Interfacing an SD Card with Patmos

Max Harry Rishøj Pedersen

Technical University of Denmark

Kongens Lyngby, June 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Sciences
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In this project an SD card is interfaced to the Patmos processor running on an Altera DE2-115 FPGA board, using the slow but simple SPI mode that such cards provide. A file system module is also built, which can access files on a FAT32 partition. The two parts connect to form a complete system for working with files on Patmos. An emphasis is placed on modularity and ease-of-use, as the work is to eventually be integrated into the Patmos project. An optimal file reading speed of 250 kB/s and writing speed of 150 kB/s has been achieved.

Preface

This thesis was written at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, as part of acquiring my B.Sc. in Software Technology.

It deals with the implementation of both hardware and low-level software in the context of greater projects and has been a very inspiring and valuable learning experience, for which I am grateful.

I would like to thank my supervisor Martin Schoeberl for introducing me to this project as well as for his guidance. A special thanks goes to Luca Pezzarossa who was immensely helpful when the hardware parts were developed, which at the time was new territory for me.

Lyngby, 22-June-2017

Max Harry Rishøj Pedersen

Contents

1 Introduction

Patmos[1] is a 32-bit, RISC-style processor optimized for low WCET (Worst Case Execution Time). It is at the core of the T-Crest[2] platform, which is aimed at real-time embedded systems and is time-predictable by enabling static analysis of the WCET.

Patmos is described in an HDL (Hardware Description Language), such that it can be synthesized unto an FPGA (Field-Programmable Gate Array). At the time of writing Patmos has no local, persistent storage capabilities and this project solves that problem by interfacing the processor with an SD card connected to an Altera DE2-115 FPGA board. Communication with the card is done over the SPI protocol and is performed by a hardware controller and a companion driver, which together provide a simple interface for accessing the raw data on the card. On top of this a file system module is built, which enables reading from and writing to files on a FAT32-formatted disk, which is the default format of SD cards, and provides a familiar interface for accessing those files in the C programming language.

It is the goal that the work of this thesis will be integrated into the Patmos project, but until then the implementation can be found in a publicly available fork of the Patmos project[1] on Github: `https://github.com/MaxRishoj/patmos`.

This thesis is structured as follows: In the next section is given an outline of work that relates to this project. In Section 3 is given the specific scope of this project, followed by an analysis of the necessary components. In Section 4 the overall design of the implementation is outlined and in Section 5 are the details of how this design was then realized. In Section 6 the results of this project are presented, along with a discussion of how the solution performs and in Section 7 suggestions are given to how it could be extended. Finally in Section 8 the project is concluded.

[1] https://github.com/t-crest/patmos

2 Related Work

In this short overview of the work that this project is built upon or which solved related problems.

2.1 Patmos / T-Crest

Being an extension to the project, none of the work presented in this thesis would be very relevant without the Patmos[1] and T-Crest[2] projects. While these projects are aimed at time-predictability however, this project does not touch on that subject but instead just utilizes Patmos as the executing processor. None the less, it is the foundation upon this project was built and it is a very interesting ecosystem.

2.2 SPI

Limited experience with hardware development, coupled with the sometimes non-exhaustive explanations in the SD specifications[3] made initial host controller design a very challenging task. The open SPI protocol is outlined in many different forms by vendors that utilize it, but a particularly great resource was found in the short and sweet "*SPI Implementation on FPGA*"[4], in which the authors provide great timing diagrams for the protocol.

3 Analysis

The following section contains an analysis of the problem this project solves. Here the scope of the problem is outlined and the details of each part are explored. After reading this section the reader should have a clear understanding of the problem, which forms the basis for understanding the design and implementation decisions.

3.1 Notation in this thesis

To avoid any confusion for the reader, following is a brief explanation of the notation used in this thesis.

All indices mentioned have index origin zero. Numbers appear in both decimal, hexadecimal and binary formats and are written `17 = 0x11 = 0b00010001` respectively. This is the notation used in the C programming language. Occasionally hexadecimal and binary ranges are represented with the use of the wildcard token "`X`" to represent "any value". An example of this is the inclusive range [`0xa0`, `0xaf`] written as `0xaX`.

3.2 Problem

The basic problem of this project is to interface an SD card with Patmos. The end result should enable programs executed on Patmos to access any files present on a connected SD card.

To achieve this, multiple parts must come together. At the lowest level, it is necessary to physically communicate with the SD card. That entails constructing a hardware controller for an SD card, such that the corrects signals can be sent to the card. Said controller will need a driver, which facilitates proper communication with the SD card, allowing for data be read and written. Finally a file system module must be written, which can interpret and utilize any FAT32-formatted partitions on the card.

This projects limits itself to supporting the SPI[2] mode of the SD card, and while all parts are developed using the standard specifications, none of them are fully compliant. Why this scope is chosen will be made clear in the following analysis.

3.3 Equipment

Development is done using the virtual machine image provided at the Patmos website hosted by DTU[3]. It provides an Ubuntu[4] installation, with all the necessary tools for Patmos development already present. The project is developed and tested on the Altera DE2-115 FPGA board, which will often just be referred to as *the board*. Hardware components are connected to Patmos, which is then

[2] Serial Peripheral Interface

[3] Patmos website: `http://patmos.compute.dtu.dk/`

[4] Linux distribution. See `https://www.ubuntu.com`.

synthesized unto the board using the existing Makefile-based[5] build system of T-Crest and Patmos. Some parts of the hardware development use the Altera software "Quartus II 14.1", which come pre-installed on the Ubuntu image. For the SD card a "SanDisk Ultra 16GB MicroSDHC UHS-I Memory Card" is used, which is placed in a MicroSD to SD adapter, allowing it to fit in the SD card slot of the board.

3.4 SD Cards

SD (Secure Digital) is a memory card format developed by the SD Card Association (SDA). While the complete specifications for SD cards and related components require a license, the SDA has released simplified versions of them which are open to the public. It is the specifications for the physical layer[3] (the card) and host controllers[5] (card slot and controller) that the following analysis is based on.

The interface of an SD card is 9 pins present on the bottom of the card. Four of these are for data and communication, while the rest is for clock, power and ground. Figure 1 shows this setup. Some later models of SD cards, of the type UHS-II (Ultra High Speed), have an additional row of pins used for operating the card in high speed mode, but these are ignored as using that mode is optional and the standard pins are unaffected.

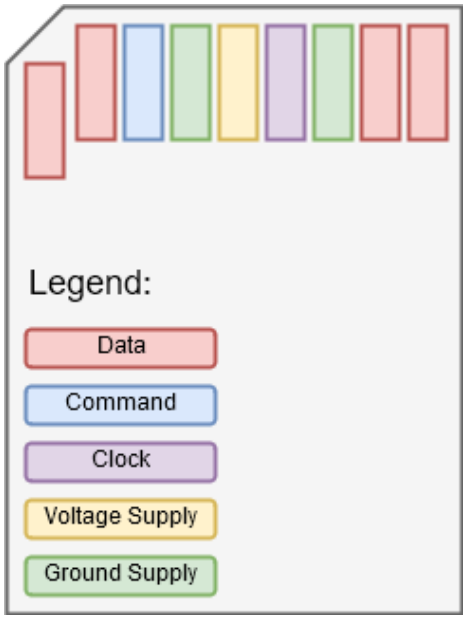

Figure 1: Pin layout of an SD card. Loosely based on Figure 3-11 in *"SD Specifications Part 1 Physical Layer Simplified Specification, Version 5.00"*[3]

[5]See `https://www.gnu.org/software/make/manual/make.html`

SD cards can store anywhere from a few megabytes to two terabytes of data and depending on the capacity, cards are separated into three capacity classes. In Table 1 is shown how the cards are classified. This is important to note as the different classes operate slightly differently in some respects.

Capacity	Class	Card Name
≤ 2 GB	Standard Capacity	SDSC
2 GB - 32 GB	High Capacity	SDHC
32 GB - 2 TB	Extended Capacity	SDXC

Table 1: Capacity classes of SD cards.

3.4.1 Modes

As already mentioned, this project utilizes the SPI mode of the SD card. The full range of modes, listed in order of decreasing data transfer speed, are: 4-bit SD, 1-bit SD and 1-bit SPI. The "1-bit / 4-bit" part refers to how many pins are used for data transfer and "SD / SPI" refers to the transfer protocol used. While using the 4-bit SD mode can achieve must faster speeds, the protocol is much more complex, which is why the SPI mode was chosen.

3.4.2 SPI

SPI (Serial Peripheral Interface) is a synchronous, full-duplex, serial communication protocol, meaning that transactions are synchronized to a clock signal and data is sent one bit a time, both ways simultaneously. Table 2 shows an overview of the signals[4].

Signal	Name
`SCK`	Serial Clock
`MOSI`	Master-Out-Slave-In
`MISO`	Master-In-Slave-Out
`CS`	Chip Select

Table 2: Signals of the SPI protocol.

SD cards in SPI mode are connected to the host controller in a master/slave fashion, where the card (slave) only reacts when commanded by the controller (master). Every clock cycle of `SCK` a bit is sent from master to slave over the `MOSI` signal, and from slave to master over the `MISO` signal. All data sent in SPI mode is a number of whole bytes and it must be byte-aligned to the `CS` signal[3, Section 7.2]. A slave will only react to the master when the `CS` signal is held low, which allows multiple slaves to operate independently while connected to a single master. While not required, it is standard to sample from `MOSI` and `MISO` on different edges of `SCK`[4].

3.4.3 Commands

A SD card is controlled with commands issued by the host controller. In SD mode commands are sent over the dedicated command line `CMD`, but in SPI mode they are sent over `MOSI`. SD commands are 6 bytes or 48 bits long and they always begin with bits `01` and end with `0`. The contents of a command is then 6-bit command index (similar to an opcode), a 32-bit parameter and a 7-bit CRC[6] (Cyclic Redundancy Code) used for detecting transaction errors. Figure 2 shows this structure.

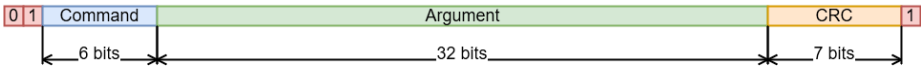

Figure 2: Structure of an SD command

The commands used in this project will be explained as they are encountered in Section 3.4.6 and Section 3.4.7, but for a complete list we refer to the specifications[3, Section 4.7.4]. A special type of command to note however, is application specific commands or ACMD. This type of command requires first sending a `CMD55` (`APP_CMD`) to indicate that the next command is an ACMD and not a standard command. An example is `ACMD41` (`SD_SEND_OP_COND`), which requires first sending `CMD55` and then `CMD41`.

3.4.4 Responses

After a command is sent the card holds `MISO` high until it returns with a response. The format of this response depends on which command it followed. The most common response in SPI mode is a R1 response, which is a single byte long and where the lower `7` bits indicate the status of the card. These bits must be inspected by the driver to figure out what was wrong with the command, if anything. The meaning of the individual bits are detailed in Table 3.

Bit	Meaning if set
0	Card is in idle state.
1	An erase sequence was reset.
2	The command received was illegal.
3	The CRC check failed for the command.
4	Error occurred in erase sequence.
5	The address in the command was misaligned to the blocks.
6	Invalid parameters were provided with the command.

Table 3: Meaning of the individual bits of a R1 response

Most of the other response types are only relevant when developing a fully compliant host controller and are ignored in this project. However, the type R7 is encountered when sending `CMD8` during initialization, so it is briefly outlined

here. Besides a large chunk of reserved bits, it contains a 3-bit voltage field which if is set to `1 = 0b0001` indicates that the card supports the standard voltage range 2.7V-3.6V, in which the boards default supply of 3.3V falls. The complete structure of the response can be seen in Figure 3.

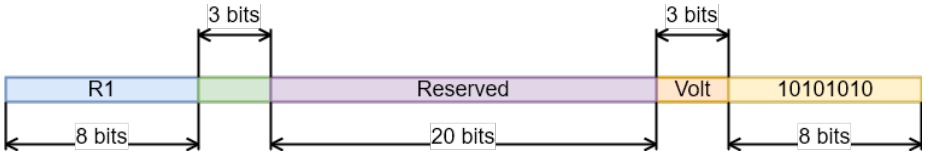

Figure 3: Structure of a R7 response. Loosely based on Figure 7-12 in *"SD Specifications Part 1 Physical Layer Simplified Specification, Version 5.00"*[3]

3.4.5 CRC

The last 7 bits an SD command is reserved for a CRC which the card can use to detect if any errors occurred in the transmission. However these codes are disabled per default in the SPI mode and are thus required for a few commands when initializing the card and in those cases they can be pre-computed as the data they cover is static. Two simple but inefficient implementations of CRC generation can be found in Appendix A and B, both of which were developed when attempting to understand the algorithm.

3.4.6 Card initialization

The card is powered up when voltage is supplied over the power line. At this point the card immediately enters an idle state and before it can be used for data transfer, it must be moved into a data transfer state. Figure 4 shows a basic flow diagram for initialization in SPI mode. The figure is based heavily on Figure 7-1 in the simplified specification for the physical layer[3, Figure 7-1]. Note that `CMD0` (`GO_IDLE`) must be issued while holding `CS` high, otherwise the card will enter SD mode. It is always possible to get back to the idle state by halting the power supply for at least 1 ms, which can be done manually by simply pulling the card from the slot and plugging it back in. This is referred to as *power cycling*.

The next command `CMD8` (`SEND_IF_COND`) sends the voltage range of the host to the card. For the normal range of 2.7V-3.6V the voltage index is 1, which results in the complete command shown in Figure 5. For other values, we refer to the specifications[3, Table 4-18]. If the card accepts the range, it responds with a R7 response that has identical voltage index. Note that because the argument to this command is static, the CRC can be pre-computed and after this command succeeds the card disables CRC checking.

The next command `ACMD41` (`SD_SEND_OP_COND`) negotiates the capacity of the card. Setting bit 30 in the argument indicates High Capacity Support (HCS)

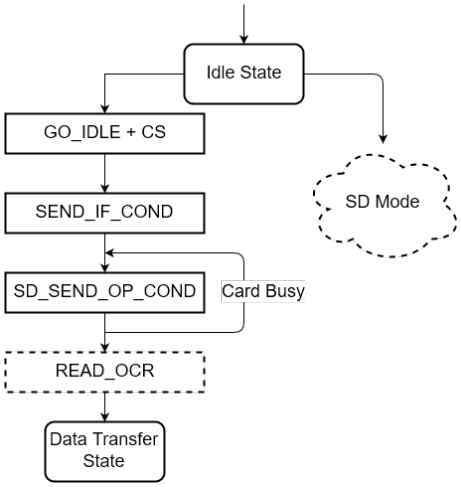

Figure 4: Basic state diagram for SPI mode initialization

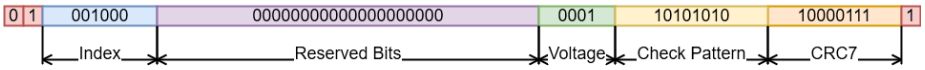

Figure 5: Typical contents of `CMD8`. Loosely based on Figure 7-1 in *"SD Specifications Part 1 Physical Layer Simplified Specification, Version 5.00"*[3]

which will not succeed for SDSC cards, but should be done for SDHC and SDXC. The card responds with an R1 response which indicates if the card has left the idle state. This will not happen if either the card is already in the process of initializing or if it does not support the HCS setting.

