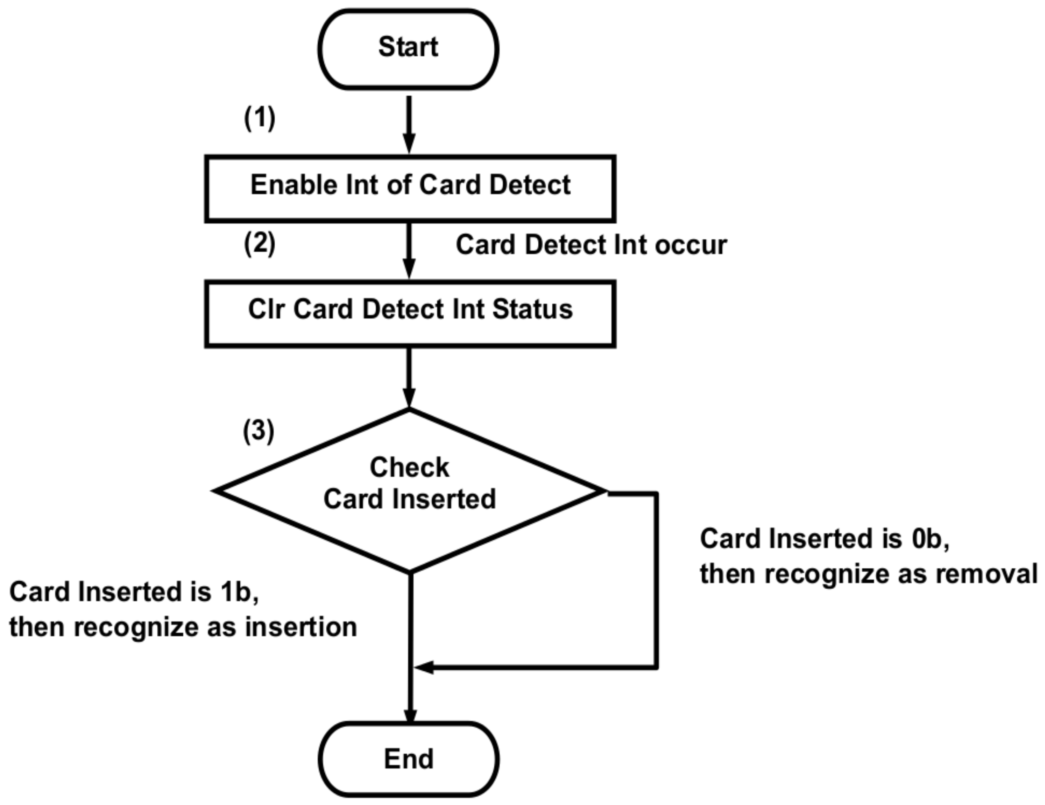

Start
(1)
Enable Int of Card Detect
(2)
Card Detect Int occur
Clr Card Detect Int Status
(3)
Check Card Inserted
Card Inserted is 0b, then recognize as removal
Card Inserted is 1b, then recognize as insertion
End

Figure 6.1: SD Card Detection Sequence [6].

6.3.2 SD Clock Control

For the system to be able to send commands to any SD card the host system must supply a clock, which will be used to synchronize the data transfers. The sequence to initialize the card's clock is shown in figure 6.2.

(1) Calculate a clock divider, which will be used to determine the SD clock frequency. This divider is used to divide the host system clock frequency so the supplied SD clock frequency matches the base clock frequency of the host controller;

(2) Set the host controller clock;

(3) Wait until the clock is stable;

(4) Enable the SD clock, so the host controller starts to supply the clock to the card.

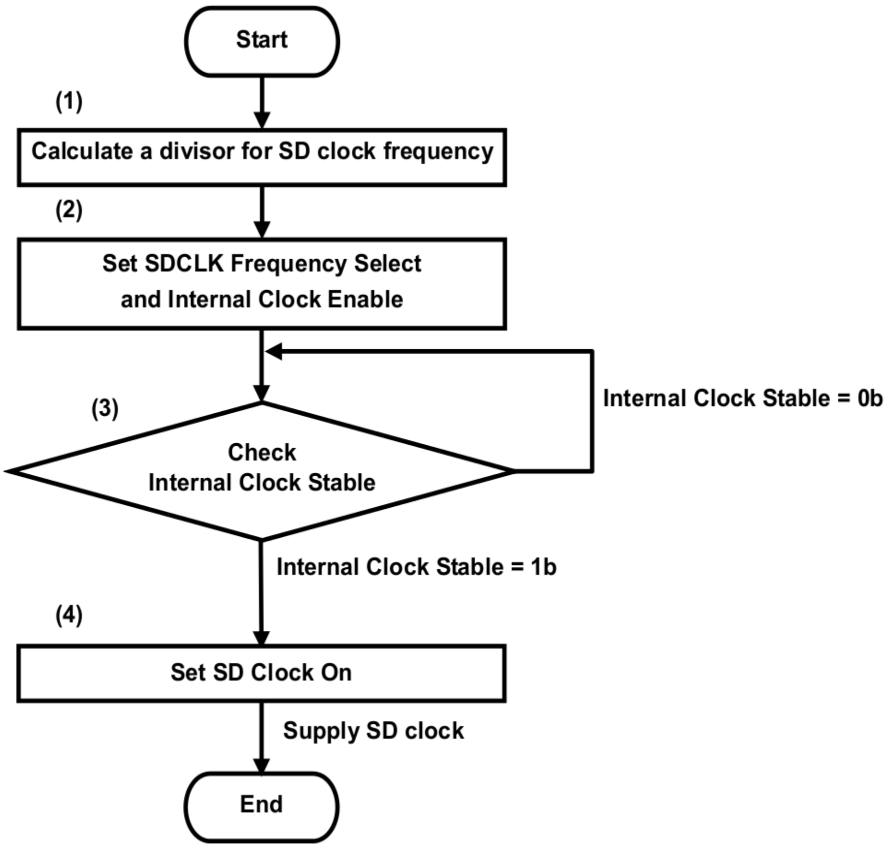

Start
(1)
Calculate a divisor for SD clock frequency
(2)
Set SDCLK Frequency Select and Internal Clock Enable
(3)
Check Internal Clock Stable
Internal Clock Stable = 0b
Internal Clock Stable = 1b
(4)
Set SD Clock On
Supply SD clock
End

Figure 6.2: SD Clock Supply Sequence [6].

6.3.3 SD Card Initialization and Identification

When a card is known to be inserted in the host controller and a clock has been supplied to it, the device driver may then proceed to initialize and identify the card to the host system. This process is described in figures 6.3 and 6.4.

(1) Reset all cards to idle state (send command 0);

(2) Check and set card voltage support (send command 8). The card only responds if the card is at least a SDHC memory card, so any SD memory card with less storage capacity or any non SD card will not respond to this command;

(3) Check the response to the previous command, and store it on a flag;

(4) If the response is an error abort the sequence;

(5) Send command 5, a SDIO specific command, to the card;

(6) If the card responds to the previous command the inserted card is a SDIO card. As the proposed driver does not support SDIO cards, the driver aborts the sequence here;

(11) Check the command 8 flag set in step (3). If the flag is set (the card responded to command 8) then the card is an SD memory card with lesser storage capacity than an SDHC card or it is not an SD card (may be a MMC card, for instance), and because these cards were not the objective of this project, the proposed device driver aborts the sequence at this point;

(12) Get the OCR register from the card (send application specific command 41) to check the card voltage support. If the card does not respond it is not a SD card;

(13) Check the received OCR information (refer to section 6.2.2);

(14) Initiate the memory part of the SD card by reissuing the application specific command 41 with the operating voltage setting defined by the host system (which should be within the supported voltage window defined on the card's OCR register);

(15) If the card does not respond to the previous command, it is not a SD card and the proposed driver aborts the sequence;

(16) Wait until the card processes the sent command;

(17) The host finally recognizes the card as a SDHC or SDXC card;

(32) Get the card's CID register (send command 2);

(33) Get the card's RCA register (send command 3) to retrieve the card's relative address (to the host controller).