At this point the card is ready to transfer data. A compliant host controller would issue a `CMD58` (`READ_OCR`) to verify the precise voltage support of the card, but it is not strictly necessary. For more information see specifications [3, Section 4.7.4].

3.4.7 Reading and writing

The memory of an SD card can be thought of simply as a large array, which can only be accessed in *blocks* of bytes. The size of a block is configurable for SDSC cards with `CMD16` (`SET_BLOCKLEN`), but SDHC and SDXC are limited to the default (for all cards) size of 512 bytes. Reading a block is done with `CMD17` (`READ_SINGLE_BLOCK`), which given a block-address returns a block of data followed by a 16-bit CRC (see Section 3.4.5). The order of the transaction is Command → Response → Data. Given that only blocks are used, it means that to read byte $i_{byte} = 1025$ (index), it requires reading block $i_{block} = \lfloor i_{byte}/S_{block} \rfloor = 2$ where $S_{block} = 512$ is the block size, and then the

byte with index $i' = i_{byte} \mod S_{block} = 1$ in that block will be byte i_{byte}.

Writing a block to a given address is done with `CMD24` (`WRITE_SINGLE_BLOCK`), which after receiving the data responds with a "Data Response Token". Only the lowest 5 bits of the token are important and the values of those are `0b100101` (data accepted), `0b01011` (data rejected because CRC checking failed) and `0b01101` (data rejected due to write error). After this token the card holds the line high until the data has been written and the card is ready for the next command.

Blocks are always aligned to the beginning of memory. This has the implication, that to modify only some bytes within a block, it is necessary to first read the entire block, change the affected bytes and then write the block back. Otherwise the non-targeted bytes within the block would simply be lost. Figure 6 shows an example of this and what approach is necessary for which blocks.

Figure 6: Illustration of actions required when altering bytes in blocks

Timing Any data of a read or a write operation must be preceded by "Data Start Token", which is the value `DAT_START = 0xfe = 0xb11111110`. Figure 7 shows timing diagrams for the read and write operations. Note that only the order of the operations is valid in this figure, since some transmissions take longer than other.

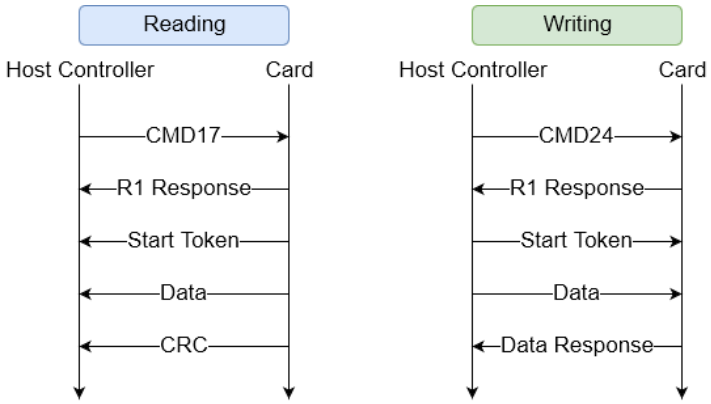

Figure 7: Timing diagram of SD read and write operations

3.5 I/O devices in Patmos

Components for Patmos are written in a modern HDL called Chisel, developed at UC Berkeley[7], which is based on the programming language Scala. I/O (Input / Output) devices connected to Patmos are memory-mapped[8], such that each has a dedicated range of memory addresses. Such a range contains $2^{14} = 16384$ 4-byte words and where it begins is determined by the configuration of the system. Any reads or writes to this memory segment will trigger a transaction between the device and Patmos.

The protocol for this transaction is an adaptation of OCP (Open Core Protocol), which operates as master/slave, where Patmos is the master and the device is the slave. Slightly different variations of this interface are available for the devices, but for the purpose of this project only the simplest variant "OCPcore" is needed. Other variants are necessary if for example the device has its own clock and clock-domain crossing is required. Table 4 shows an overview of the signals in "OCPcore". The table is based on Table 3.8 in the Patmos Reference Handbook [8, Table 3.8] as well as the source code of the interface.

Name	Bits	Description	Possible values
`MCmd`	3	Command from master.	`IDLE`, `WR`, `RD`
`MData`	32	Data from master.	Any
`MAddr`	32	Address from master.	`0x00000000` - `0xFFFFFFFC`
`MByteEn`	4	Byte enable signal.	Any
`SResp`	2	Response from slave.	`NULL`, `DVA`, `FAIL`, `ERR`
`SData`	32	Data from slave.	Any

Table 4: Signals of the "OCPcore" interface

Sending data to a device through this interface is then done by writing to the memory region associated with the device. As an example consider the UART (Universal Asynchronous Receiver/Transmitter), which is per default mapped to the region `0xf008XXXX`. Sending data to this device could be done with `(volatile int *)0xf0080004 = 42`, in which a transaction begins. Patmos begins with setting `MCmd` to `WR` (a write command), `MAddr` to `0xf0080004` and `MData` to `42 = 0x0000002a`. The UART device then inspects `MAddr` and `MData` to determine what it must do, before responding by setting `SResp` to `DVA` (data available), at which point the transaction is done and execution continues. Reading data is much the same, except that `MData` is disregarded and the slave must place the returning data on `SData`.

3.6 Master Boot Record

To locate file systems present on the card, it is necessary to consult the "Master Boot Record"[9]. This is located in the first logical sector (512 bytes) of the disk and in it is found the *partition table*. This table is always located in bytes 446 - 509 and consists of four 16-bit partition entries. Table 5 shows an overview of the values in such an entry. The only parts relevant to this project is the type of

the partition, the address of the first sector of the partition and the size of the volume. The type is used for ensuring it is a FAT32 partition and the address together with the size denotes where the FAT32 volume resides on the disk. The CHS (Cylinder-Head-Sector) address fields can be ignored, as they are irrelevant for an SD card that has no heads or cylinders. Only the LBA (Logical Block Addresssing) address of the first sector is relevant. The LBA address can be interpreted as the absolute sector index and the supported partition types for this project are `0x0c` and `0x0b`, both of which indicate FAT32.

Offset	Bytes	Description
`0`	1	Status indicating if partition is bootable. Ignored.
`1`	3	CHS address of first sector in partition. Ignored.
`4`	1	Partition type.
`5`	3	CHS address of last sector in partition. Ignored.
`8`	4	LBA address of first sector in partition.
`12`	4	Number of sectors in partition.

Table 5: Fields in a partition table entry of a MBR

3.7 FAT32

While there are many file systems available to choose from, SD cards come pre-formatted with FAT32 or potentially FAT16 for SDSC cards. This project limits itself to supporting cards already formatted to FAT32, so a user might occasionally have to format a card before using it, but this is simple as most operating systems provide that functionality.

FAT[10] is an acronym for "File Allocation Table" and refers to the table that the file system uses to organize files and folders, while 32 is reference to the size of entries in this table. Common for all FAT file systems is that they segment the space of a disk into *clusters* which in turns are divided into *sectors*. For FAT32, the typical size of a sector is 512 bytes and a cluster consists of either 1, 2, 4, 8, 16, 32, 64 or 128 sectors. Both cluster and sector addresses begin at zero.

An important thing to note about the FAT file system is that it represent data in little-endian format. This means that for multi-byte values, the LSB (Least Significant Byte) is stored last in memory. That is the opposite of Patmos which operates with the big-endian format and has the MSB (Most Significant Byte) last. Therefore it is important that the implementation converts between the two formats when exchanging data.

3.7.1 Structure of a FAT file system

Figure 8 shows the structure of a FAT volume. The "Root Directory" region does not exist on FAT32 volumes, so it will not be discussed further.

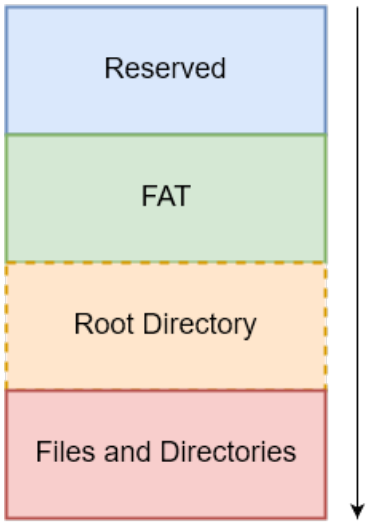

Figure 8: General structure of a FAT volume

Reserved region At the very beginning of the volume, in the first sector of the reserved region (and the volume) is located the "BIOS Parameter Block" (BPB). This sector contains information about how the volume is formatted, including the size of clusters and sectors. Table 6 shows the most important fields of this structure. The byte ranges are inclusive, so bytes 11 - 12 refer to a 2-byte value that occupies byte 11 and 12. Fields up to and including byte 35 are present on all FAT volumes, while the rest are present only on FAT32. For a detailed listing of all the fields, we refer to the FAT specifications[10, Page 9].

Name	Bytes	Description
`BytesPerSec`	11 - 12	Number of bytes per sector. Usually 512.
`SecPerClus`	13	Number of sectors per cluster. Always a power of two and ≤ 128.
`ReservedSecs`	14 - 15	Number of sectors in the reserved region. Usually 32 for FAT32.
`NumFATs`	16	Number of FATs. Usually 2 to handle data corruption in one of them.
`FATSize`	36 - 39	Number of sectors in a FAT.
`RootCluster`	44 - 47	Cluster index of the root directory.
`FSInfoSec`	48 - 49	Sector index of the `FSInfo` structure in the reserved region. Usually 1.

Table 6: Relevant fields in the BPB

FAT region After the reserved region is the FAT region. This contains the FAT(s) and is located at sector `ReservedSecs` in the volume. Usually there are two copies of the FAT, but this is dictated by `NumFATs`. How the FAT works is explained in Section 3.7.2.

Files and directories region The last region is where file data is stored, as well as the directory entries that make up the folders of the system. All data here is aligned in clusters, but note that neighbouring clusters are not necessarily related.

3.7.2 The File Allocation Table

The contents of a file in a FAT file system is stored in the clusters belonging to that file. The clusters of a file are not (necessarily) sequential in memory, but are chained together as a singly-linked list. If for example a file "`file0.txt`" occupies three clusters and begins at cluster 9, the chain may go $9 \rightarrow 13 \rightarrow 7$. The beginning of this chain is stored in the directory entry of the file (explained in Section 3.7.3), while the links are stored in the FAT. Every entry in the FAT is 32 bits wide and is either the index of the next cluster in the chain or a status value. Even though the entry is 32 bits, only the 28 lowest bits should be used as the rest are reserved. Table 7 provides an overview of the possible values.

Table Value	Meaning
$=$ `0`	Cluster is *free*.
$\geq$ `0x0FFFFFF8`	Cluster is the last in the chain.
$=$ `0x0FFFFFF7`	Cluster is marked *bad* and should be avoided.
$<$ `0x0FFFFFF7`	Next cluster in the chain is *in* the entry value.

Table 7: Possible values of a FAT entry

The FAT is indexed simply by the index of a cluster. The entry (next cluster or status) for cluster i is stored in the i'th entry of the FAT. If there is a next cluster in the chain, it is simply the lowest 28 bits of the entry. A cluster being marked free means that the cluster does not belong to any file and can freely be claimed by new or expanding files. If a cluster is marked bad, it indicates that the cluster is prone to read / write errors and should be ignored by the file system.

Figure 9 shows an example of how a file could be stored in a FAT. On the left side is a visualization of how the file contents could be stored and on the right side an example of how the FAT could look. The "EOF" (End Of File) is an entry value >= `0x0FFFFFF8` and indicates that the cluster is the last in the cluster chain. Zeroes mark free entries and three dots are entries with irrelevant contents.

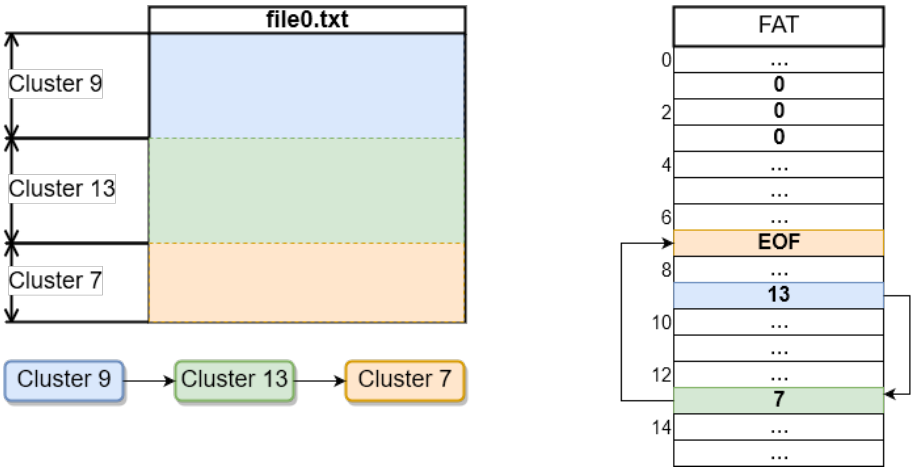

Figure 9: Example of a file stored in a FAT

3.7.3 Directories

Directories are just like files with regard to the FAT. They have a start cluster and may span multiple clusters linked in a chain, just like files. Instead of file data however, the clusters contain a list of *directory entries*, which are also referred to as "short entries" in this thesis. A directory entry is a 32-bit structure that contains information about a file or directory. Figure 10 shows the structure of a directory entry and Table 8 contains a listing of the fields within. The table is heavily based on the "FAT 32 Byte Directory Entry Structure" table in the FAT specifications[10, Page 23].

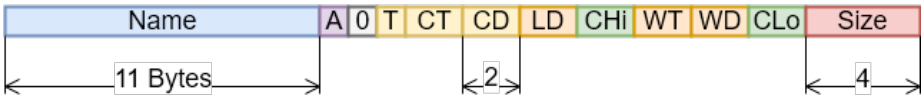

Figure 10: Structure of a directory entry

The first byte The first byte (byte 0) of a directory entry is special, in that it informs about the entry's status. If this byte indicates that an entry is free then the rest of the fields must be ignored. Table 9 shows the possible values of this byte along with their meaning.

Name The `Name` field in a directory entry stores the short name (see Section 3.7.4) of the file or directory it represents. The first 8 bytes are the ASCII[11] representation of the (short) name of the file. A byte that represents an ASCII

Name	Bytes	Description
`Name`	0 - 10	Name of the file / directory.
`Attrib`	11	Attribute of the file.
`Res`	12	Reserved and set to 0.
`CTimeTen`	13	Millisecond at time of creation.
`CTime`	14 - 15	Time of file creation.
`CDate`	16 - 17	Date of file creation.
`LDate`	18 - 19	Date of last file access.
`Clusigh`	20 - 21	Two **most** significant bytes of the files first cluster.
`WTime`	22 - 23	Time of last write to file.
`WDate`	24 - 25	Date of last write to file.
`ClusLow`	26 - 27	Two **least** significant bytes of the files first cluster.
`FSize`	28 - 31	Size of the file in bytes. Set to 0 for directories.

Table 8: Fields of directory entry

Value	Meaning
`0xE5`	Directory entry is *free*.
`0x00`	Entry is free and there are no occupied entries beyond it.
`0x05`	A regular directory entry where the first character is `0xE5`.
Otherwise	A regular directory entry.

Table 9: Possible values of the first byte in a directory entry

character is from here on just referred to as a *character*. The last 3 bytes are the file extension, if it exists for the file. No characters in this field are allowed to be lower case[6]. The complete (short) name of the file is then the name characters and, if the file extension exists, a dot and the file extension. Any empty characters, including inside the file extension, are represented with the value `0x20` which is an ASCII space[7]. Figure 11 shows how the file names "`file0.txt`" and "`file1`" would be stored. Some bytes are forbidden in any part of the name, but for these we refer to the specifications[10, Page 24].

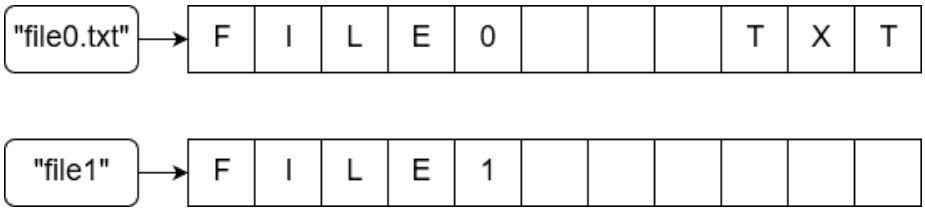

Figure 11: Examples of the storage of short file names

[6]The reason is that the lower-case representation is country-specific for some characters.
[7]As in what is produced by pressing the "space bar" on a keyboard

Attribute The `Attrib` field is a single byte which indicates the type of the entry. Each bit has a specific meaning and can be set in almost any combination. The bits and their meaning is listed in Table 10.

Bits	Hex	Name	Meaning if set
0	`0x01`	`AttReadOnly`	File is read only.
1	`0x02`	`AttHidden`	File is hidden.
2	`0x04`	`AttSystem`	File is a system file.
3	`0x08`	`AttVolID`	Entry represents the ID of the volume.
4	`0x10`	`AttDir`	Entry represents a directory.
5	`0x20`	`AttArchive`	File has been modified. Used by utilities.
0, 1, 2, 3	`0x0F`	`AttLong`	Entry is part of a long name entry chain.

Table 10: Meaning of `Attrib` bits

Time and date Most of the entry consists of time related fields. Of all the fields however, only the "last write" fields (`WTime` and `WDate`) are required by the specifications. The time and date format is quite interesting as it is very compact, but it is not explained here and we refer to the FAT specifications[10, Page 25]. This is because none of the fields are supported by the implementation (see Section 6.3.2).

File size The last field, `FSize`, is 4 bytes wide and stores the size of the file, or zero (`0`) in case of a directory. It must always have the correct size of the file and must thus be updated when the file changes size. This means that even a very small change to file, for example adding a character to the end, can require two reads and two writes to the disk: One for the file contents and one for the directory entry. A limit inherent to the FAT32 format, is that file sizes must fit in these four bytes. Therefore no files larger than $2^{32} = 4294967296$ bytes or roughly 4 GB can exist.

3.7.4 Long names

As is explained in Section 3.7.3, there are only 11 bytes available for the file name in a directory entry. Only 8 of these are for the name itself, while the rest are dedicated to the file extension. Furthermore, the name is always stored in uppercase in a directory entry. To get around these limits, the FAT32 format employs what is called "long directory entries", also referred to as "long entries". It is a special type of directory entry that can store part of the full (long) name of a file, while still being compatible with systems that only support short directory entries. Table 11 shows an overview of the fields of a long entry. An important thing to note about the long directory entries is that they store 16-bit UNICODE[12] characters for the name, instead of 8-bit ASCII.

A short directory entry that has a long name can then have a chain of long directory entries preceding it, which is illustrated in Figure 12. Note how the

Name	Bytes	Description
`LOrd`	0	Ordinal of the entry. Last in chain masked with `0x40`.
`LName1`	1 - 10	First 5 name characters in the entry.
`LAttrib`	11	Attribute field of the entry. Must be `AttLong`.
`LType`	12	Must be zero, to indicate a long directory entry.
`LChksum`	13	Checksum of name in related short directory entry.
`LName2`	14 - 25	Next 6 name characters in entry.
`LRes`	26 - 27	Must be zero.
`LName3`	28 - 31	Last 2 name characters in entry.

Table 11: Fields of a long directory entry

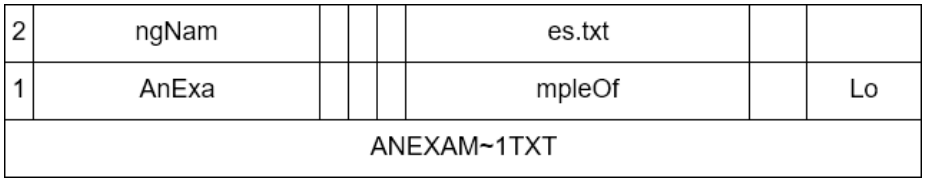

Figure 12: Example of a chain of long directory entries

short name entry (bottom) has a shorted version of the name stored with a *tail* at the end. The number on the left is the ordinal of the entry. The long entry immediately preceding the short entry has ordinal `LOrd = 1` and then it counts up. The last entry in the chain must have its ordinal masked with `0x40`. In the example in Figure 12, the second and last entry would have stored `LOrd = (2 | 0x40) = 0x42`.

Short name generation In the short entry that follows long entry chain, a short name must still be stored. This short name must be unique in the directory, which is also true for the long name. The specifications give suggestions to how this short name should be generated and we refer to those for more details[10, Page 30]. In List 1 is given an outline of what must be done.

1. Strip leading periods and all spaces from the name.
2. Store the first 6 characters of the long name in uppercase. We call this *basis-name*.
3. Store the first 3 characters of the extension in uppercase. We call this *basis-ext*.
4. Find a number n such that the name *basis-name* $+ \sim +$ n $+$ *basis-ext* is unique in the directory.
5. If n does not fit in the remaining bytes of the `Name` field, remove one character from the end of basis name and search again, until a max of $n = 999999$.

List 1: Outline of short name generation

An important note is, that there are no strict requirements for how the tail number n should be selected. One might expect that it always begins with 1, then 2 and so forth, but this is not required.

Checksum The `LChksum` field in a long directory entry is a 1-byte field that must contain a checksum, calculated from the bytes in the short name of the file. As such it is the same for all long entries in the chain. If it is not correctly calculated the file system should ignore the entries. The formula for the checksum is:

$$C_{short}(S) = \sum_{c \in S} \text{rrot}(c) \mod 255 \tag{1}$$

Here S is the string in question and "rrot" is a right-rotate operation on a byte. Thus the checksum is simply the sum of all the bytes rotated right one bit. A simple implementation of right-rotation of a byte in C is:
`rx = (x >> 1) + (x << 7)`.

3.7.5 Limitations of FAT32

FAT32 being a fairly simple file system, carries some implications that might not be immediately obvious. First of all there is the strict limit on file sizes, as already mentioned. Secondly, there are no indices and no sorting of directory entries in the format, which means that to find a file in a directory, all entries in the folder will have to be checked, in the worst case. This results in a worst case time complexity for path resolution that grows linearly with the number of entries in the folder. Thirdly, FAT is case-insensitive which means that "file.txt" and "FilE.TXT" are the same name in such a system.

4 Design

4.1 Structure

The three parts of this project can be viewed as three stacked layers, where each layer provides functionality to the layer above it. The bottom layer is the physical layer, in which the host controller resides. This layer facilitates the physical communication with the SD card. The next layer is the driver, which interacts with the host controller to allow for transferring blocks of data to and from the card. Above that there is the file system layer, which uses the driver to enable writing and reading of files. It is on top of this that the application layer resides. Figure 13 illustrates this model.

These layers are designed to be independent of each other. It is possible to switch out the file system module for any other file system module, as long as it can work with the generic interface specified in 4.4. This allows the individual parts to be reused in future work, for example using the file system module for an external USB hard-disk formatted to FAT32.

The driver and host controller are not completely independent however, which is why they are grouped together in the figure. This is because the driver depends on the host controller and as such, if the host controller is switched out then so must the driver. While it is possible to decouple the host controller and the driver by utilizing a generic interface, this is not done in this project. It seems unlikely that a new driver will be written for the implemented host controller, instead of designing a new host controller that supports SD mode.

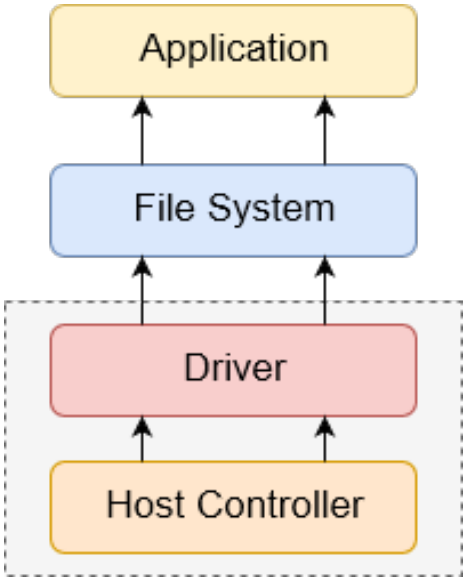

Figure 13: The module layers of the project

4.2 Host Controller

A few choices had to be made, in regard to the use of the SPI protocol in the host controller. These are outlined in the following section.

4.2.1 Buffering

First, it should be noted that it is possible to "bit-bang" the SPI protocol. Since there are no requirements for the clock rate, one could just connect a register directly to the pins and access that register from code. One would then set the MOSI bit in the register, flip the clock bit, read out the MISO bit and flip the clock bit again, to constitute a complete clock cycle. This is very slow however, even for the SPI mode.

Therefore it was decided to include buffers for the host controller. Data to be transferred is placed in an outgoing buffer register, while incoming data is placed in another. Both these buffers are 8 bit large, since the protocol is byte-based. Another sensible size would have been 32 bits, since that is the default word size of Patmos and the size of `MData` and `SData` (see Table 4). A register is then needed to keep track of which bit in the buffers is currently being transferred. This register needs to be large enough to contain the size of the buffer registers, which in the 8-bit case is at least 4 bits. This pointer register then increments every `SCK` cycle. Figure 14 shows how this structure looks, when the 6th bit is about to be transferred.

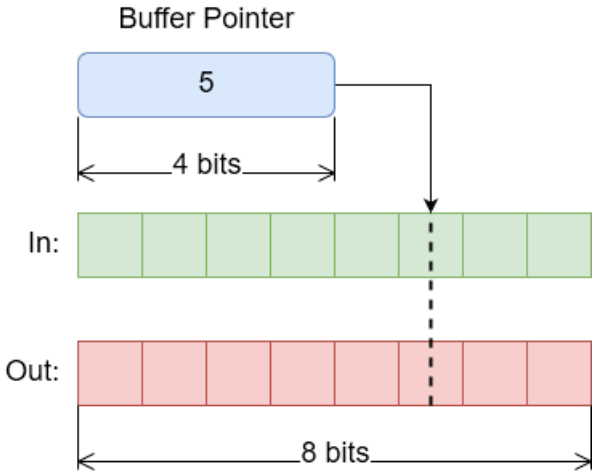

Figure 14: Structure of transmission buffers

4.2.2 Transferring data

The host controller has two states: "idle" and "active". In the idle state it simply waits for a data transfer to be initiated. It holds `SCK` *low*, ensuring that

nothing happens in the slave. When a byte is received from the driver, it saves it to the outgoing buffer register, resets the buffer pointer and enters the active state which initiates a transmission. The transmission is performed by holding `MOSI` to the value of the i'th bit of the outgoing register in the i'th cycle of `SCK`, which is handled by the buffer pointer register. On the falling edge of `SCK`, `MISO` is sampled and stored in the i'th bit of the ingoing register. The driver can then read the value of the ingoing register.

4.2.3 Ongoing transmissions

It is important that the host controller is not interrupted while a transmission is ongoing. That means that no new data must be written to the outgoing buffer during a transfer. Nothing should be read from the ingoing buffer either, since that data would be incomplete and thus meaningless.

Two approaches were considered for ensuring this. First, the OCP protocol could be utilized. By withholding the `DVA` response until the transmission is complete, the CPU can do nothing but wait. The other approach is to have an exposed register indicate whether a transmission is ongoing, which the driver can poll. If the next operation to be performed after writing is another write, then the first approach would be slightly faster as the CPU could immediately execute the write upon completion. In polling approach, the CPU would have to first execute a read for transmission register, then a comparison and possibly a branch instruction, before reaching the next write. If the next operation is not a write however, the polling approach is faster, as in that case the CPU is free to perform other instructions while the write happens.

Ultimately, the polling approach was decided upon.

4.2.4 Clock rate

The rate of `SCK` must be variable. During the initialization phase of the SD card, the clock rate must not be higher than 400kHz [3, Section 4.2.1], but a higher rate is wanted for data transfer. To allow for this the host controller permits clock rates that are even divisors of the CPU clock rate. The host controller has a register initialized to such a divisor and then every CPU clock cycle it counts down in that register. Upon reaching one, `SCK` will switch and the counting register will reset to the divisor. For the default clock speed of Patmos on the Altera DE2-115 board, which is 80MHz[8, Table 2.15], this results in permitted `SCK` rates:

$$F_{sck} = \{r \mid r = (1/i) \cdot 80\,\text{MHz}, i \in \mathbb{N}, i > 1\} \tag{2}$$

$$= \{40\,\text{MHz}, 20\,\text{MHz}, ..., 400\,\text{kHz}, ...\} \tag{3}$$

The reason that $i > 1$ is that the host controller updates every clock cycle of the CPU, which at a maximum will flip clock signal every full cycle of the CPU, resulting in half the rate of the CPU.

4.2.5 Interface

The driver interacts with the host controller purely by reading and writing registers. In Table 12 is listed the registers that are exposed to the driver. Note that the offset is in 4-byte words such that offset 2 from byte 10 is byte 18. Also note, that the `buf` register is actually split into two registers, `bufIn` and `bufOut`, in the implementation.

Register	Offset	Value on read	Action on write
`buf`	0	Last byte read from card	Send a byte to the card
`cs`	1	-	Set the chip select pin
`en`	2	Non-zero if ready for data	-
`clkdiv`	3	-	Set the clock divisor

Table 12: The exposed registers of the host controller

4.3 Driver

The driver has to interact with the host controller and provide read / write functions to the library layer. A primary aim for its interface is for it to be as generic as possible without being inefficient, to allow other types of disks to adopt it.

4.3.1 Initialization

An initialization function `disk_init` is necessary for the driver to function. It is in this function that it must initialize the SD card if available and the user must call this function once before using the disk, to ensure that the disk is ready.

The function does not need to take any configuration parameters. The user should not be required to know anything about the disk, which would need to be passed in through arguments. If anything, information should travel from the initialization function to the user. Therefore, the function must both output a return value that indicates whether the initialization was a success or not, as well as fill a `DiskInfo` data structure with information about the disk. In any implementation of this interface where no initialization is necessary, the `DiskInfo` struct should still be filled. The `DiskInfo` struct contains information about the disk relevant to the file system. It only has one field, `blocksz`, but a struct was chosen anyway to ensure type safety and to accommodate future extensions. The field `blocksz` indicates the size of a block on the disk, which is necessary information for the file system, so it can adjust how many reads or writes it must issue.

4.3.2 Read / write

For the read and write functions `disk_read` and `disk_write`, some choices were made about the parameters. The initial approach was to have them take pa-

rameters for a byte position on the disk, a data buffer to read from / write into and a byte count that indicates how much should be transferred. This is very intuitive and makes it easy to write only a few bytes to the disk. If for example the file system wanted to update the file size in a directory entry, it would be very simple with this interface.