The skipped sequence steps are specific to SDIO or non SD cards.

6.3.4 SD Command Issue

The sequence shown in figure 6.5 details the process of issuing a command to a SD card.

(1) Check the status of the command line (the line used to send commands) for the command inhibit signal;

(2)(3)(4) Check which type of command is about to be sent. If the command may start a data transfer, the driver must check the status of the data line for any command inhibit signal (step 4);

(5) Write the command arguments (if any) on the corresponding argument register;

(6) Set the command register on the host controller to send the desired command to the card;

(7) The command has been sent to the card. If the command was a request then wait for the response (refer to the other sequence).

6.3.5 Finalize SD Command

When a command is sent to a SD card with the intent of starting a data transfer, or if the command acts as a request, the driver must stay alert for the card's response. The required sequence of actions is shown in figure 6.6.

(1) Wait for the command complete interrupt;

(2) Clear the command complete interrupt;

(3) Read the command response register on the host controller;

(4) If the command starts a data transfer and generates a transfer complete interrupt when done, then wait for the interrupt, else move to step (7);

(5) Wait for the transfer complete interrupt;

(6) Clear the transfer complete interrupt;

(7) Check the response for errors;

(8)(9) Return the error status to the system application that originated the request.

6.3.6 Transferring Data

The sequence in figure 6.7 describes the complete process of reading and writing data blocks on a SD card.

(1) Set the size of the blocks to be transferred;

(2) Set the number of blocks that are going to be transferred;

(3) Write the SD data command argument in the host controller argument register;

(4) Set the transfer mode. The developed driver offers single and multi-block data transfers;

(5) Set the command register on the host controller to send the desired command to the card;

(6) Wait for the data transfer command completion interrupt;

(7) Clear the interrupt;

(8) Check the host controller response register for any transfer error;

(10) If the device driver is writing to the card, wait until the system application that wants to write to the SD card fills the write buffer, which has the same size of a data block. When the buffer is full an interrupt is generated;

(11) Clear the buffer ready interrupt;

(12) Write the buffer contents to the block data register in the host controller;

(13) Send the block to the card, and repeat from step (10) until all blocks have been sent;

(14) If a system application is reading from the SD card, wait until the card fills a read buffer, which has the same size of a data block. When the buffer is full an interrupt is generated;

(15) Clear the buffer ready interrupt;

(16) Read the data block from the host controller data register;

(17) Continue to read blocks from the card until the requested number of blocks have been read;

(19) Wait for the transfer complete interrupt, which confirms that the data transfer is over;

(20) Clear the interrupt.

6.4 Conclusions

This chapter introduced SD cards, the SD standard, as well as the SD bus and SD host controller protocols. These concepts will be applied in the next chapter, which will describe the implementation of a SDHC card device driver.

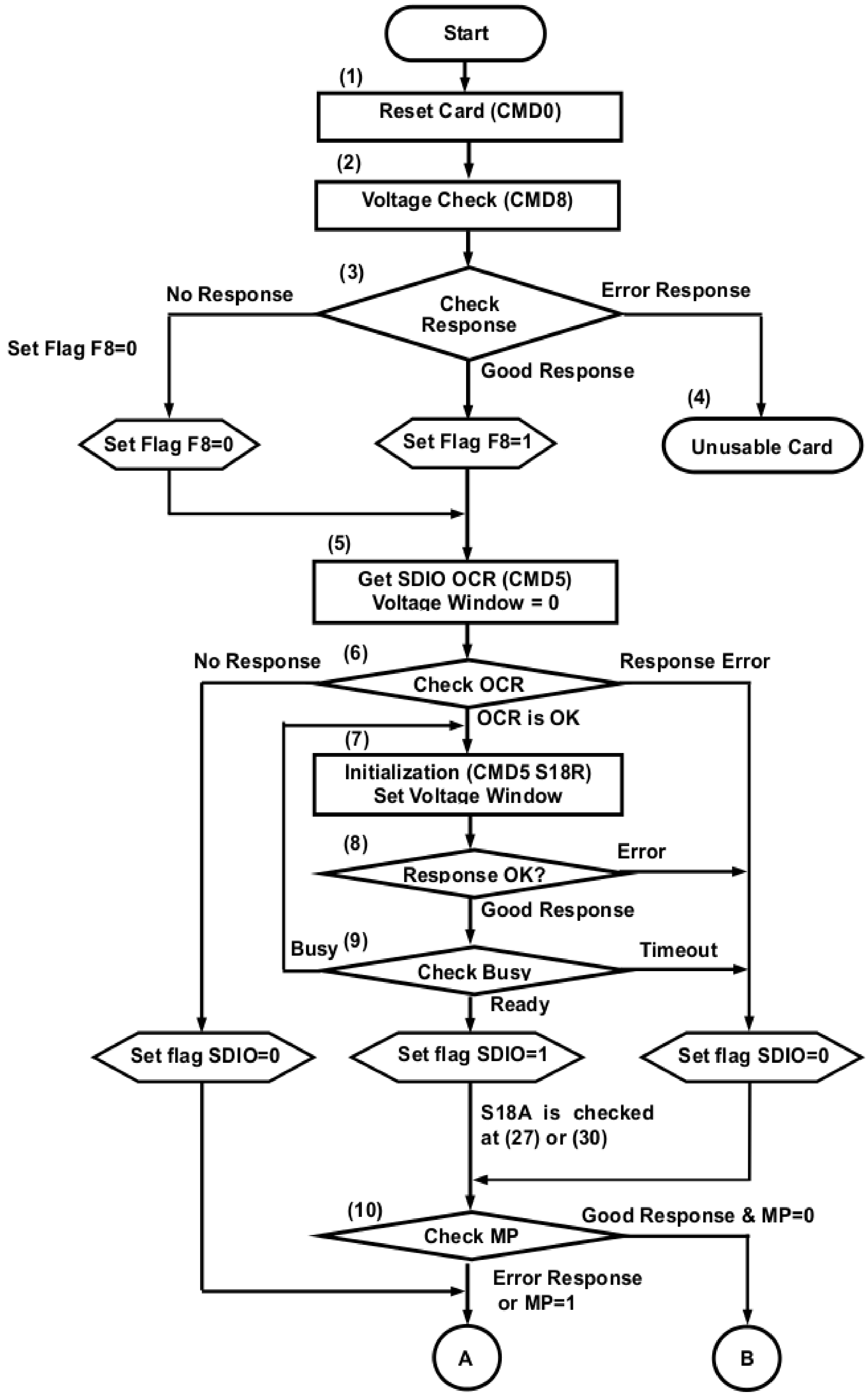


Figure 6.3: SD card initialization and identification sequence (part 1) [6].