However that approach does not fit very well with the block based model of an SD card and the sector based model of the FAT file system. In the case that only part of a block needs to be written to, as in the example above, the `disk_write` function would have to read out the block on the card, change the bytes and write it back, as was discussed in Section 3.4.7. It was found during implementation, that more often than not, the file system would have already read out the sector that the write was to happen in. When for example creating new directory entries, the sectors of the directory have just been searched through. Therefore, having the interface method also read out and write back the sector would be unnecessary and inefficient. To avoid having the driver repeat the read, the function caller would the have to align byte address with the containing block. Instead of doing this, it was decided to modify the interface to work on blocks instead. So instead of a byte index it takes a block index and instead of a byte count it takes a block count. The size of blocks are then dictated by `blocksz` field in `DiskInfo`, which is set upon disk initialization. This interface forces the file system module to only read and write in blocks, which also helps keeping the number of reads and writes low when developing, as they are not hidden. A nice coincidence (probably not) is that the usual block size on an SD card and the usual sector size of FAT partition is the same, 512 bytes.

4.4 Interface

The interface to the driver then looks as shown in Table 13. All functions return an integer which indicate success and they all set `errno` (more on this in Section 5.1.1. Note that this interface does not expose the fact that an SD card is being used for the disk.

Function	Description
`disk_init(*inf)`	Initialize the disk and write configuration to `inf`.
`disk_write(pos, buf, n)`	Write `n` blocks from `buf` to disk, starting at block `pos`
`disk_read(pos, buf, n)`	Read `n` blocks from disk, starting at block `pos`, into `buf`

Table 13: The interface of the driver layer

4.5 FAT module

The FAT module must utilize the the driver to provide a pleasant-to-use interface for interacting with files on the card. List 2 shows an overview of functionality one might expect from such a library.

1. Reading from and writing to files.
2. Creating and deleting files.
3. Creating and deleting folders.

List 2: Expected functionality of file system module

While there are a lot of ways to provide this functionality, it was decided to attempt to mimic the interface of a subset of C standard library system calls, as they are defined in the POSIX standard[13].

Most importantly, this should be very familiar interface for C programmers. It can be expected that people who have written C for some time, are very used to interacting with files using file descriptors (see Section 4.5.1). Secondly, if the necessary system calls are in place, the "Newlib"[8] port that T-Crest uses can be directed to use the library to provide the usual file-related functions of the C standard library, `fopen`, `fputs`, etc. This would mean that existing code that uses these functions could be run on T-Crest / Patmos with an SD card attached. However this is not done in the current implementation.

Table 14 lists the set of exposed functions from the file system layer. The function names and signatures are chosen to closely match the system calls they mimic, except the partition and initialization functions as they have no system call counterpart. Some terms might be unclear, like "descriptor" and "cursor", but they will be explained shortly.

4.5.1 File descriptors

The first thing to note about this interface is that everything works on *file descriptors*. A file descriptor is a non-negative integer that refers to an open file[9] data structure, that contains information about the file in question.

In the C standard library there are three reserved file descriptors. The standard input pipe `STDIN_FILENO = 0`, the standard output pipe `STDOUT_FILENO = 1` and `STDERR_FILENO = 2`, which is the standard pipe for errors. All other positive integers below a configurable maximum, are available for file descriptors.

Normally the operating system keeps track of the set of file descriptors and open files[14]. However, in our case there is no operating system available, so the module must handle this itself. This is one of the reasons that it is necessary for the file system module to have an initialize function, which is not normally

[8] See `https://sourceware.org/newlib/libc.html`

[9] Technically also pipes and streams, but this is irrellevant for this project.

Function	Description
`fat_load_pinfo(i)`	Load the `i`'th partition on disk.
`fat_load_first_pinfo()`	Load the first FAT32 partition on disk.
`fat_init(pinfo)`	Initialize the file system module using the partition info `pinfo`.
`fat_open(path, oflag)`	Open the file at `path` according to `oflag`. Returns a file descriptor on success.
`fat_close(fd)`	Close the file with descriptor `fd`.
`fat_read(fd, buf, sz)`	Read `sz` bytes from file with descriptor `fd` into `buf`. Returns the number of read bytes.
`fat_write(fd, buf, sz)`	Write `sz` bytes from `buf` into file with descriptor `fd`. Returns the number of written bytes.
`fat_lseek(fd, pos, w)`	Set the cursor of file with descriptor `fd` to `pos` according to `w`.
`fat_unlink(path)`	Delete the file at `path`.
`fat_rmdir(path)`	Delete the directory at `path`.

Table 14: Interface of the file system layer

necessary when working with files in C. The way these files are managed in the file system module, is simply an array of open-file structures. A file descriptor is then simply an index into this array, offset by three to account for the reserved descriptors. This structure enforces a compile-time constant size of the array and thus a constant maximum of open files.

4.5.2 Open files

It is not necessary for this project to support all the functionality that is normally associated with files. Following is a discussion of the information that is / is not associated with the open-file structure in this project.

Permissions It was decided not to implement permissions for files in this project. Permissions allow the file system to mark files, such that some operations will fail for it, like writing to read-only file. Users can also open files in specific modes, such as read-only or write-only. Attempting to perform an illegal operation on a file, e.g. writing to a read-only file, would result in an error.

While this is very useful functionality, it is not strictly necessary. Therefore, it was decided against and could instead be an easy extension to the file system module in future work. When it is to be implemented, such information should be stored in the file descriptor structure, as it is necessary to consult before every read or write.

Availability When opening a file, the returned file descriptor should be the lowest possible integer that is not already in use. Finding such a number is simply a matter linearly searching from the start of the open-file array. Since the file descriptor must not already be in use, it is necessary to mark its availability somehow. It was chosen to simply do this by storing a `free` flag in the open-file structure that is either zero (taken) or one (free).

Cursor Files in C are expected to have a cursor associated with them. A cursor is the current position in the file, from where all reading and writing begins. Upon reading or writing, the cursor moves forward according to the number of bytes read or written. Figure 15 shows an illustration of this. This model of moving forward in the file, fits well with how files are stored in a FAT file system. As cluster are singly-linked and pointing forward, it is much easier (faster) moving forwards than moving backwards, as that would require searching from the start of the cluster chain.

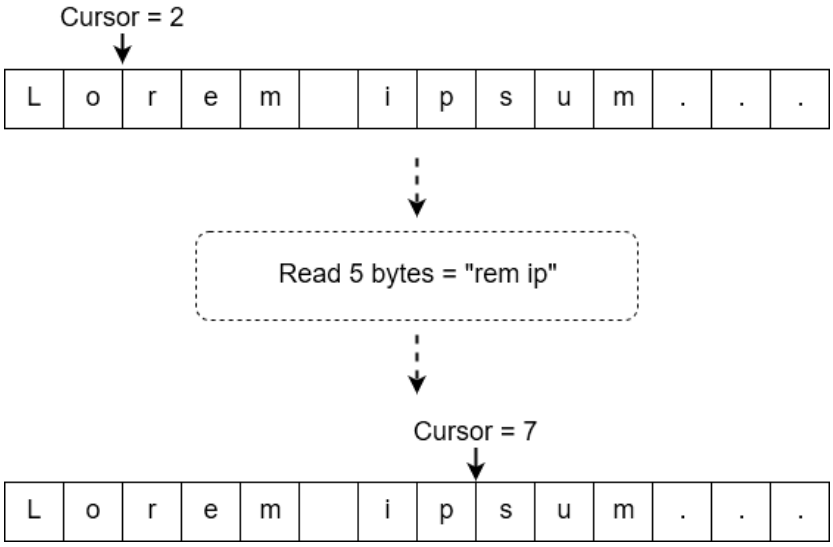

Figure 15: Illustration of how a cursor in a file works

Position of directory entry Whenever a file changes size or is written to, there must be written to fields in its directory entry (`FSize` and `WTime` and/or `WDate` respectively). For this reason, it is necessary to store the position of its directory entry in the structure. Since the only way to open a file is through its directory entry, the position is known at that time.

Size The size of a file is relevant every time a read or write happens. A read operation must not read past the end of a file and a write operation past the end of a file requires, that the size of the file be adjusted and maybe even a new

cluster be reserved for the file. The size of a file is stored in its directory entry and therefore could be read from there. However, by storing it in the structure (which is in memory) we can avoid having to read the directory entry from disk every time. For this reason, it was decided to store the file size in the structure as well.

First cluster When the user wants to move the cursor backwards, it is necessary to begin from the first cluster in the chain and move forward. As with the file size, the first cluster in the chain is stored in the directory entry, but it was chosen to keep it in memory too to minimize disk reads. The case is less strong here than for the file size, since seeking backwards in a file probably happens much less than reading and writing, but the memory cost of 32 bits was deemed worth it.

5 Implementation

This section details how each part of the design was realized. See Appendix C for an overview of all the files related to the implementation. On the Ubuntu development image, the full implementation can be built, synthesized to the board and run from the "`patmos`" folder with the command:
`make gen synth comp config download APP=sdtest BOARD=altde2-115-sd`

5.1 Coding in C

Some choices in the implementation are relevant to both the driver and the file system. Following is a short explanation of these.

5.1.1 Error Handling

When errors occur in the code they must be identified and dealt with. A distinction is made between expected errors, such as a search function not finding its target, and errors due to user input, such as attempting to delete a file that is not there. In the implementation, expected errors are generally indicated by the return value of the function. The function caller then inspects this value before interacting with any output values. This is a simple approach that is nice to work with.

For errors occuring in the file sytem interface functions however, the implementation sets the global `errno` variable, which is defined in the standard library header `errno.h`. This is how the system calls, that the interface is modelled on, work. The caller is then to inspect `errno` after each function call, where `errno == 0` indicates success and anything else indicates failure. No other values are used but those defined in `errno.h` and the interpretation of each value depends on the context. See Appendix D for an overview.

5.1.2 Integer types

Working with FAT32 involves using a lot of unsigned integers with sizes from 8 to 32 bits. To avoid an inordinate amount of `unsigned` keywords in the code, while also being strict with using the correct types, it was decided to use the `uint8_t`, `uint16_t` and `uint32_t` types defined in the standard library header file `stdint.h`. The interface functions still use `int` and `off_t` however, to match the system calls.

5.2 Host Controller

A Chisel component was created called `SDHostCtrl`, located in the code file `SDHostCtrl.scala`. It has three output pins, from host controller to card port, and two input pins. Table 15 shows an overview of the pins. Notice the direction of data pins. The output pin on the host controller is the input pin on the card and is named `sdDatIn` to match the card semantics. Also notice that the write protection pin is ignored.

Name	Direction	Description
`sdClk`	Output	Clock signal.
`sdCs`	Output	Chip select signal.
`sdDatIn`	Output	Input data signal for card.
`sdDatOut`	Input	Output data signal for card.
`sdWp`	Input	Write protection pin from card. Ignored.

Table 15: Pins for host controller component

5.2.1 VHDL / Verilog

The Chisel code is compiled by the `make` build system of the platform. This generates, among other things, a Verilog file "Patmos.v" for the complete processor and components. In this file is found Verilog code for the `SDHostCtrl` component. A VHDL file "`patmos_de2-115-sd.vhdl`" is present in the project directory, which glues the components together and it is in here that the connections of the Verilog file are connected to the processor.

Both of these files are referenced in the Quartus project file "patmos.qsf" and used when the processor, along with the SD host controller, is synthesized to the board.

5.2.2 OCP signals

All communication between the CPU and the host controller happens through the "OCPcore" interface (see Table 4). The host controller, being the slave, observes `M.Cmd` to await read or write commands and then inspects `M.Addr` to determine which register is to be accessed, initiating a transaction if necessary. Any read or write puts `DVA` (Data Available) on `S.Resp` the next cycle. This includes reading and writing to invalid addresses (not associated with a register) or writing to a read-only register.

5.2.3 Registers

Table 16 shows an overview of the registers in the host controller. The "R/W" describes whether the registers can be read (R) from or written (W) to by the driver. A dash indicates that the register is internal to the component and can not be accessed by the driver.

5.2.4 Clock signal

In Chisel there is no explicit clock signal. Updates to registers utilize the implicit clock signal, such that an assignment to a register can be expected to have effect the next clock cycle. The implicit clock signal in the host controller component has the same frequency as the CPU, which is 80 MHz. As mentioned in the design, this signal is downsampled to a variable frequency in the host controller. This is done with three registers. First is the `clkReg` register, which is directly connected to the `sdClk` pin. The `clkDivReg` register holds the divisor of the

Name	Bits	R/W	Description
enReg	1	R	Is a transaction active?
bufInReg	8	R	Buffer from card to host controller.
bufOutReg	8	W	Buffer from host controller to card.
bufPntReg	8	-	Points to currently transmitting bit.
clkDivReg	16	W	Divisor of clock rate.
clkCntReg	16	-	Counts to divisor of clock rate.
clkReg	1	-	Clock signal to card.
sdCs	1	W	Chip select signal to card.
ocpDataReg	32	-	Holds data to be returned from reads.
ocpRespReg	2	-	Holds OCP response.

Table 16: Registers in host controller

implicit clock signal and can be written to by the driver. The `clkCntReg` register counts down from `clkDivReg` to one, updating every implicit clock cycle. When `clkCntReg` reaches one it is reset to `clkDivReg` and `clkReg` is flipped. This produces a downsampled clock rate for `sdClk`. If for example `clkDivReg = 100` and the implicit clock rate is 80 MHz, `clkReg` and therefore `sdClk` will have a frequency of $80\,\text{MHz}/(2 * 100) = 400\,\text{kHz}$.

This clock generation only happens when a transaction is active, which is when `enReg` is not zero. If a transaction is not active `sdClk` is held low.

5.2.5 Transactions

A transaction is begun when the driver writes to the `bufOutReg` register. When this happens, the following is done:

- Set `bufOutReg = io.OCP.M.Data` to prepare for sending the data.
- Set `bufInReg = 0` to clear the register and prepare for receiving.
- Reset `bufPntReg = 8` to prepare sending least significant bit first.
- Reset `clkCntReg = clkDivReg` to reset the clock signal generation.

While a transaction is active a steady clock signal is sent to the card over `sdClk`. On the falling edge of `sdClk`, bit (`bufPntReg` - 1) of `bufInReg` is set to the value of the `sdDatOut` pin, which constitutes the sampling of the card. At all times is `sdDatIn` set to bit (`bufPntReg` - 1) of `bufOutReg`. When a full clock cycle has been generated, `bufPntReg` is decremented by one and upon reaching zero, the transaction is complete and `enReg` is set to low again.

5.2.6 Pin assignment

The pins of the boards SD card slot were assigned to the pins in `SDHostCtrl`. This was done in the "Pin Planner" tool in Quartus. They all operate with 3.3 V and 8 mA. Figure 16 shows a screenshot from the Pin Planner tool in Quartus. Here can be seen the name of the pins on the board (Location), the

names of the pins in the VHDL code (Node Name) as well as the voltage and power levels. The names of the pins on the board were read from the manual of the board[15, Table 4-31].

Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength
iSDHostCtrl_wp	Unknown	PIN_AF14	3	B3_N0	3.3-V LVTTL		8mA (default)
iSDHostCtrl_do	Unknown	PIN_AE14	3	B3_N0	3.3-V LVTTL		8mA (default)
oSDHostCtrl_clk	Unknown	PIN_AE13	3	B3_N0	3.3-V LVTTL		8mA (default)
oSDHostCtrl_di	Unknown	PIN_AD14	3	B3_N0	3.3-V LVTTL		8mA (default)
oSDHostCtrl_cs	Unknown	PIN_AC14	3	B3_N0	3.3-V LVTTL		8mA (default)

Figure 16: Screenshot of pin assignment in Quartus "Pin Planner"

5.2.7 Configuration

Inside the project directory is an XML configuration file "`altde2-115-sd.xml`". In here it is specified which devices are to be built with the processor and how they are configured. The configuration for the `SDHostCtrl` component is minimal and only specifies that it is located at offset 11 and uses the "OCPcore" interface. Using this offset results in the memory locations specified in Table 17.

Registers	R/W	Address
`bufInReg`	R	`0xf00b0000`
`bufOutReg`	W	`0xf00b0000`
`csReg`	W	`0xf00b0004`
`enReg`	R	`0xf00b0008`
`clkDivReg`	W	`0xf00b000c`

Table 17: Memory locations of host controller registers

5.3 Driver

The driver is implemented in the code files `sd_spi.c` and `sd_spi.h`. The functionality of the driver is wrapped in generic disk functions, which are implemented in the code files `sddisk.c` and `sddisk.h`.

Internally in the driver errors are indicated by the return value of functions. Functions return a `SDErr` value which is an `enum` that indicate different error scenarios. The disk interface functions all use the `errno` approach however.

5.3.1 Sending bytes

The most basic operation of the driver is to send and receive a byte of data to the card. This is performed by the `spi_send` function, which takes a byte as its argument and returns the received byte. Sending a byte to the card is done by placing it in the exposed `outBufReg` register with a write to its memory

location. As the byte is sent, the `inBufReg` is simultaneously filled with data from the card. The host controller does not prevent a transaction from being interrupted, so this is the responsibility of the driver. If a transaction is active, the memory-mapped `enReg` register will contain a non-zero (one) value. As such, the entirety of the function can be summed up as: Wait for transaction to be done, write the output byte (argument), wait for that transaction to end, read and return the input byte.

It can be argued that if this function is the only function to initiate transactions, and the function waits for a transaction to be done after starting it, then it is superfluous to wait for transactions in the beginning of the function. However, it was chosen to leave it in since it is a minimal time loss and ensures that no transaction is ever interrupted, even if future development breaks the single-entry contract[10].

5.3.2 Issuing commands

All interaction with the SD card happens through SD commands, which were explained in Section 3.4.3. Sending commands and receiving responses is done by the `sd_cmd` function. The function takes 6 individual bytes as input: The index of the command `cmd`, the chunks of the 32-bit argument `arg0 - arg3` and lastly the CRC7 of the entire command structure. It returns the 8-bit R1 response of the command.

As mentioned, the function takes the command index as a one-byte argument. However, the index is actually only 6 bits and the command structure sent must always begin with the bits `01`. Therefore, before sending, the command byte is bitwise OR'ed with `0b0100000 = 0x40`.

Besides that, it simply transfers the bytes in the provided order. After sending the command, the function waits for the card to return a response. The card only updates when the clock signal is provided, which only happens when a transaction is ongoing, so to receive the response the function sends dummy bytes with the value `255 = 0xff` to the card. This is the same as simply running the clock signal while holding the `sdDatIn` signal high. The contents of these dummy bytes could be anything, but holding the line high was chosen, as that is what the card does on the `sdDatOut` line inbetween responses and data.

Any response by the card will arrive a few, but variable amount of transaction cycles later and will always be aligned with a transaction cycle. Receiving a response is therefore done simply by waiting for the card to respond with a byte that is not `0xFF`, which is then the response.

CRC7 It can be argued that it would be simpler or cleaner if `sd_cmd` did not take in the CRC7 code of the command, but instead calculated it from index and argument. It needs to happen for every command and the current implementation requires that the function caller calculates it instead. However, this was decided against because the SPI mode of the SD card has CRC checking

[10] This is an example of "defensive programming"

disabled by default, and for performance and simplicity's sake it was left disabled. It is therefore only needed in a few commands when initializing the card and there it can be (and is) statically calculated. Thus by providing the CRC7 in the argument, the caller can simply provide a dummy code when it is no longer necessary.

Clearing buffers It was discovered during implementation, that the card would not respond correctly when sending multiple commands in succession. The issue seemed to be, that the internal buffers in the card would contain data from dummy bytes of the previous command, leading the card to understand the next command wrong. This was worked around by clearing the buffers before every command. The function `spi_clear` does this. It holds the `sdCs` signal high (to ensure the card does not react) while transferring a dummy byte. This function is called in very beginning of `sd_cmd`.

5.3.3 Setting the clock rate

The rate for the cards clock signal is adjustable. This is done by writing to the memory mapped divisor register `clkDivReg`. However, the host controller does not verify if this value is valid. That is the responsibility of the driver.

Setting the clock rate is done by the function `spi_set_clockrate`, which takes the target clock rate as its argument and returns an `SDErr`. The function first retrieves the clock frequency of Patmos, using the `get_cpu_freq` function provided by the header `machine/rtc.h` that works by accessing the memory mapped `CpuInfo` device. It then verifies that the target rate evenly divides the Patmos clock rate and that the target rate is at the most half the Patmos clock rate, which is the maximum possible. If any of these criteria are not met, the function returns an error and does not change the host controllers clock rate. If both are met, it calculates the divisor, which is $d_f = f_{\text{cpu}}/(2f_{\text{target}})$.

5.3.4 Initialization

Being able to send commands to the card, as well as adjusting the clock rate, is all that is necessary to operate the card. Before the card can be used it must be initialized, which is done by the `sd_init` function. It takes no arguments and returns an `SDErr`.

The initialization process of the card is outlined in Section 3.4.6. Following is a rundown of the implementation of this process.

1. First the clock rate is lowered to 400 kHz which is necessary during the initialization[3, Section 4.2.1]. The specifications for the physical layer dictate that the card should be have at least 80 clock cycles to initialize before the process is begun, which is done simply by sending $80/8 = 10$ dummy bytes, while holding the `sdCs` signal high.

2. Then the GO_IDLE command is sent. It has no arguments and the CRC7 for the command is calculated to be 0x4A, which placed in the upper 7 bits give 0x94. The response is then checked and it must be 1 = 0b00000001 to indicate it is in idle state. Anything else and the function aborts and reports the error, which can then be checked.

3. Then the SEND_IF_COND command is sent. The non-zero arguments here are arg2 = 0x01 to indicate the supported voltage range 2.7-3.6V and arg3 = 0xAA = 0b10101010 which is a constant check pattern. Again the CRC7 is static, this time calculated to 0x86.

The response of the SEND_IF_COND is a R7 response. The first byte is a regular R1 response and checked as such (0x01), while the remaining four bytes should be the echo-back of arg0 - arg3. This is verified and if anything is wrong, the initialization is aborted and an error reported.

4. The next step is the SD_SEND_OP_COND command. At this point the CRC7 is no longer necessary and we just send the dummy bits 0xff instead. This is an ACMD, which means that first a CMD55 is sent and then a "CMD41", which then contains the arguments to the ACMD. The only relevant argument to the command is whether to enable High Capacity Support or not, which is indicated by bit 6 in arg0. A value of arg0 = 0x40 indicates HCS and arg0 = 0x00 indicates no HCS.

After the command has been issued successfully, the card will return a regular R1 response. If the initialization is complete, it will indicate that it no longer is in the idle state (0x00). Otherwise it will that it is busy. It responds busy if either it is already performing the final initialization process or if it does not support HCS and it was requested in the argument. Therefore, the card simply repeatedly sends the command a fixed number of times (AMD41_MAX_TRIES = 1000, found experimentally), first with HCS requested and then without. This happens in the function sd_send_op_cond_cmd which as its argument takes a flag indicating HCS or not.

5. The last thing in initialization is setting the block length. This is done with the SET_BLOCKLEN command, which takes the requested block length as its 32-bit parameter. The only supported block length in this project is 512 bytes, which is represented across the arguments as: arg0 = 0x00, arg1 = 0x00, arg2 = 0x02, arg3 = 0x00. The response is a regular R1 which is checked. This block length is then saved in the DiskInfo struct.

6. After initialization is done, the clock rate is increased to the maximum allowed, which is 20 MHz.

5.3.5 Writing data

Writing is done by the function sd_write_single_block, which writes a block of data to the card and returns an SDErr. As its arguments it takes the target

block address on the card and a pointer to block being written.

The SD command for writing a block of data is `WRITE_BLOCK`, which as its argument takes the 32-bit block address of the write destination. After verifying the response of the card, a "Start Block Token" is sent followed by bytes of the data block. After the data has been sent, the driver waits for the card to respond with a "Data Response Token". This takes a variable amount of time and the driver waits (by sending dummy bytes) up to `SD_WRITE_MAX_WAIT` transaction cycles. If no response has been received by then, the function exits with a timeout error. Otherwise the token is inspected to see if the data has been accepted or rejected. If the data is rejected, then the function returns with an error indicating the reason. If the data response token does not fit any of the expected values, the functions also returns with an error indicating a bad response. After the data response token, the card responds busy by holding the `sdDatOut` line low, while the data is written to the card. The driver waits for this up to a maximum of `SD_BUSY_MAX_WAIT` transaction cycles. If this does not time out, then the write is complete and the function returns.

5.3.6 Reading data

Reading is done by the function `sd_read_single_block`. Like its writing counterpart, it takes the block address to read from along with a pointer to the array where the data is to be stored. It also returns an `SDErr`.

Reading is done by issuing the `SD_READ_SINGLE_BLOCK` command, which like its writing counterpart does it take a 32-bit block address as its argument. After the command has been issued and its response verified, the driver waits for a "Start Block Token" up to a maximum of `SD_READ_MAX_WAIT` transaction cycles. No response here results in the function terminating with a timeout error. Immediately following the token is the data which are then read and stored sequentially in the output array. The data block is appended with a 16-bit CRC16 code which is simply read and ignored. After this the read operation is complete and the function returns.

5.3.7 Generic interface

The functions just described are not meant to be used directly. Instead, they are utilized by the generic interface provided in `sddisk.h`, which the file system then interacts with. This interface is the one described in Section 4.4. If an error occurs in the driver functions, the interface functions set `errno = EIO` to indicate an I/O error.

5.4 FAT Library

The file system library is implemented in the files `fat32.h` and `fat32.c`.

5.4.1 Exit codes

Most functions in the file system module, both exposed and internal, return an integer to indicate success or failure. These codes will from here be referred to as *exit codes*. The meaning of exit codes are dictated by the definitions `FAT_SUCCESS = 0` and `FAT_FAIL = -1`. These values match the return values of the system calls mimicked by the interface.

5.4.2 Handling endianness

Values in FAT partitions are stored in little-endian format, but Patmos uses the big-endian format. This means that whenever data is read from or written to the disk, it must be converted between the two. Converting between the formats is achieved by reversing the byte order using bit-shifting. Extracting values is done by the functions `fat_get_uint8`, `fat_get_uint16` and `fat_get_uint32`, which take a pointer to where the data begins. Inserting values is done by the functions `fat_set_uint8`, `fat_set_uint16` and `fat_set_uint32`, which take the value to be inserted along with a pointer to where the data is to be stored.

Initially this functionality was contained in only two functions, which took an extra parameter that indicated the byte size of the data. This worked and was generic as it could handle any byte size. However, because these functions are heavily used, making them faster should be a priority. Therefore the specialized instances were created instead, in which the loops are unrolled and the functions inlined. This should allow the compiler to optimize the code much better[16].

5.4.3 Data structures

The important data structures of the file system implementation are listed in Table 18.

Name	Description
`FatPartitionInfo`	Holds information about the FAT volume.
`FatDirEntryIdx`	An index of a directory entry. Consists of cluster, sector and entry index.
`FatFile`	Represents an open file. Includes cursor, file size and location on disk.

Table 18: Central data structures of the implementation

5.4.4 Partition

Before being able to initialize the file system module, information about the FAT partition needs to be loaded and provided. This information is stored in a `FatPartitionInfo` struct and is read from the MBR on the disk (see Section 3.6). This is done by the function `fat_load_partition_info`, which accepts an index `idx` $\in [0, 3]$ and a pointer to the `FatPartitionInfo` struct that must be

initialized. It returns an exit code and if it indicates failure, any information in the pointed-to struct should be disregarded.

The provided `idx` is the index of the partition table entry of the target FAT partition. If the index is not known, the `fat_load_first_partition_info` function can be used instead, which simply tries all the entries in order. If `idx` is out of range the functions returns with an error. It reads the partition table entry, verifies that it points to a FAT32 partition and finds the beginning of the FAT32 volume. The first sector of the volume contains information about the partition, which is read and stored in the `FatPartitionInfo`. Some of the fields are not directly read from the disk, but calculated from the read values to avoid having to calculate them later.

5.4.5 Initialization

Before the file system module can be used it must be initialized. This is done by the function `fat_init`, which takes a pointer to a `FatPartitionInfo` as its argument and returns an exit code. In List 3 is listed the basic steps of the function.

1. Store global partition info.
2. Initialize the open files array, marking all descriptors free.
3. Flag the module as initialized.

List 3: Steps in file system initialization

Global partition info The function takes as its argument a pointer to a `FatPartitionInfo` struct `fat_pinfo`. This struct is copied into a globally available copy, which most other file system functions rely upon. Having a global `FatPartitionInfo` is necessary for the interface functions to work, without having to pass in partition info through the arguments which would be cumbersome.

A downside to this approach is that only one FAT partition can be active at a time. This was deemed acceptable however, since the board only has one card slot anyway and SD cards are rarely formatted with many partitions. If multiple partitions were to be supported, it would be necessary to implement some kind of mounting, akin to what an operating system would do.

Open files Open files are represented with the struct `FatFile` and are stored in the array `fat_open_files`. Upon initialization, all these have their `free` field set to `1` to indicate the descriptor is not taken. No other fields in the struct needs to be modified, since they are initialized when a file is opened.

Flag as initialized Before the function returns, the global `fat_initialized` is set to `1`. This indicates that the partition has been initialized and the data

in `fat_pinfo` is valid. Attempting to use any of the interface functions without this flag set, besides the partition related ones, will result in an error.

5.4.6 Path resolution

Path resolution is done by the helper function `fat_resolve_path`. It returns an error code indicating success or failure. As its arguments it takes the path to resolve, a pointer to a buffer in which to read data and a pointer `idx` to a `FatDirEntryIdx`, which will be filled with the directory entry index of the file / folder if found.

Searching is done simply by linearly searching all entries in the clusters and names of the files and folders are extracted using the helper function `fat_get_long_name_uint8`. As its arguments it accepts a pointer to a sector buffer, a pointer to a `FatDirEntryIdx` and a pointer to where the name should be stored. It returns an exit code determining whether a name, short or long, was successfully retrieved.

The name contains `uint8`, because the function retrieves the name of an entry, truncating the 16-bit UNICODE characters in the long name (if it exists) to 8-bit characters. This is done because handling UNICODE characters (like converting to upper-case for comparison) is very complicated without an operating system that provides locale support. In the case that no long name exists, the function outputs the short name. This allows using only one function call to retrieve the name of an entry, whether it is stored in a single or multiple entries, which simplifies the logic when comparing names of files and directories.