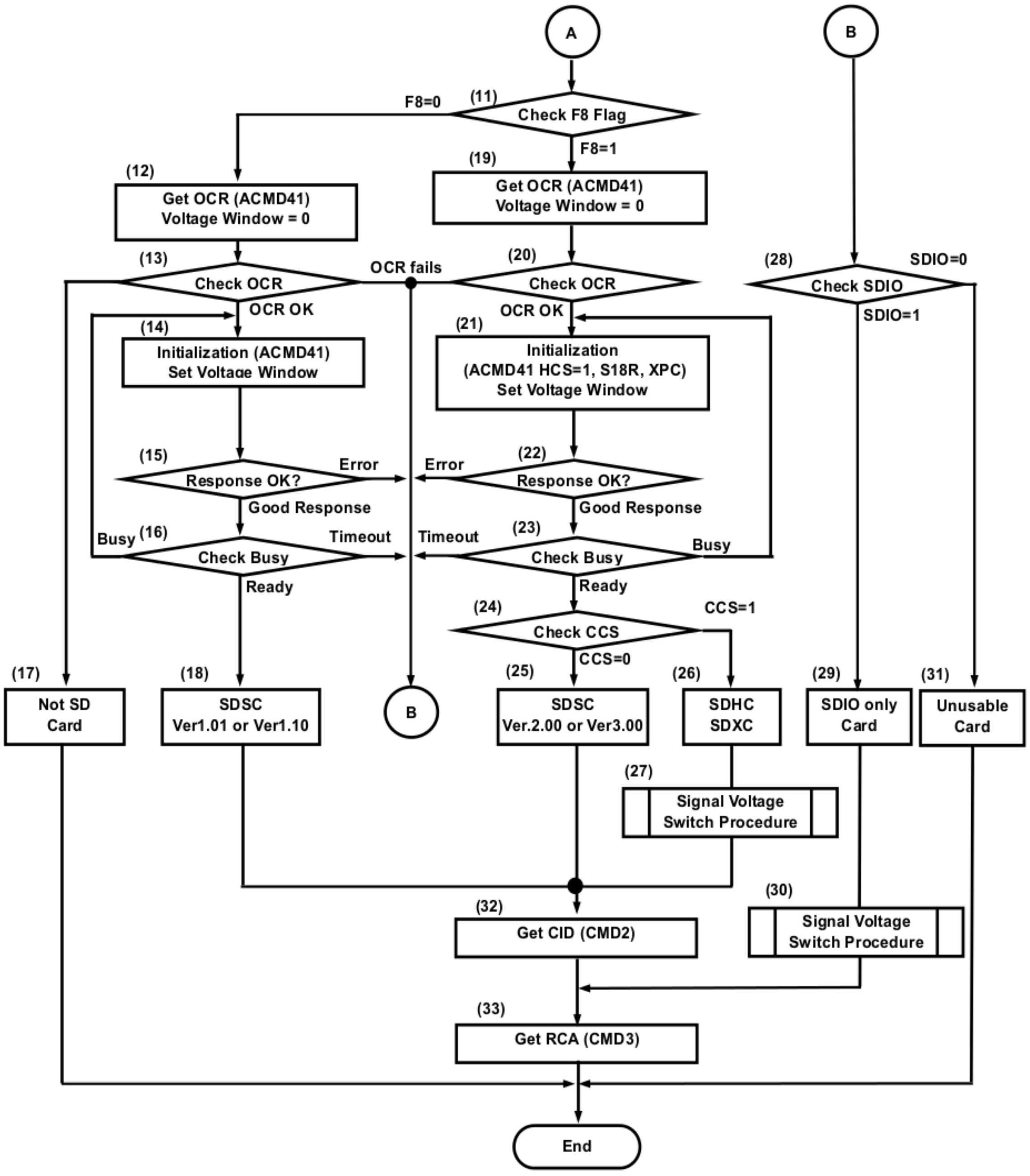


Figure 6.4: SD card initialization and identification sequence (part 2) [6].

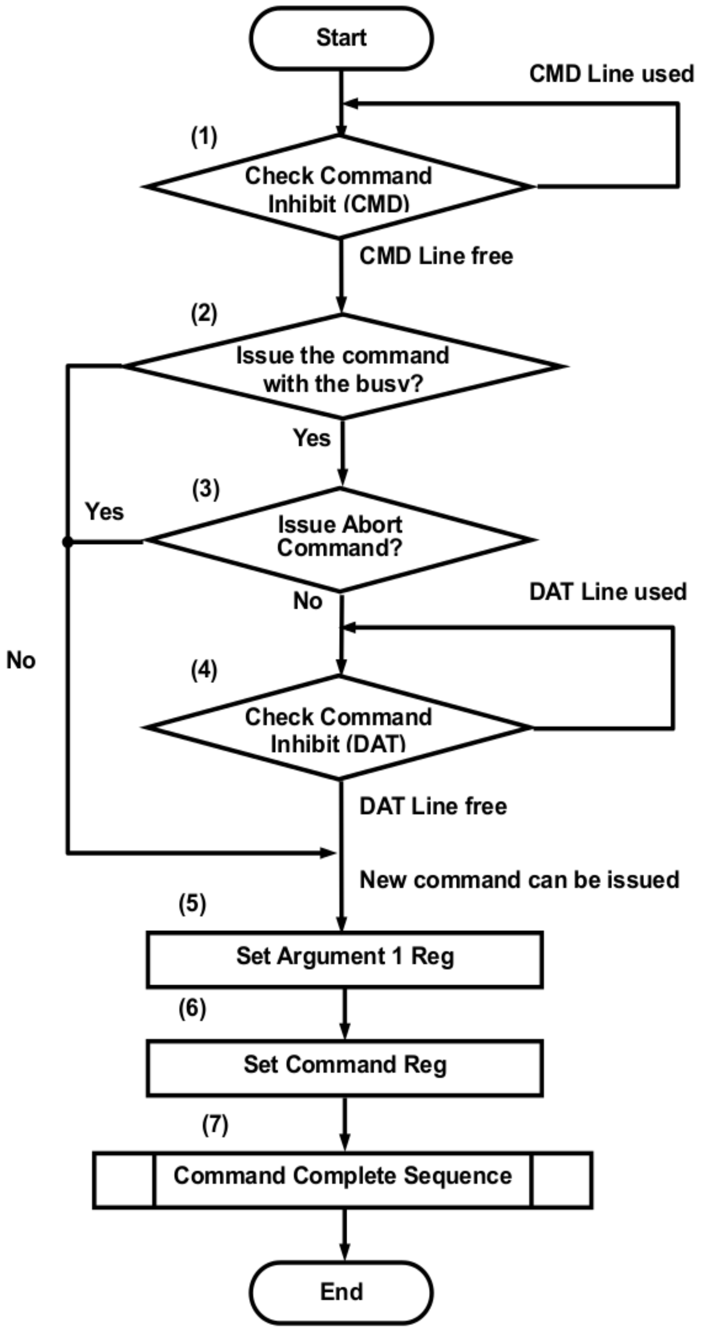


Figure 6.5: SD command issue sequence [6].