The argument `idx` must point to the value of the directory entry index for which the name is to be extracted. If the entry is a short entry, then the short name is simply extracted. If the entry is the first in a long name entry chain, a call is made to `fat_get_long_name_uint16` passing along the provided.

This function traverses the entry chain to extract the long name, while also updating the value of `idx` and loading any new sectors into the provided sector buffer. The effect of this is that upon function termination, the directory index will point to the short name entry of the file and the sector buffer will be loaded with the containing sector. Thus when other functions are forward traversing a directory, they can call this function on every entry to retrieve the name and it will update the index and sector accordingly, which avoids reading any entry twice.

An earlier implementation had the path resolution only retrieve names when encountering a short name entry, which resulted in the name retrieval functions reading many entries (and loading many sectors) that had already been inspected in the forward traversal. Figure 17 illustrates this scenario. The numbers indicate the order that the entries are traversed. First the short entry is found, then the long name is extracted and then the search continues.

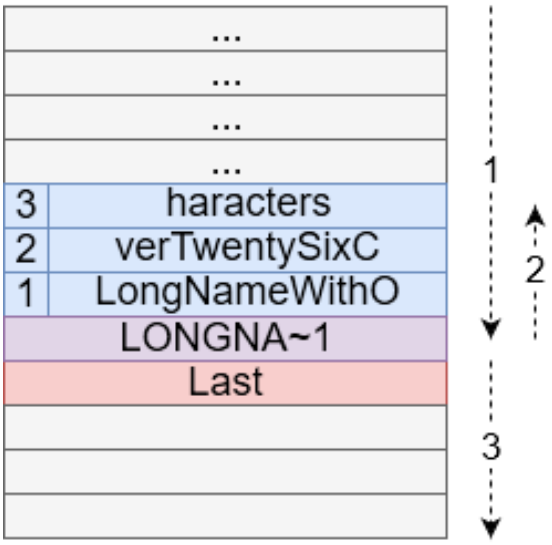

Figure 17: Illustration of the **slow** way to retrieve long names

5.4.7 Opening files

The way to open files is with the `fat_open` function. The function returns the file descriptor for the open file on success and `-1` on failure. As its arguments it takes a path to the file in question, as well as an integer `oflag` which indicates how the file should be opened. The individual bits of `oflag` dictate what should happen and are extracted using the masks defined in the standard library header file `fcntl.h`. In Table 19 is shown an overview of the different values. Note that the permission flags have no effect on function behaviour.

Mask	Meaning
`O_RDONLY`	Open file in read-only mode. Has no effect.
`O_WRONLY`	Open file in write-only mode. Has no effect.
`O_RDWR`	Open file in read/write mode. Has no effect.
`O_CREAT`	Create the file if it does not exist.
`O_EXCL | O_CREAT`	If both are set, fail if the file exists.
`O_TRUNC`	Delete all existing data in file.
`O_APPEND`	Move the cursor to the end of the file.

Table 19: Masks of `oflag`

The function first checks if the file already exists with `fat_resolve_path`. If it does not, it is created, which is explained in Section 5.4.10. If the file does exist, then the path resolution has already loaded the sector of the directory entry, as well as saved the index, which is used by the `fat_load_dir_entry`

function to actually open the file. That consists of finding an available file descriptor and initializing the `FatFile`. Any failure in this function also fails `fat_open`, which can happen if the maximum number of open files have been reached or if the path points to a directory instead of a file.

At this point the file is open. If `O_TRUNC` was set then the file contents are deleted. This is done by setting the size to zero in the `FatFile` and the directory entry (requires write to disk), as well as releasing all but the first cluster in the files cluster chain. The details of this are explained in Section 5.4.11. Lastly the cursor is moved to the end of the file, if that was requested. This is explained in detail in Section 5.4.9. If any errors occurred after the file was opened, then the file is closed again before returning.

5.4.8 Closing files

Closing an open file is done with `fat_close`, which accepts a file descriptor and returns an exit code. Besides validating that the file descriptor is valid (in range and open file), the function just frees the file descriptor by setting `free = 1`. Any writes or reads with the descriptor will now fail and all other fields are reset when the file descriptor is claimed by a new call to `fat_open`.

5.4.9 Seeking in files

Setting the cursor in a file is done with the function `fat_lseek`. It takes a file descriptor, a position `pos` and an integer `whence`, which dictates how `pos` is interpreted. The accepted values of `whence` are defined in the standard library header `unistd.h` and are shown in Table 20 along with their meaning.

Value	Meaning
`SEEK_SET`	Absolute position. Put cursor at byte `pos`.
`SEEK_CUR`	Relative position to cursor. Move cursor `pos` bytes.
`SEEK_END`	Relative position to end of file. Put cursor `pos` bytes from end of file.

Table 20: Accepted values of `whence`

The function returns the new absolute position of the cursor, which is relative to the start of the file. Both the return value and the `pos` parameter have the type `off_t` which is a signed 64-bit integer type defined in `sys/types.h`. Besides being the type in the corresponding system call, it is also necessary to have that type, at least for the `pos` parameter. Since the maximum size of a FAT32 file is `UINT_MAX = 4294967295` (defined in `limits.h`), but `pos` must be signed to allow for negative relative positions, the `pos` parameter needs more than 32 bits.

The absolute position that the cursor should end up in is first calculated. If `whence` is not a valid value or the calculated position is not within the file, the function exits with an error.

Two values in the `FatFile` needs to be updated: `pos` and `current_cluster`. After the final position has been calculated, the number of clusters that must

be traversed is calculated, which is also the amount of look-ups in the FAT. The function then traverses the cluster chain using the function `fat_get_table_value`, and sets the fields after finding the last cluster. This `fat_get_table_value` simply retrieves the value in the FAT for a given cluster. It outputs this value to a pointer passed in the arguments, which allows it to return an exit code that can indicate I/O errors.

5.4.10 Creation

Creating a file is arguably the most complicated of all the file system tasks. It is wrapped in an internal function `fat_create`, which returns an exit code. As its parameters it takes:

1. The path of the file `path` to be created
2. A pointer to a `FatDirEntryIdx` `pdir_idx`, which will be set to the index of the parent directory
3. A pointer to a sector buffer in which data will read
4. A pointer to a file descriptor, which will be set to the descriptor for the newly created and opened file
5. A flag `isfile`, which indicates whether to create a file or a folder.

The process of creating a file can be summarized as follows: Find the parent directory, find space for the directory entries, reserve a free cluster and create the directory entries.

Resolving the directory The first task is finding the cluster and directory entry index of the parent directory, as it is in there that the new directory entries must be placed. Both of these values can be found by `fat_resolve_path`, unless the file is in the root directory in which case the values are found in the global `fat_pinfo`. Before this can be done, the path of the parent directory must be separated from the file path, which is done by a linear search for path delimiters and handled by the helper function `fat_idx_of_next_path_delim`. If the resolution fails for any reason, the function exits with an error code indicating the problem.

Finding space After finding the parent directory, free space for the directory entry / entries must be found. If the file name can fit in a short name entry, then only a single short name entry is required, but otherwise long entries must be created. The number of long entries required, is calculated from the length of the name and the number of characters per long name entry.

Finding space for the entries is done by linearly searching through the entries of the directory. That is, checking all the entries, in all the sectors of all the clusters. A destination is found when a large enough streak of empty entries have been located. If the last entry of the directory is encountered, it is stored in a flag as it is then necessary to mark a new entry as ”last”, just after the

short entry. A thing to note here is that a long name entry chain can never cross a cluster boundary, so if the entry chain otherwise would be placed across a cluster boundary *and* overwrite the previous last entry, it is necessary to mark the last entries in the first cluster as free. Figure 18 illustrates this scenario.

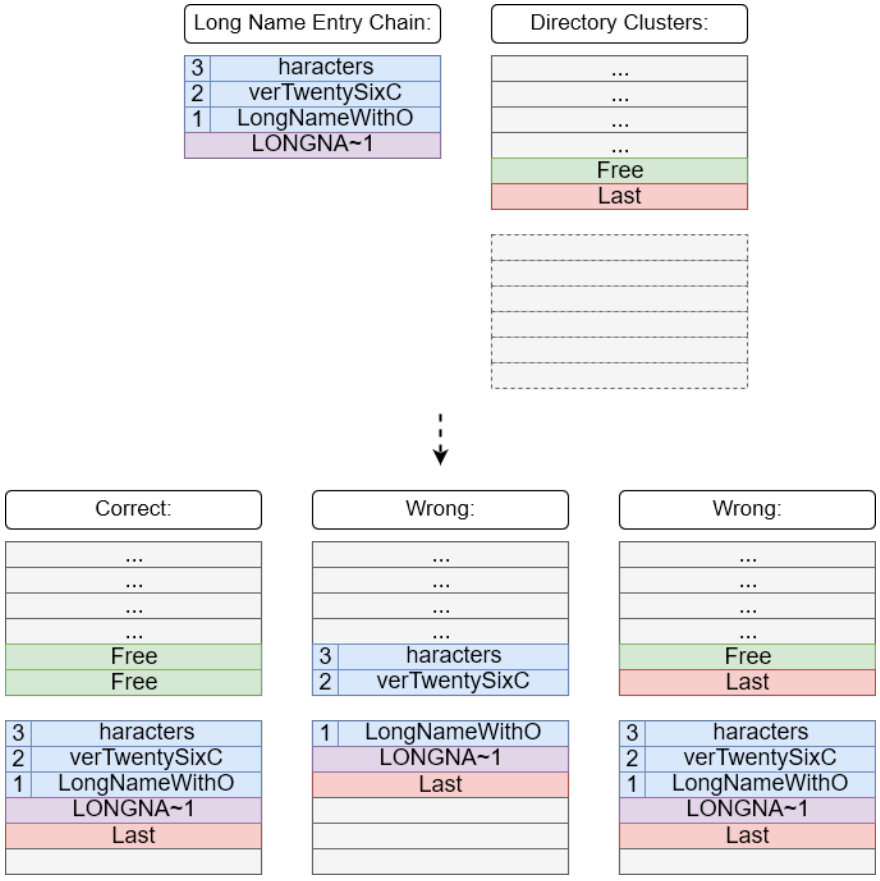

Figure 18: Illustration of how a long name entry chain must be placed in clusters

Generating the short name If long name entries are to be created for the file, then a short name must be generated to store in the short name entry. While traversing the directory in search of space, the function also figures out the number of the short name tail. This number must be such that the short name is unique in the directory and, while not required, it is preferred that it is as low as possible. One could be tempted to then simply count the number of collisions of the name without the tail and then that would be the number. However, every time the number increases to a new order of magnitude, it will take up another character of the name. With fewer characters for the non-tail part of the name, new collisions can occur.

Table 21 illustrates this problem by providing an example of a set of file names in a directory. Assume that the files have been created in the order they are listed. When the last file `LongFarewell9` is to be created, a naive implementation might shorten the name to `LONGF~10`, since there are 10 collisions on the first 6 characters, but that name is already taken. Because the tail now takes up another character, one has to account for the number of collisions on the first 5 characters. But once again, there are so many collisions that the tail must be extended and now it is the first 4 characters that matter.

Long Name	Possible short name
`LongFileName0`	`LONGFI~1`
`LongFileName1`	`LONGFI~2`
...	...
`LongFileName9`	`LONGF~10`
...	...
`LongFileName99`	`LONG~100`
`LongFarewell0`	`LONGFA~1`
...	...
`LongFarewell8`	`LONGFA~9`
`LongFarewell9`	`LONG~101`

Table 21: Illustration of name collision problem when tail expands

The solution to this problem was to count the number of collisions that happen when using any number of the name characters. This is recorded in an array `short_name_nums`, wherein `short_name_nums[i]` is the number of collisions when using `i` characters. The array is filled by traversing the characters of the short name entries. The tail number can then be found by searching from the end of the array (all characters), until a number is found which fits in the characters then available. A number fits when $n = S_i < 10^{(|S|-2)-i}$, where n is the tail number, S is the counting array, $|S| = 9$ is the length of the array and i is the index in the array. If for example using 6 out of 8 characters in the name, the number of collisions must be $n < 10 = 10^{7-6}$. Figure 19 shows an example of how such an array would look, if inserting the file "`LongFaint`" into the directory of Table 21.

Cluster reservation Reserving a new cluster in the FAT is done by the function `fat_acquire_next_cluster`. This function searches for a free cluster in the FAT and reserves it. It takes two parameters: A pointer to a cluster index `cluster` and a flag indicating whether to link the current cluster to the next. The `cluster` pointer must point to a valid cluster index and if linking is requested, it must be the index of the cluster that is linked from.

The function begins in the FAT at the entry for the `cluster` value and searches linearly forward, looping around to encompass the entire FAT. Upon finding a free cluster, it stores it at the location of the `cluster` pointer and marks it as a "last" cluster (in the FAT). If no free entries exist, then the disk

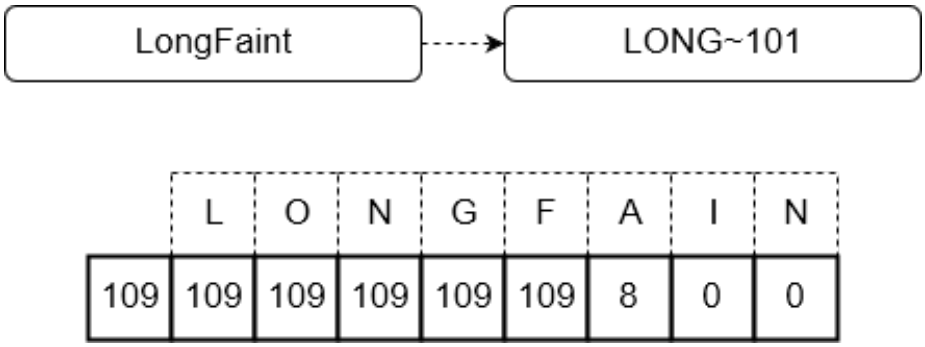

Figure 19: Example of a short name number array

is considered full and the function exits with an error. If the FAT is in a bad state, meaning an unexpected value is encountered, then it also exits with an error.

If linking is requested, then it links the old cluster to the new cluster, which amounts to extending the cluster chain by inserting the index of the new cluster in the entry of the old. If it is not, then the initial value of `cluster` simply marks the beginning of the search and nothing more.

Writing the entries Finally the entries are ready to be written to the disk. For the contents of the entries, see Section 3.7.3 and Section 3.7.4. Besides the name entries, it might also be necessary to mark the following entry as the last of the directory.

The entries are written in the order they appear on the disk, which is long name entries first, then the short name entry and then maybe the "last" entry. To minimize disk writes, the sector buffer is only written to the disk when the end of a sector is reached. After writing the last entry to the disk, the "file" creation is complete. If the creation was indeed for a file, `isfile == 1`, then the file is opened and the file descriptor stored at the location of the pointer from the arguments. If the creation was for a folder, then the cluster of the folder is stored in the `FatDirEntryIdx` from the arguments.

5.4.11 File deletion

File deletion is done with the function `fat_unlink`, which takes the path to a file as its argument and returns an exit code.

The function begins by resolving the path to the file with `fat_resolve_path`, which provides the directory index. Then it verifies that the file is not open already, by comparing said directory index to open files. If the file is already open, then the function exits with an error. It then deletes the file using the helper function `fat_delete`, which accepts the path of the file and a pointer to

sector buffer as its arguments. It returns an error code indicating whether the deletion was a success.

This function works quite similarly to the `fat_create` function, in that it finds the parent directory and then linearly searches forward in it, while keeping track of some key positions. While the creation function searches for free space and keeps track of name collisions, the deletion function searches for the target files directory entries, while keeping track of the free entries surrounding it. The basic task is to locate the target entries, long and short, delete them and then free the clusters occupied by the file. However, if the entries are the last non-empty entries in the directory, then the function should place a new "last" entry. If this is not done, the directory will never shrink in size and neither will the number of entries that must be searched during name collision checking, because that requires searching the entire directory. The correct place to put the "last" entry is illustrated in Figure 20, where "before" is on the left and "after" is on the right.

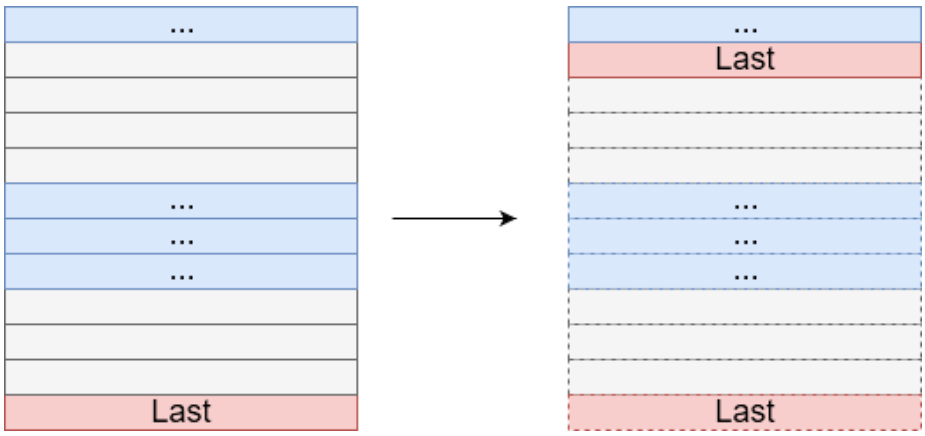

Figure 20: Illustration of where the "last" entry should be placed

The function thus searches until it finds the entries and then continues to search until either the end of the directory *or* the next non-empty entry is found (as in that case the deleted entries were not last). If the end is found, then it is sufficient to just place the "last" entry. Otherwise it reverts to the beginning of the entry chain and deletes them all.

Besides handling the directory entries, it also frees the entire cluster chain related to the file, using the `fat_free_cluster_chain`. This function takes a cluster as its argument and returns an exit code. Beginning from the provided cluster, it marks the cluster as free in the FAT and continues to the next cluster in the chain, until the end of the chain is reached. Any data already in the cluster is without importance. When the cluster is claimed by a new file, the size field will prevent any data from being read from the cluster, before said data has been overwritten. In the case that a folder claims the cluster, `mkdir`

will "empty" it as described in Section 5.4.12.

5.4.12 Folder deletion

Folder deletion is done by the function `fat_rmdir`. The function takes the path to the directory in question as a parameter and returns an exit code. It is required that the folder to be deleted is empty and if that is not the case, the function exits after setting `errno = ENOTEMPTY`. After ensuring the folder is empty, which is done with a linear search, it is deleted with `fat_delete`.

5.4.13 Reading

Reading from files is done with the function `fat_read`. As its arguments it takes a file descriptor, a pointer to where the data should be stored and then the number of bytes to read. It returns the number of bytes read.

The functions first verifies its arguments. An invalid file descriptor or a closed file and the function exits with an error. If the read would extend beyond the end of the file, the number of bytes to read is reduced so that it only reads until the end of the file. Reading is then done a sector at a time. First any odd bytes up a sector boundary is read, then whole sectors and finally the remaining bytes.

5.4.14 Writing

Writing to files is done by the function `fat_write`. Similar to `fat_read`, it returns the number of bytes written. The arguments are a file descriptor, a pointer to where the data to write is stored and lastly the number of bytes that should be written. Once again the parameters are verified before the function proceeds.

Unlike reading however, the number of bytes to write is not capped to the end of the file, as that would mean files could never expand. The function then uses the same logic of first writing up to a sector boundary, then whole sectors and finally the remaining bytes. Any sectors not completely full with new data are first read from the disk, then altered before being written back. Upon reaching the end of a cluster, a new cluster is reserved and linked with `fat_acquire_next_cluster`. After the writing is complete, the directory entry for the file is updated with the new size if it changed.

6 Results

6.1 Performance

Measuring the speed of the implementation allows it to be better understood. The actual speed of the implementation can be compared to other implementations, which can help reveal where improvements can be made.

The most fundamental area to test is the raw interaction with the SD card, since everything else is built on top of that. Afterwards, the performance of reading and writing to files in the file system is tested. This performance is what really matters to the user, as they will interact with files only. The file system module is bound to include some overhead, as besides reading and writing the raw file data, it also maintains the structure of the file system. How much overhead it introduces is important, as that percentage can be used to calculate the possible speed-up if the file system module was used with another disk / driver.

Timing is done by using the `clock` function in the standard library header `time.h`. It returns the number of clock ticks elapsed since the beginning of the program, which can be converted to seconds by dividing with the constant `CLOCKS_PER_SEC` from the same header file. Thus the clock time is sampled just before and just after the code that is timed, and afterwards the elapsed time is converted to seconds. To achieve accurate timings it is crucial that as little as possible is done between the time samples, other than what is being timed. For the most part the only extra thing done in the following tests is to maintain a counter.

All the code for testing running times can be found in the file `sdtime.c`.

6.1.1 Disk

Reading from and writing to the disk / SD card was timed with the test functions `ptest_time_disk_read` and `ptest_time_disk_write`. They read or write a given number of bytes, one sector at a time and starting from an arbitrary sector (since that does not affect the time). The measured timings are shown in Table 22 and are plotted in Figure 21.

Data size	Time to read	Time to write
1 MB	1.49 s	2.98 s
2 MB	2.29 s	5.96 s
4 MB	5.89 s	13.16 s
8 MB	11.85 s	24.78 s
16 MB	24.96 s	48.92 s
32 MB	49.47 s	101.33 s
64 MB	98.02 s	201.46 s

Table 22: Measured times of disk reading and writing

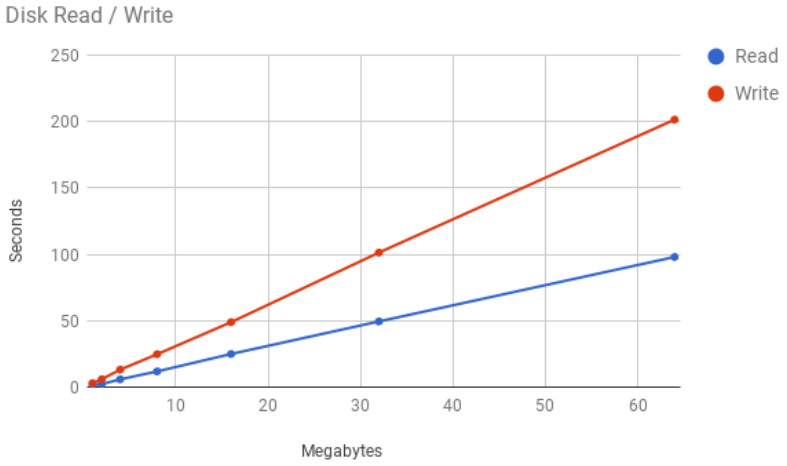

Figure 21: Plot of disk read and write times

The time spent for both operations grows linearly with the size of the data, as is to be expected. This amounts to the average speeds displayed in Figure 22.

$$V_{\text{disk\_read}} \approx 0.653\,\text{MB/s} \qquad \approx 650\,\text{kB/s}$$
$$V_{\text{disk\_write}} \approx 0.312\,\text{MB/s} \qquad \approx 300\,\text{kB/s}$$

Figure 22: Average disk speeds

6.1.2 File System

When reading and writing to files on the file system, there are many sources of overhead. These include resolving paths, reading and modifying the FAT or interacting with directory entries. Following is the test results for the file system when measuring such overhead.

Single file The first thing to test is the speed of which data can be read from and written to files. Reading was tested by the function `ptest_time_fat_read` which reads from an existing file and writing was tested by `ptest_time_fat_write` which write data to a new file. Data was transferred in whole sectors at a time,

aligned with the sector boundary. The results of this is shown in Table 23 and plotted in Figure 23.

Data size	Time to read	Time to write
1 MB	3.71 s	3.21 s
2 MB	7.34 s	12.85 s
4 MB	15.41 s	22.95 s
8 MB	30.54 s	46.50 s
16 MB	59.03 s	97.41 s
32 MB	116.91 s	190.61 s
64 MB	236.45 s	399.46 s

Table 23: Measured times of file reading and writing sector-aligned data

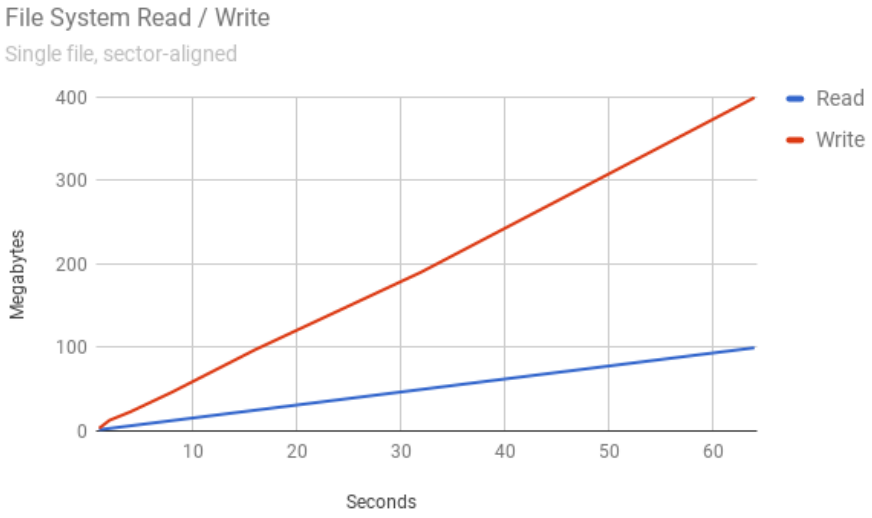

Figure 23: Plot of sector-aligned reading and writing times

Once again the time grows linearly with the data size, for both operations. This is as expected, since the overhead of finding and reserving clusters does not change with data size. This amount to the average speeds displayed in Figure 24. Note that the writing time suffers more than the reading time, which is because the size must be written to disk every time a sector is written, while reading only has to resolve the cluster chain.

These are the fastest possible speeds with the current implementation. Transferring a sector at a time ensures maximum data rate and by aligning it with the sector boundary, it is ensured that only a single read or write is required for the file contents. If the data is not aligned, this time doubles since every write

$$V_{\text{fat\_read}} \approx 0.270\,\text{MB/s} \qquad \approx 250\,\text{kB/s}$$
$$V_{\text{fat\_write}} \approx 0.160\,\text{MB/s} \qquad \approx 150\,\text{kB/s}$$

Figure 24: Average file system read / write speeds in optimal conditions

now affects two sectors (this was experimentally verified). If the data is less than a sector in size, then one should expect that the time spent is scaled up roughly by the same factor that the data size is scaled down, i.e. half as much data at a time equals double the time spent. This could be avoided by buffering the data. Note that this calculation is an approximation, since as the data size shrinks, the fraction double-cost boundary-crossing transactions decrease.

Creation A significant source of overhead when working with new files stems from file creation itself. A file here can refer to both a file and a directory, since there is no measurable difference in speed between the two. Creating a file carries a constant overhead of reserving a cluster and writing to the directory entry, but more importantly it also requires searching the entirety of the directory for any naming collisions. This means that the time spent will grow linearly with the amount of files already in the folder, even in the best case.

Resolving the path to the parent directory also inflicts an overhead. This overhead depends on where each part of the path is in their containing directory, as well as how many parts there are. In the best case (each part is the first in their directories) the overhead is small and constant and in the worst case (each part is last in their directories) it is linearly proportional to the size of the directory.

The last point of overhead comes from the length of the name. Since the file name must be compared to every file name in the parent directory, the time taken to do this comparison grows linearly with length of the shortest of the file names being compared.

The performance of creation was tested by `ptest_time_fat_create_many`, which creates a given number of files in a directory. Two runs where measured: One where the name could fit in a single short name entry and one where the names needed three long name entries. The measurements are shown in Table 24 and plotted in Figure 25.

Deletion The final operation of the file system that was measured is file deletion. The two time consuming tasks of deletion are path resolution and the freeing of the cluster chain. Path resolution gets slower as the directories in the path grow, as already discussed. The freeing of the cluster chain is a constant time operation per link in the chain, so it grows linearly with the size of the file.

Number of files	Time for short name	Time for long names
1	0.036 059 s	0.037 261 s
2	0.068 453 s	0.077 058 s
4	0.137 238 s	0.157 292 s
8	0.274 601 s	0.318 329 s
16	0.611 529 s	0.645 567 s
32	1.185 676 s	1.316 568 s
64	2.409 729 s	2.731 181 s
128	4.972 075 s	6.981 452 s
256	10.663 522 s	20.050 353 s
512	27.629 904 s	66.212 767 s
1024	73.076 722 s	239.282 583 s

Table 24: Measured times for file creation

To isolate the effects of both problems, two different deletion times were measured. The timing was done with the functions `ptest_time_deletion_single` and `ptest_time_deletion_many`, which delete one large file and multiple empty files respectively. Deletion of a single large file showcases the time growth from freeing the cluster chain, while many small files showcase the problem of path resolution. The results are shown in Table 25 and Table 26, and are plotted in Figure 26 and Figure 27.

File size	Time
1 MB	0.290 s
2 MB	0.582 s
4 MB	1.130 s
8 MB	3.028 s
16 MB	4.443 s
32 MB	11.625 s
64 MB	23.083 s
128 MB	41.365 s

Table 25: Measured times for single file deletion

6.2 Correctness

Besides the speed, another point of interest is that the implementation works. This was ensured by testing the functionality as it was developed. All correctness-testing functions can be found in the code file `sdtest.c`.

After initial development, almost all testing was directed towards the file system. This is because it is assumed that any errors in host controller or driver would manifest themselves as transaction errors, which would then appear when being used by the file system. The interface of the driver is simple enough that the file system covers all use cases.