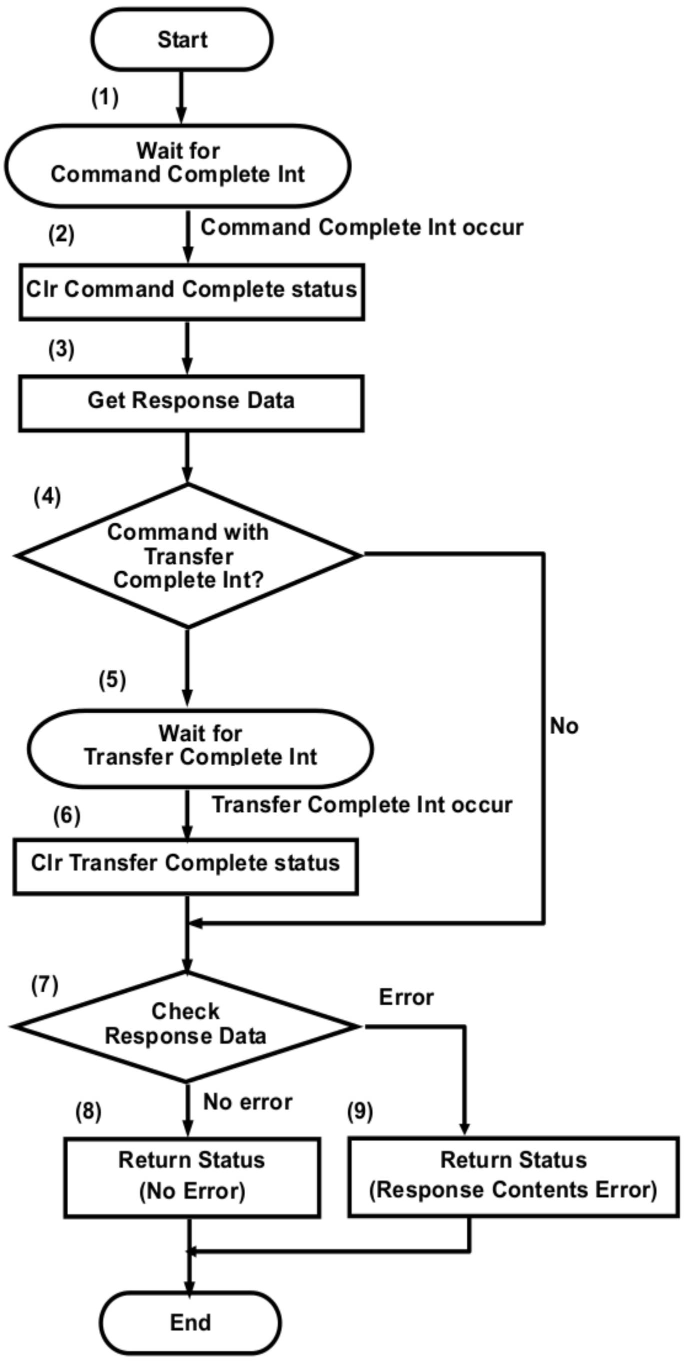

Start
(1)
Wait for Command Complete Int
(2)
Command Complete Int occur
Clr Command Complete status
(3)
Get Response Data
(4)
Command with Transfer Complete Int?
No
(5)
Wait for Transfer Complete Int
(6)
Transfer Complete Int occur
Clr Transfer Complete status
(7)
Check Response Data
Error
No error
(8)
(9)
Return Status (No Error)
Return Status (Response Contents Error)
End

Figure 6.6: Finalize SD command sequence [6].

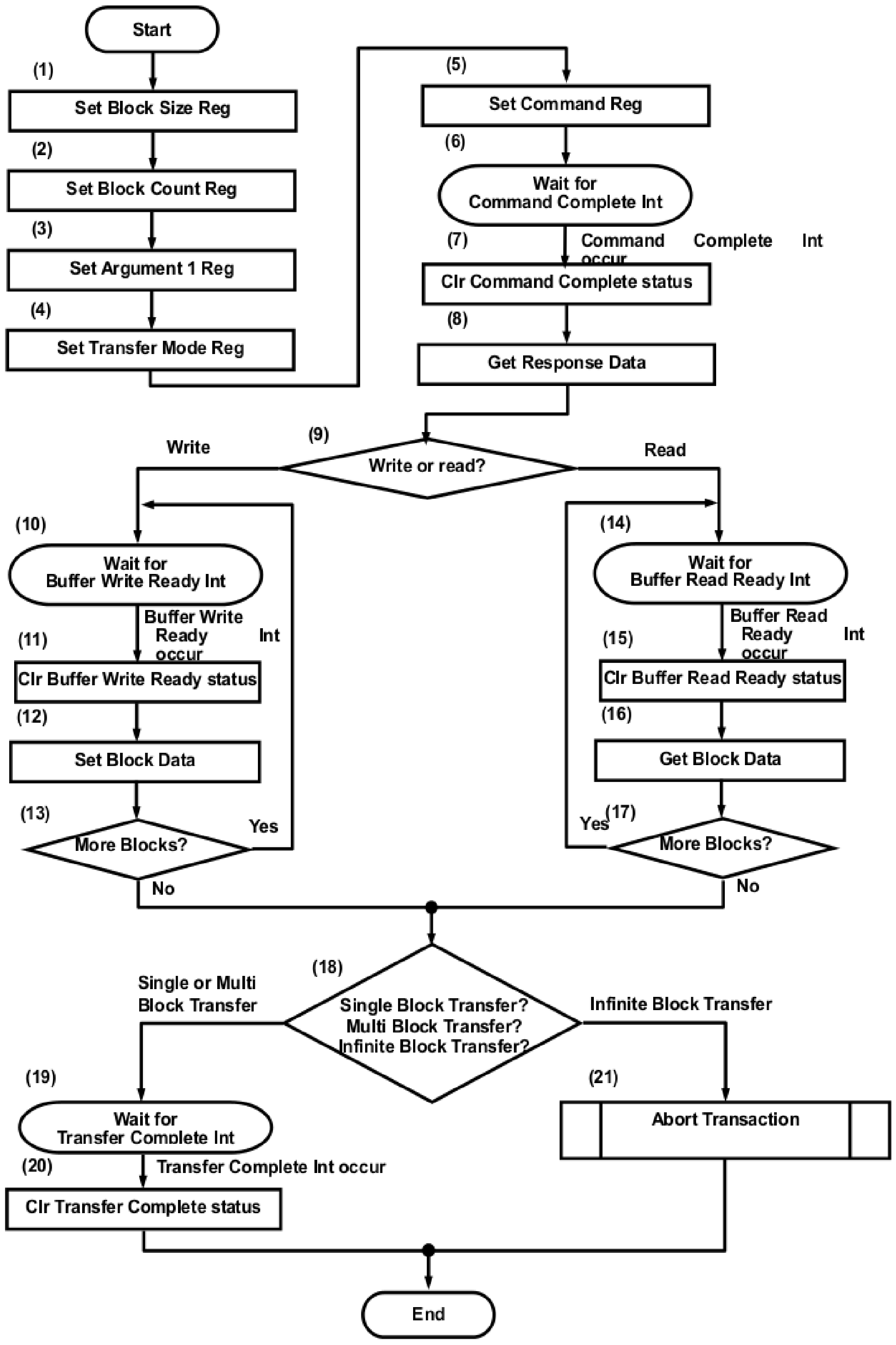

Start
(1)
Set Block Size Reg
(2)
Set Block Count Reg
(3)
Set Argument 1 Reg
(4)
Set Transfer Mode Reg
(5)
Set Command Reg
(6)
Wait for Command Complete Int
(7)
Command Complete Int occur
Clr Command Complete status
(8)
Get Response Data
(9)
Write or read?
Write
Read
(10)
Wait for Buffer Write Ready Int
Buffer Write Ready Int occur
(11)
Clr Buffer Write Ready status
(12)
Set Block Data
(13)
More Blocks?
Yes
No
(14)
Wait for Buffer Read Ready Int
Buffer Read Ready Int occur
(15)
Clr Buffer Read Ready status
(16)
Get Block Data
(17)
More Blocks?
Yes
No
(18)
Single Block Transfer? Multi Block Transfer? Infinite Block Transfer?
Single or Multi Block Transfer
Infinite Block Transfer
(19)
Wait for Transfer Complete Int
(20)
Transfer Complete Int occur
Clr Transfer Complete status
(21)
Abort Transaction
End

Figure 6.7: Finalize SD command sequence [6].

Chapter 7

Secure Digital Device Driver

This chapter describes the last deliverable of this project, which is the creation of a device driver for the SD card interface on the Raspberry Pi running the RTEMS operating system. The following sections describe how the SD protocol was implemented, how the device driver works with RTEMS and also provides information on how to use the device driver.

7.1 Implementing the Secure Digital Protocol

To create a device driver for a SD card, namely a SDHC card which was the type of card used during this project, it must implement the SD protocol. This protocol is described in the SD standard (refer to chapter 6), and is used by the device driver to interact with the SD card. The protocol implementation also used some other code as reference [9].

The following subsections will briefly describe the most relevant functions used in the driver implementation.

7.1.1 card_detection

```
static
rtems_status_code card_detection(struct emmc_block_dev *e)
```

Listing 7.1: Card detection function prototype.

Listing 7.1 shows the prototype of the function which checks the host controller status bit for the presence of a SD card. This function implements the sequence shown in section 6.3.1

7.1.2 sd_get_base_clock_hz

```
static
uint32_t sd_get_base_clock_hz( struct emmc_block_dev *e )
```

Listing 7.2: SD get base clock function prototype.

Listing 7.2 shows the prototype of the function used to to get the SD card base clock rate through the Raspberry Pi mailbox interface (refer to section 5.1.1).

7.1.3 sd_get_clock_divider

```
static
rtems_status_code sd_get_clock_divider ( struct emmc_block_dev *
    e, uint32_t* divider , uint32_t base_clock , uint32_t
    target_rate )
```

Listing 7.3: Get clock divider function prototype.

Listing 7.3 shows the prototype of the function which calculates a clock divisor which is needed to supply a clock to the card (refer to the protocol sequence in section 6.3.2. This function is mainly based in code from [9].

7.1.4 sd_handle_interrupts

```
static
void sd_handle_interrupts ( struct emmc_block_dev *e )
```

Listing 7.4: Interrupt handling function prototype.

Listing 7.4 shows the prototype of the function which handles the card's interrupts. It reads the host controller interrupt register (refer to section 6.3) and clears any pending interrupt.

7.1.5 sd_issue_command

```
static
rtems_status_code sd_issue_command( struct emmc_block_dev *e,
    enum commands command, uint32_t argument , useconds_t timeout
    )
```

Listing 7.5: SD issue command function prototype.

Listing 7.5 shows the prototype of the function that starts the process of sending a SD command to the card. It starts by handling any pending card interrupts (refer to section 7.1.4), and stops if the card removal interrupt was handled, meaning that the card was removed. If the card was not removed, proceed with the command issue (refer to section 7.1.5).

7.1.6 sd_issue_command_int

```
static
rtems_status_code sd_issue_command_int ( struct emmc_block_dev *
    e, enum commands cmd_reg, uint32_t argument, useconds_t
    timeout )
```

Listing 7.6: SD issue command intermediary function prototype.

Listing 7.6 shows the prototype of the function that actually sends SD commands to the card. It does so by implementing the protocol sequences shown in section 6.3.4 and 6.3.5.

7.1.7 sd_card_init

```
static
rtems_status_code sd_card_init (struct emmc_block_dev *e )
```

Listing 7.7: SD card initialization function prototype.

Listing 7.6 shows the prototype of the function responsible by the host controller and SD card initialization. It starts by getting the host controller SD card slot version, which specifies the version of the SD host controller specification it complies to. This should return a version of at least 2.00, since the driver does not take into account any older specification.

This is followed by a check for a valid card by performing the card detection sequence (refer to section 6.3.1), resets the host controller circuit and supplies the card with the SD clock. For that it gets the SD base clock rate, calculates a clock divider using that base clock rate (refer to section 6.3.2) and finally supplies the SD clock to the card.

At this point it follows the protocol sequence shown in section 6.3.3, and defines on the driver device structure the card's block size and block count, which will be needed to mount the card's file system.

7.1.8 disk_ioctl

```
static
int disk_ioctl(rtems_disk_device *dd, uint32_t req, void *arg)
```

Listing 7.8: Card I/O function prototype.

Listing 7.8 shows the prototype of the I/O function that is assigned to the card's device file on the system. It implements a disk IOCTL function (refer to section 4.5) to attend the system's data requests to the SD card. It receives the device file, the request code and a buffer that may be used to store the read data from the card, or may contain data to be sent to the card. Depending on the request, it calls the write or read functions (refer to section 7.1.11).

7.1.9 sd_ensure_data_mode

```
static
int sd_ensure_data_mode ( struct emmc_block_dev *e )
```

Listing 7.9: SD card data mode check function prototype.

Listing 7.9 shows the prototype of the function that starts the protocol sequence shown in section 6.3.6. It checks the card status and setups the card for data transfer mode. If successful calls the function described in section 7.1.10.

7.1.10 sd_do_data_command

```
static
int sd_do_data_command ( struct emmc_block_dev *e, int is_write ,
     uint8_t *buf , size_t buf_size , uint32_t block_no )
```

Listing 7.10: Data transfer setup function prototype.

Listing 7.10 shows the prototype of the function that continues the work started in 7.1.9. It calculates how many blocks were requested, the transfer direction (read or write) and which transfer mode (single or multi-block) to use before sending the command (refer to section 7.1.5).

7.1.11 SD I/O directives

```
int sd_read ( struct emmc_block_dev *e, uint8_t *buf , size_t
    buf_size , uint32_t block_no )

int sd_write ( struct emmc_block_dev *e, uint8_t *buf , size_t
    buf_size , uint32_t block_no )
```

Listing 7.11: SD card I/O function prototypes.

Listing 7.11 shows the prototypes of the functions which are called by the card's IOCTL directive (refer to section 7.1.8) and starts a read or write operation. Both reading and write functions start by checking the card status (refer to section 7.1.9). If the function reports that the card is ready send a data command (refer to section 7.1.10).

7.2 The Device Driver Table

RTEMS provides its device drivers through a device driver table that can be used by an application to select only the needed drivers for a particular device setup. This table is provided in the RTEMS confdefs.h header file which consists of a series of conditional macros that adds or removes RTEMS features available to the application, such as network or clock drivers or support for a specific file system. This permits tailoring RTEMS for a specific application, keeping the system size as small as possible (a desirable feature to have in an embedded system). As described in section 4.5, a device driver table entry provides the set of directives that the device driver provides to the system.

The device driver created for this project only provides the initialize directive (see listing 7.12), as any operation with the card itself is done through an IOCTL function (refer to section 7.1.8) assigned to the card during initialization.

```
rtems_device_driver sd_card_disk_init (rtems_device_major_number
    major, rtems_device_minor_number minor, void *arg )
```

Listing 7.12: Device driver table initialization function prototype.