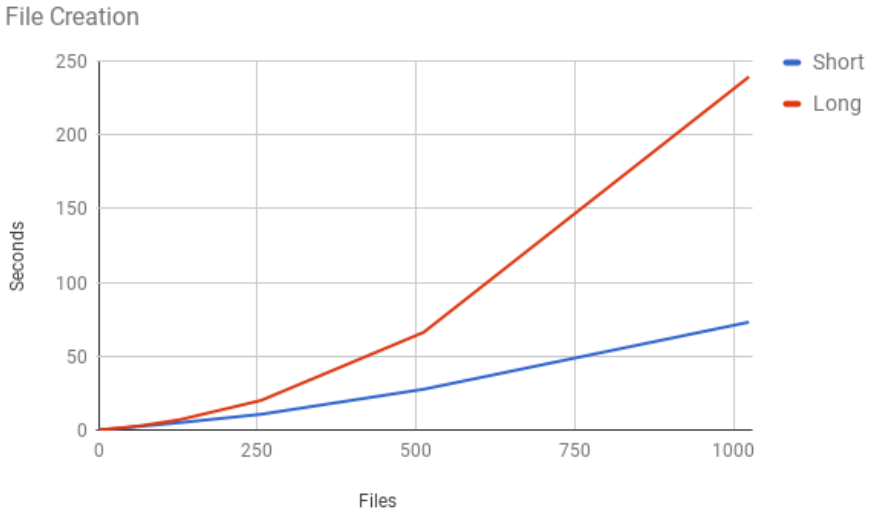

Figure 25: Plot of file creation times

Number of files	Time
1 MB	0.011 155 s
2 MB	0.047 123 s
4 MB	0.068 929 s
8 MB	0.145 311 s
16 MB	0.257 437 s
32 MB	0.403 303 s
64 MB	0.963 906 s
128 MB	2.148 345 s
256 MB	6.131 733 s
512 MB	20.921 028 s
1024 MB	71.415 339 s

Table 26: Measured times for deleting multiple empty files

6.3 Completeness

Both host controller, driver and the file system module were developed using specifications. Furthermore, the interface to the file system was modelled closely after the C standard library system calls. Therefore, to avoid any confusion, it should be clearly stated where and how the implementation differs therefrom.

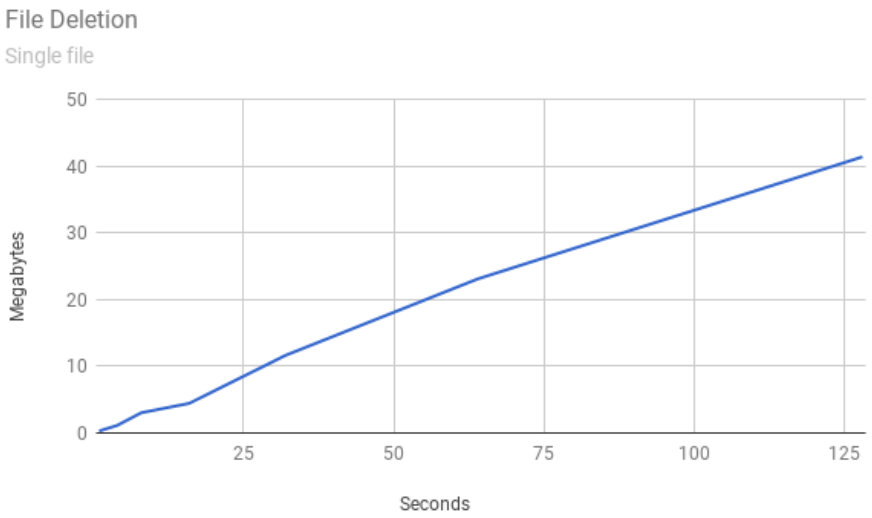

Figure 26: Plot of single file deletion times

6.3.1 SD Host Controller and Driver

The host controller supports SDSC, SDHC and SDXC cards in SPI mode only. It only allows a block length of 512 bytes but all cards support this.

CRC The host controller and driver does not support any form of CRC7 generation or verification. CRCs are disabled in SPI mode per default, so it is suspected that it is less necessary when working at the lower clock frequencies. In the process of trying to understand the algorithm however, two software implementations for CRC generation were created. One is a very compact version that only handles CRC7, found in Appendix B, and the other is a generic implementation that handles any generation polynomial, found in Appendix A. These should only serve as a reference though. If CRC was to be used for every transaction, it should be generated in hardware for better performance.

Read / Write commands The driver only supports reading and writing a single block at a time. An obvious extension would be to support the multiple-block read and write commands. How much this would improve the transaction speed is unknown, but it could be significant. It should be noted however, that any effort to optimize for speed is almost certainly better spent implementing support for the 4-bit SD mode of the SD card.

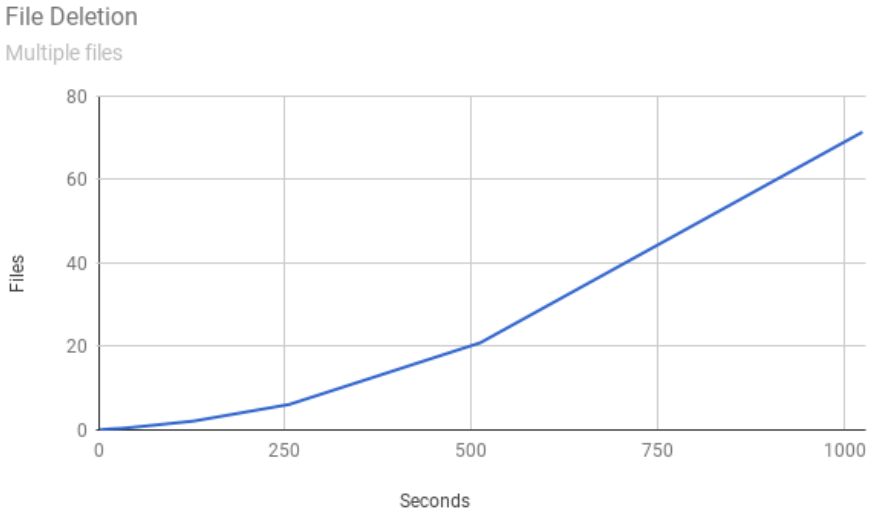

Figure 27: Plot of multiple file deletion times

6.3.2 FAT File System

UNICODE characters The FAT specification dictates that file names can contain (almost) any 16-bit UNICODE character. The implementation however does not support this. Its not possible to create new names with UNICODE characters, as the interface only accepts 8-bit character strings. If a UNICODE character is encountered in a name already on the disk, it is truncated to an ASCII value (only in memory) which could result in an invalid equality comparison being made. In almost all cases however, this would simply lead to a file with such a name being unable to be opened. Handling UNICODE is not trivial, especially not without locale support from the operating system, which is the reason it was left unsupported.

Permissions FAT files and directories have read / write permissions attached to them. The implementation completely ignores this however, freely allowing a read-protected file to be written or deleted. Implementing this is not overly complex and could probably be done in a fair amount of time, but it did not make it into this project. It was down-prioritized because it has no effect as long as the user avoids modifying protected files, which will most likely be the case the vast majority of the time.

Time Files and directories have fields that specify when the file was last accessed, when it was last read and when it was created. Since Patmos without an operating system has no concept of world time and time zones, properly setting

these was left out. Upon creation the implementation sets the field to 00:00:00 UTC, which is the date 01/01/1970 at time 00:00:00.

Invalid characters Related to UNICODE characters is the verification of the characters in a new file name. The implementation does not verify whether the provided characters are legal before creating a file. Implementing this would be fairly trivial, but did not make it in.

6.3.3 File System Interface

All of the file system interface functions set `errno = EPERM` if they are used before the file system is initialized, which none of the system calls they match do.

Seeking The system call `lseek` allows seeking beyond the end of a file. Any writes to the file will then write zero (`0x00`) bytes instead of the data bytes, until the file size catches up to the cursor. If `fat_lseek` encounters a seek beyond the file size, it sets `errno = EINVAL` to indicate invalid parameters. The described functionality was left out because it was not understood until late in the project, and it is easily replicated by the user by just writing zero bytes.

Deletion The system call `unlink` allows queued deletion of open files. If the function is called on a file that is already open, it will succeed but the file will not be deleted until the file is closed by all associated file descriptors. This implementation simply exits with an error and sets `errno = EBUSY`.

7 Future work

At the time of writing, this project exists as a public fork of the T-Crest / Patmos project on Github. The immediate next step is to get it merged upstream and integrated.

Besides getting integrated into the Patmos project, a number of improvements to the implementation could be made. First of all, implementing support for the 4-bit SD mode of the card would likely lead to speed improvements of at least an order of magnitude. It is also obvious to cover the points mentioned in 6.3, as adherence to the standards will likely benefit anyone using writing software with modules. Things behaving as one expects is rarely a bad thing. Beyond these fairly obvious points, a few select additions to the project could be interesting.

7.1 Newlib

As briefly mentioned in Section 4.5, T-Crest uses a port of the open-source Newlib[11] as its implementation of the C standard library. While this port has

[11] Newlib website: https://sourceware.org/newlib/

the stream functions `fopen`, `fclose`, `fwrite`, etc., "implemented", they just return with an error that indicates that the supporting system calls (`open`, `close`, `write`, etc.) are not implemented. By changing this implementation to use the file system module, it enables the use of the stream functions, which is the "normal" way to interact with files in C. These functions also come with the added benefit of being buffered, which would greatly increase the speed when transferring less than a sector of data at a time.

7.2 Threading

The current implementation has no guarding against race conditions, which could easily happen if more than one thread operated on files simultaneously. Securing against this would at a minimum require the file system to have mutual exclusion in the open files array, as well as implementing locking for files. At that point it would also be very beneficial to implement permissions for open files, as parallel threads could then be allowed to operate on the same file simultaneously in read-only mode, increasing overall efficiency.

7.3 WCET analysis

Patmos and the T-Crest platforms are developed such that WCET analysis is feasible. However, such analysis was not done as part of this project and it seems like a logical next step to ensure that especially the file system module is WCET-analyzable.

8 Conclusion

The aim of this project was to interface an SD card with Patmos, such that programs executed by Patmos on an FPGA with a card attached can have access to persistent storage in the form of files. This was achieved by designing a host controller for an SD card in SPI mode, writing a driver that facilitates data transactions to the card and lastly writing a file system module that can access, create and delete files, given that the card is formatted to FAT32. Both the driver and the file system provide generic interfaces, that do not expose the internals, such that any of the two could reasonably be substituted by other components, i.e. using another disk instead of an SD card. The interface to the file system module mimics system calls in the C standard library in the hopes of being familiar to C programmers, as well as to be easily integrated into the Newlib port of the T-Crest platform.

The final result achieved a maximum speed of roughly 250 kB/s when reading from files and 150 kB/s when writing to files. While this is not very impressive, it is mostly due to the low transfer speed of the SPI-based host controller. However, the generic interfaces allow the host controller be easily substituted by a future solution, that can support the high-speed transfer modes of modern SD cards.

References

[1] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 11–21, 2011.

[2] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[3] SD Specifications Part. 1: Physical layer simplified specification version 5.00. *SD Association*, 2016.

[4] Trupti D Shingare and RT Patil. SPI implementation on FPGA. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 2(2):7–9, 2013.

[5] SD Specifications Part. A2: Sd host controller simplified specification version 3.00. *SD Association*, 2011.

[6] William Wesley Peterson and Daniel T Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[8] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, Daniel Prokesch. *Patmos Reference Handbook*, 2017.

[9] Master Boot Record. `https://technet.microsoft.com/en-us/library/cc976786.aspx`. Accessed: 20/06/2017.

[10] Microsoft Corporation. Fat: General overview of on-disk format. fat: General overview of on-disk format. December 2000.

[11] Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1. Standard, International Organization for Standardization, Geneva, CH, April 1998.

[12] The Unicode Consortium. The Unicode Standard. Technical Report Version 6.0.0, Unicode Consortium, Mountain View, CA, 2011.

[13] The Open Group Base Specifications Issue 7 / IEEE Std 1003.1$^{\text{TM}}$-2008, 2016 Edition. Standard, The Open Group, Geneva, CH, 2016.

[14] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Prentice Hall Press, 2014.

[15] Terasic Inc. *Terasic DE2-115 User Manual*.

[16] Michael E. Lee. Optimization of Computer Programs in C. `http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html`. Accessed: 20/06/2017.

Appendices

A Generic CRC generation code

```
#define dat_t uint64_t

int crc(dat_t poly, int n, dat_t dat, int dat_len) {
  dat_t res = dat << n;
  dat_t div = poly << (dat_len - 1);

  dat_t lim = 1 << n;
  do {
    res = res ^ div;
    while ((res ^ div) > res) // Align first 1
      div >>= 1;
   } while (res >= lim);

  return (int)res;
}
```

B Compact CRC7 generation code

```
#define dat_t uint64_t

int crc7_compact(dat_t dat) {
  dat_t poly = (dat_t)0b10001001 << (40-1); // Shift far left
  dat <<= 7; // Shift far left
  do {
    dat ^= poly;
    while ((dat ^ poly) > dat) poly >>= 1; // Align left most 1s
  } while (dat >= 1 << 7); // Keep going until dividend is gone
  return (int)dat;
}
```

C Files of the implementation

```
patmos/
├── c/
│   ├── sdtest.c
│   ├── sdtime.c
│   └── libsd
│       ├── fat32.c
│       ├── fat32.h
│       ├── sd_spi.c
│       ├── sd_spi.h
│       ├── sddisk.c
│       └── sddisk.h
└── hardware/
    ├── config/
    │   └── altde2-115-sd.xml
    ├── quartus/
    │   └── altde2-115-sd/
    │       ├── patmos.qpf
    │       ├── patmos.qsf
    │       └── patmos.sdc
    ├── src/
    │   └── io/
    │       └── SDHostCtrl.scala
    └── vhdl/
        └── patmos_de2-115-sd.vhdl
```

D Values of `errno`

Value	Meaning
`EPERM`	File system module is not initialized.
`EIO`	I/O error occured during the operation.
`ENOENT`	The file can not be found.
`EMFILE`	Maximum number of files already open.
`EEXIST`	Trying to create file that already exists.
`EISDIR`	Path points to directory.
`ENOTDIR`	A non-final part of path is file.
`ENOSPC`	Not enough available disk space.

Table 27: Values of `errno` for `fat_open`

Value	Meaning
`EPERM`	File system module is not initialized.
`EIO`	I/O error occured during the operation.
`EINVAL`	File descriptor out of range.
`EBADF`	File descriptor does not match open file.

Table 28: Values of `errno` for `fat_close`

Value	Meaning
`EPERM`	File system module is not initialized.
`EIO`	I/O error occured during the operation.
`EINVAL`	File descriptor out of range.
`EBADF`	File descriptor does not match open file or corrupt FAT.
`ENOSPC`	Not enough available disk space.

Table 29: Values of `errno` for `fat_write`

Value	Meaning
EPERM	File system module is not initialized.
EIO	I/O error occured during the operation.
EINVAL	File descriptor out of range.
EBADF	File descriptor does not match open file or corrupt FAT.

Table 30: Values of `errno` for `fat_read`

Value	Meaning
EPERM	File system module is not initialized.
EIO	I/O error occured during the operation.
EINVAL	File descriptor out of range.
EBADF	File descriptor does not match open file.
EAGAIN	Bad cluster encountered during search.
EPIPE	Position is outside of the file.

Table 31: Values of `errno` for `fat_lseek`

Value	Meaning
EPERM	File system module is not initialized.
EIO	I/O error occured during the operation.
ENOENT	The file can not be found.
EBUSY	File to be deleted is open somewhere.
EBADF	Corrupt FAT.

Table 32: Values of `errno` for `fat_unlink`

Value	Meaning
EPERM	File system module is not initialized.
EIO	I/O error occured during the operation.
EEXIST	Name is not unique in the directory.
EBADF	Corrupt FAT.

Table 33: Values of `errno` for `fat_mkdir`

Value	Meaning
EPERM	File system module is not initialized.
EIO	I/O error occured during the operation.
ENOENT	The folder can not be found.
ENOTDIR	Path does not point to a file.
ENOTEMPTY	Folder is not empty.
EBADF	Corrupt FAT.

Table 34: Values of `errno` for `fat_rmdir`