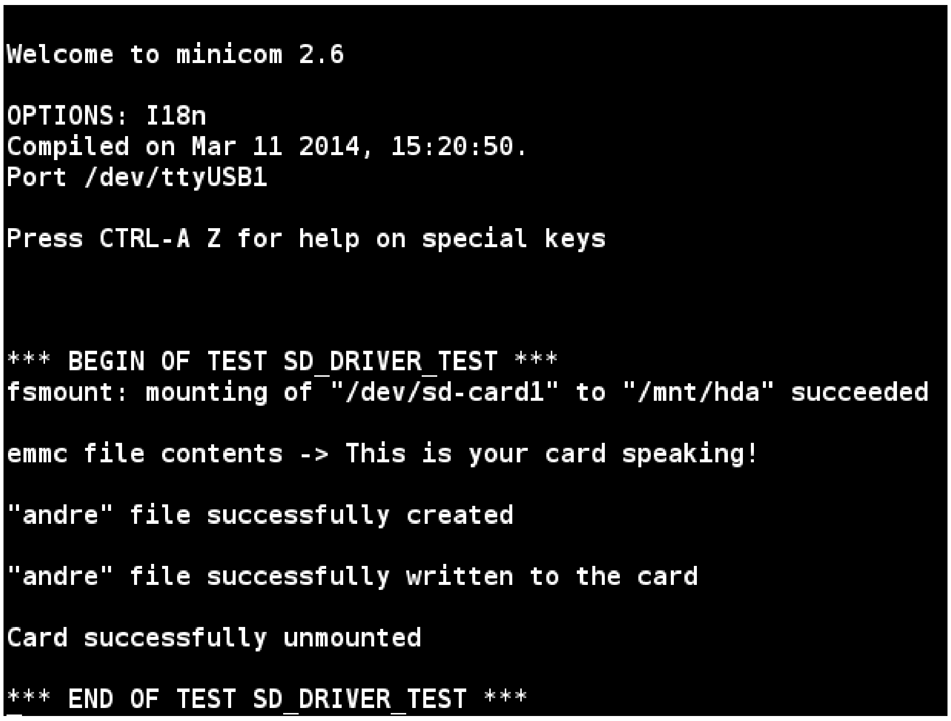
Listing 7.12 shows the prototype of the function that initializes the device driver, preparing the RTEMS system for the SD card and at the same time initializes the SD card to be used. It initializes the disk I/O management component of RTEMS, creates a device file for the card, initializes driver data structures and the card itself (refer to section 7.1.7). If all goes well it setups the card on the system and assigns it the disk IOCTL function which RTEMS applications will use to read and write to the SD card.

7.3 Testing the Device Driver

The test application reads and registers the card FAT partition on the system using the device file created by the device driver (hard-coded at /dev/sdcard, so the partition device file becomes /dev/sdcard1), and mounts the card in a mount point directory (/mnt/hda).

With the card mounted on the system, the test begins. It starts by reading a text file from the card called emmc which should already be in the card. It prints the file contents (proving that is can read from the SD card) and adds a new line of text to that file. Then creates a new file on the card named ANDRE with the same line of text that was added to the emmc file proving that the device driver can also write to the SD card.

In the end the application unmounts the SD card from the system, which will flush the RTEMS disk buffers to the card ensuring that no data is lost.



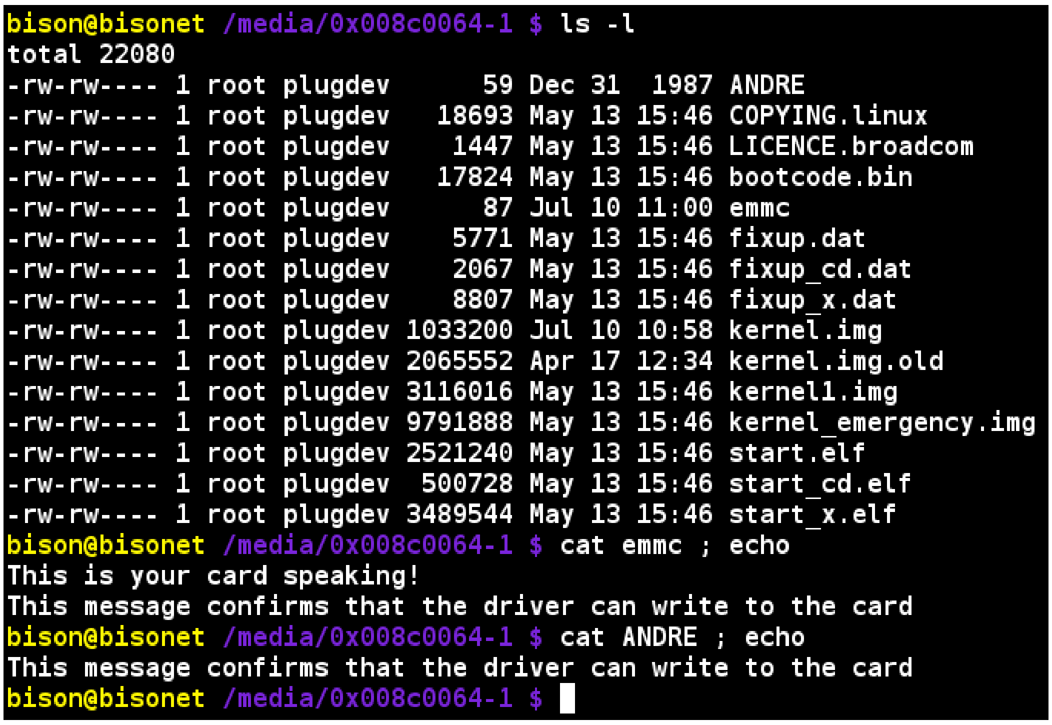
```
Welcome to minicom 2.6

OPTIONS: I18n
Compiled on Mar 11 2014, 15:20:50.
Port /dev/ttyUSB1

Press CTRL-A Z for help on special keys



*** BEGIN OF TEST SD_DRIVER_TEST ***
fsmount: mounting of "/dev/sd-card1" to "/mnt/hda" succeeded

emmc file contents -> This is your card speaking!

"andre" file successfully created

"andre" file successfully written to the card

Card successfully unmounted

*** END OF TEST SD_DRIVER_TEST ***
```

Figure 7.1: The test application running on the Raspberry Pi.

The results of the test can be seen in figures 7.1 and 7.2, and the test itself can be found either in bitbucket [16] or in the CD-ROM that was delivered with this report.

7.4 Conclusions

This chapter has described the relevant aspects of the device driver implemented for this project, as well as how it was tested. It works with SDHC cards as that was the only type of card accessible during the project development, but the base driver work is done and can be expanded to support more SD cards.



```
bison@bisonet /media/0x008c0064-1 $ ls -l
total 22080
-rw-rw---- 1 root plugdev      59 Dec 31  1987 ANDRE
-rw-rw---- 1 root plugdev   18693 May 13 15:46 COPYING.linux
-rw-rw---- 1 root plugdev    1447 May 13 15:46 LICENCE.broadcom
-rw-rw---- 1 root plugdev   17824 May 13 15:46 bootcode.bin
-rw-rw---- 1 root plugdev      87 Jul 10 11:00 emmc
-rw-rw---- 1 root plugdev    5771 May 13 15:46 fixup.dat
-rw-rw---- 1 root plugdev    2067 May 13 15:46 fixup_cd.dat
-rw-rw---- 1 root plugdev    8807 May 13 15:46 fixup_x.dat
-rw-rw---- 1 root plugdev 1033200 Jul 10 10:58 kernel.img
-rw-rw---- 1 root plugdev 2065552 Apr 17 12:34 kernel.img.old
-rw-rw---- 1 root plugdev 3116016 May 13 15:46 kernel1.img
-rw-rw---- 1 root plugdev 9791888 May 13 15:46 kernel_emergency.img
-rw-rw---- 1 root plugdev 2521240 May 13 15:46 start.elf
-rw-rw---- 1 root plugdev  500728 May 13 15:46 start_cd.elf
-rw-rw---- 1 root plugdev 3489544 May 13 15:46 start_x.elf
bison@bisonet /media/0x008c0064-1 $ cat emmc ; echo
This is your card speaking!
This message confirms that the driver can write to the card
bison@bisonet /media/0x008c0064-1 $ cat ANDRE ; echo
This message confirms that the driver can write to the card
bison@bisonet /media/0x008c0064-1 $ 
```

Figure 7.2: The SD card state after the test, read on another computer.

Chapter 8

Conclusions and Future Work

Developing this project provided me with experience in driver development and gave me the opportunity to contact with a RTOS system and to interact with its developer community. The greatest challenge I faced was the low level development needed and the necessity of interacting with Hardware with their own unique problems. Hardware proved to be tricky to program and the fact that RTEMS does not have a debugger for the Raspberry Pi did not help either.

The experience acquired during this project also helped me to craft a proposal and to be accepted in the Google Summer of Code 2014 [13], where I will continue the work started with this project by continuing to increase the RTEMS support for Raspberry Pi peripheral devices.

8.1 Future Work

The work done for this project can f course be improved upon and extended. The rename POSIX unit test, for instance, is considered at this point to be final and has already made its way to the official RTEMS code base, but it represents a test revealing that there is some work to be done in the RTEMS rename function implementation. This was not an objective for this project, but nevertheless by having a test in hand the implementation can now be corrected.

As for the SD device driver it can be optimized to provide better I/O performance by taking advantage of more voltage and operating modes that each SD card provides, and by using DMA access to the card's memory. More important than that, the device driver could also be more inclusive and stable (in RTEMS standards), as it just covers SDHC cards and even in that category only one card was used to test the driver. Because each type of card have its oddities, this means that the proposed driver can not be classified as stable as that would imply to test the driver with much more cards which was impractical to do during this project.

Apart from the tangible work done, the knowledge gathered during this project will definitely become an asset in the future, as I am now more confidant working in low level system programming, which is important when working around any operating system. Also the knowledge of the SD protocol can be useful outside Raspberry Pi and RTEMS since the SD standard is present in so many places in today's mobile world, thus representing an advantage for future projects that can be either based on this project or benefit from the knowledge it provided.

Appendix A

Rename POSIX test results

```
*** BEGIN OF TEST FSRENAME MOUNTED IMFS ***
Initializing filesystem MOUNTED IMFS

Old is a simbolic link and rename operates on the simbolic link
    itself

Testing rename      with arguments: symlink01, name02     EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 78
Testing lstat       with arguments: name02, &statbuf      EXPECT
    "0"
PASS
Testing if name02 is now a symlink
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 86
Testing unlink      with arguments: name01                EXPECT
    "0"
PASS
Testing unlink      with arguments: name02                EXPECT
    "0"
PASS
Testing unlink      with arguments: symlink01             EXPECT
    "-1"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 94

New is a simbolic link and rename operates on the simbolic link
    itself

Testing rename      with arguments: name02, symlink01     EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 116
Testing lstat       with arguments: symlink01, &statbuf   EXPECT
    "0"
```

```
PASS
Testing that symlink01 is not a symlink
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 124
Testing unlink     with arguments: name01             EXPECT
    "0"
PASS
Testing unlink     with arguments: name02             EXPECT
    "-1"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 131
Testing unlink     with arguments: symlink01          EXPECT
    "0"
PASS

Testing with symbolic link loop's

Testing rename     with arguments: "path01, name01"    EXPECT "
    ELOOP"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 149
Testing rename     with arguments: "path01, name01"    EXPECT "
    ELOOP"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 152
Testing unlink     with arguments: name01             EXPECT
    "-1"
PASS
Testing unlink     with arguments: symlink01          EXPECT
    "0"
PASS
Testing unlink     with arguments: symlink02          EXPECT
    "0"
PASS
Testing rename     with arguments: "name01, path01"    EXPECT "
    ELOOP"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 180
Testing rename     with arguments: "name01, path01"    EXPECT "
    ELOOP"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 183
Testing unlink     with arguments: name01             EXPECT
    "0"
PASS
Testing unlink     with arguments: symlink01          EXPECT
    "0"
PASS
Testing unlink     with arguments: symlink02          EXPECT
    "0"
PASS

Rename file with itself
```

```
Testing rename      with arguments: name01, name01        EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 244
Testing unlink      with arguments: name01                EXPECT
    "0"
PASS

Rename file with itself through a hard link in another directory

Testing rename      with arguments: name01, path01        EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 271
Testing unlink      with arguments: name01                EXPECT
    "0"
PASS
Testing unlink      with arguments: path01                EXPECT
    "0"
PASS
Testing rmdir       with arguments: dir01                 EXPECT
    "0"
PASS

Rename directory with file

Testing rename      with arguments: "dir01, name01"       EXPECT "
    ENOTDIR"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 343
Testing unlink      with arguments: name01                EXPECT
    "0"
PASS
Testing rmdir       with arguments: dir01                 EXPECT
    "0"
PASS

Rename file with directory

Testing rename      with arguments: "name01, dir01"       EXPECT "
    EISDIR"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 367
Testing unlink      with arguments: name01                EXPECT
    "0"
PASS
Testing rmdir       with arguments: dir01                 EXPECT
    "0"
PASS

Rename directory with ancestor directory
```

```
Testing rename     with arguments: "dir02 , path01"      EXPECT "
    EINVAL"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 390
Testing rmdir      with arguments: path01                EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir02                 EXPECT
    "0"
PASS

Rename directory with non empty directory

Testing rename     with arguments: dir01 , dir02         EXPECT
    "-1"
PASS
Testing errno for EEXIST or ENOTEMPTY
PASS
Testing unlink     with arguments: path01                EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir01                 EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir02                 EXPECT
    "0"
PASS

Rename empty directory with another empty directory

Testing rename     with arguments: dir01 , dir02         EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 448
Testing rmdir      with arguments: dir01                 EXPECT
    "-1"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 454
Testing rmdir      with arguments: dir02                 EXPECT
    "0"
PASS
Testing rename     with arguments: "dir02 , path01"      EXPECT "
    EMLINK"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 483
Testing rmdir      with arguments: path01                EXPECT
    "-1"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 497
Testing rmdir      with arguments: dir02                 EXPECT
    "0"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 498
Testing rmdir      with arguments: dir01                 EXPECT
    "0"
```

```
PASS

Rename files within directories protected with S_ISVTX

Testing rename     with arguments: path01, name02       EXPECT
    "-1"
PASS
Testing errno for EPERM or EACCES
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 535
Testing unlink     with arguments: path01               EXPECT
    "0"
PASS
Testing unlink     with arguments: name02               EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir01                EXPECT
    "0"
PASS
Testing rename     with arguments: name02, path01       EXPECT
    "-1"
PASS
Testing errno for EPERM or EACCES
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 577
Testing unlink     with arguments: path01               EXPECT
    "0"
PASS
Testing unlink     with arguments: name02               EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir01                EXPECT
    "0"
PASS

Rename file with non existant file

Testing rename     with arguments: name01, name02       EXPECT
    "0"
PASS
Testing unlink     with arguments: name01               EXPECT
    "-1"
PASS
Testing unlink     with arguments: name02               EXPECT
    "0"
PASS
Testing rename     with arguments: "name02, name01"     EXPECT "
    ENOENT"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 660
Testing unlink     with arguments: name01               EXPECT
    "0"
```

```
PASS
Testing unlink      with arguments: name02              EXPECT
    "-1"
PASS

Rename file with non existant filepath

Testing rename      with arguments: "path01, name01"    EXPECT "
    ENOENT"
PASS
Testing unlink      with arguments: name01              EXPECT
    "-1"
PASS
Testing rmdir       with arguments: dir01               EXPECT
    "0"
PASS

Rename directory with non existant directory

Testing rename      with arguments: dir01, dir02        EXPECT
    "0"
PASS
Testing rmdir       with arguments: dir01               EXPECT
    "-1"
PASS
Testing rmdir       with arguments: dir02               EXPECT
    "0"
PASS

Rename file with a name size exceeding NAME_MAX

Testing rename      with arguments: "name01, filename"  EXPECT "
    ENAMETOOLONG"
PASS
Testing unlink      with arguments: name01              EXPECT
    "0"
PASS
Testing unlink      with arguments: filename            EXPECT
    "-1"
PASS

Rename directory with current directory

Testing rename      with arguments: ".", dir01          EXPECT
    "-1"
PASS
Testing errno for EINVAL or EBUSY
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 787
```

```
Testing rename     with arguments: dir01 , "."           EXPECT
    "-1"
PASS
Testing errno for EINVAL or EBUSY
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 801

Rename directory with previous directory

Testing rename     with arguments: ".." , dir01          EXPECT
    "-1"
PASS
Testing errno for EINVAL or EBUSY
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 817
Testing rename     with arguments: dir01 , ".."          EXPECT
    "-1"
PASS
Testing errno for EINVAL or EBUSY
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 831
Testing rmdir      with arguments: dir01                 EXPECT
    "0"
PASS

Testing empty filepaths

Testing rename     with arguments: "name01, ""       EXPECT "
    ENOENT"
PASS
Testing            with arguments: name01                EXPECT
    "0"
PASS
Testing rename     with arguments: "", name01"       EXPECT "
    ENOENT"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 869
Testing            with arguments: name01                EXPECT
    "0"
PASS

Rename two files on a directory with no write permission

Testing rename     with arguments: "name01 , name02"    EXPECT "
    EACCES"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 947

Rename file between two directories , with and without write
    access

Testing rename     with arguments: "name01, path01"     EXPECT "
    EACCES"
FAIL   testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 971
```

```
Testing rename     with arguments: "path01 , name01"     EXPECT "
   EACCES"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c: 978
Testing            with arguments: name01               EXPECT
    "0"
PASS
Testing            with arguments: path01               EXPECT
    "0"
PASS
Testing            with arguments: path01               EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir01                EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir02                EXPECT
    "0"
PASS

Rename two files on a directory with no execute permission

Testing rename     with arguments: "path01  , path02"    EXPECT "
   EACCES"
PASS

Rename file between two directories , with and without execute
    access

Testing rename     with arguments: "path01 , path02"     EXPECT "
   EACCES"
PASS
Testing rename     with arguments: "path02 , path01"     EXPECT "
   EACCES"
FAIL    testsuites/fstests/mimfs_fsrename/../fsrename/test.c:
    1103
Testing            with arguments: path01               EXPECT
    "0"
PASS
Testing            with arguments: path01               EXPECT
    "0"
PASS
Testing            with arguments: path02               EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir01                EXPECT
    "0"
PASS
Testing rmdir      with arguments: dir02                EXPECT
    "0"
```

```
PASS

Rename files across diferent filesystems

Testing rename     with arguments: "name01, path01"     EXPECT "
   EXDEV"
PASS
Testing            with arguments: path01               EXPECT
    "-1"
PASS
Testing            with arguments: name01               EXPECT
    "0"
PASS


Shutting down filesystem MOUNTED IMFS
*** END OF TEST FSRENAME MOUNTED IMFS ***
```

Listing A.1: Rename unit test results.

Bibliography

[1] Raspberry pi wiki pages. http://elinux.org/R-Pi_Hub. Last access 10 of July 2014.

[2] European Space Agency. Esa - operating systems. http://www.esa.int/TEC/Software_engineering_and_standardisation/TECLUMKNUQE_2.html. Last access 10 of July 2014.

[3] SD Association. https://www.sdcard.org. Last access 10 of July 2014.

[4] Bunnie's Blog. On hacking microsd cards. http://www.bunniestudios.com/blog/?p=3554. Last access 10 of July 2014.

[5] Broadcom Corporation, Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB40WW. *BCM2835 ARM Peripherals*, 2012.

[6] Technical Committee. *SD Specifications Part A2 SD Host Controller Simplified Specification*. SD Association, version 3.00 edition, February 2011.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, February 2005.

[8] On-Line Applications Research Corporation. Rtems git commit. http://git.rtems.org/rtems/commit/?id=27d240e050697cf3d5b1632fafd2312c5bc26e52. Last access June 2014.

[9] John Cronin. Github repository. https://github.com/jncronin/rpi-boot. Last access 10 of July 2014.

[10] Edisoft. Rtems centre - rtems architecture. http://rtemscentre.edisoft.pt/index.php?module=ContentExpress&file=index&func=display&ceid=21&meid=37. Last access 10 of July 2014.

[11] Raspberry Pi Foundation. Github firmare repository. https://github.com/raspberrypi/firmware/wiki/Accessing-mailboxes. Last access 10 of July 2014.

[12] Raspberry Pi Foundation. What is a raspberry pi. http://www.raspberrypi.org/help/faqs/. Last access 10 of July 2014.

[13] Google. Google summer of code andre marques acepted proposal. https://www.google-melange.com/gsoc/project/details/google/gsoc2014/andremarques/5668600916475904. Last access 10 of July 2014.

[14] The IEEE and The Open Group. *The Open Group Base Specifications, Issue 7, IEEE Std 1003.1*. 2013.

[15] ARM Ltd. Arm information center. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/I1014942.html. Last access 10 of July 2014.

[16] Andre Marques. Bitbucket repository. https://bitbucket.org/asuol/rtems-graduation-project/src/cd2f71c4b4db273a1fe6396197c9d38bce0873d1?at=master. Last access 10 of July 2014.

[17] On-Line Applications Research Corporation. *Getting Started with RTEMS*, 4.10.99.0 edition, February 2013.

[18] On-Line Applications Research Corporation. *RTEMS C User's Guide*, 4.10.99.0 edition, February 2013.

[19] On-Line Applications Research Corporation. *RTEMS Development Environment Guide*, 4.10.99.0 edition, February 2013.

[20] On-Line Applications Research Corporation. *RTEMS POSIX 1003.1 Compliance Guide*, 4.10.99.0 edition, February 2013.

[21] SD Group (Panasonic, SanDisk, Toshiba) and SD Card Association. *SD Specifications Part 1 Physical Layer Simplified Specification*, version 3.01 edition, May 2010.

[22] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 8th edition, July 2009.

[23] David Welch. Github raspberrypi repository. https://github.com/dwelch67/raspberrypi/tree/master/bootloader05. Last access 10 of July 2014.