Universiteit Leiden
Opleiding Informatica

Implementing I/O Infrastructure Improvements for S.M.A.C.K.

Name: Lars van Luik
Studentnr: 0728276

Date: 25/08/2014

1st supervisor: Prof. Dr. H.A.G. Wijshoff
2nd supervisor: Dr. K. F. D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

S.M.A.C.K. is a small and simple operating system developed at LIACS to run on the BeagleBoard. The simple design allows students to learn how operating systems work. To keep it simple only the most essential features of an operating system are implemented. One essential feature that is missing is support for storage devices. The BeagleBoard does have an SD-card slot, but there are no drivers to use it. The only way to boot is using a RAM disk that is loaded by the boot loader. This is not very convenient. It would make S.M.A.C.K. more user friendly if it was possible to access data on SD-card and directly boot from it. This thesis will describe the implementation of an SD-card driver for S.M.A.C.K. to add a support for a permanent storage device and the implementation of two methods to improve performance: DMA and a buffer cache. Several experiments were then performed to test the effectiveness of DMA and the buffer cache on the BeagleBoard. The results show that a buffer cache and DMA transfers can improve the performance in specific situations.

Contents

1 Introduction

S.M.A.C.K. is a small and simple operating system developed at LIACS. It was initially developed for the BeagleBoard [1] and has also been ported to the x86 architecture [2]. It is used in the operating systems course to teach students how operating systems work. Because of the small and simple design students can easily implement features and experiment with it. In order to keep it simple it only contains the most essential components of of a modern operating system such as a memory management system, a process scheduler and a virtual file system. One essential feature that is missing is support for storage devices. While most modern operating systems rely on a storage device like a hard drive to boot from S.M.A.C.K. did not have any support for storage devices. The only way to boot S.M.A.C.K. was from a RAM disk that was loaded by the boot loader. Once booted there was no way to access any other device which meant that all data was lost after the system was shutdown. The BeagleBoard does have a SD/MMC interface that could be used as a storage device however, S.M.A.C.K. does not have a driver that allows this to be used. The first goal of this project is to implement an SD-card driver for S.M.A.C.K. on the BeagleBoard. Together with the existing file system drivers it then becomes possible to access files on the SD-card and even boot the system with the root file system on the SD-card instead of on the RAM disk.

Disk access takes a lot of time compared to memory access [8]. Because of this most modern operating systems implement direct memory access and disk caching mechanisms to improve the performance of storage devices. Direct memory access is a technique that allows the copying of data from and to memory to be handled by a separate controller independent of the main processor. This means that the processor can do other work during the time it would normally spend on copying data. A cache is a system that keeps recently used data in memory to provide faster access when the data is needed again. If the data is in the cache there is no need to access the slow disk, instead the data can be accessed from fast memory. The next goal of this project is to implement a buffer cache and DMA support for the SD-card driver and investigate how these features will affect the performance.

The following sections of this thesis will describe the design and implementation of the SD-card driver, the DMA driver and the buffer cache. Section 2 gives an introduction on how SD-cards work, explain more about the SD controller on the BeagleBoard and the implementation of a driver. Section 3 explains about DMA, the controller on the BeagleBoard and how the driver is implemented. In Section 4 is described what a buffer cache is, some cache techniques are explained and the implementation on S.M.A.C.K. is described. And finally in Section 5 experiments to measure the performance of the implemented features will be described and the results are discussed.

2 SD-Card

The BeagleBoard is equipped with an SD-card slot. An SD-card is a type of flash memory based storage it is is an improvement of the MultiMediaCard (MMC). It is available in various sizes and capacities and is used in many small or portable devices such as cameras and phones. This section describes the SD protocol, the MMC/SD controller on the BeagleBoard and the implementation of a driver for it.

2.1 SD-Card protocol

The SD-card protocol is a serial protocol that uses a command line to send commands, data lines to transfer data and a clock for synchronisation. The protocol is formally defined in [4]. Table 1 shows the pin assignment. The protocol can be used in either 1 bit mode or 4 bit mode. Both

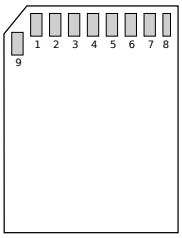

Figure 1: SD-card pin numbering

Pin	Name	Function
1	DAT3	Data line 3
2	CMD	Command line
3	VSS	Ground
4	VDD	Power
5	CLK	Clock
6	VSS	Ground
7	DAT0	Data line 0
8	DAT1	Data line 1
9	DAT2	Data line 2

Table 1: SD-card pin description

modes are almost identical, the only difference is the number of data lines that are used. In 1 bit mode data is sent over 1 data line and in 4 bit mode data is interleaved over 4 data lines.

The command line is used to send commands and receive responses. The command line uses master-slave communication. The host is the master and the card is the slave. This means that all commands are sent by the host. The card can only respond. Table 2 shows some of the commands with their index number and a short description. It is not a complete list of available commands but it contains all commands that were used for this project. There are two types of commands: normal commands which have CMD in front of the index number and application-specific commands which have ACMD in front of the index number. Application specific commands provide a way for card manufacturers to extend the available commands. Some application specific commands are reserved or required by the specification [4]. Application specific commands have the same structure as normal commands and can have the same index number. All commands are identified only by their index number. CMD55 is used to indicate that the next command will be an application-specific command.

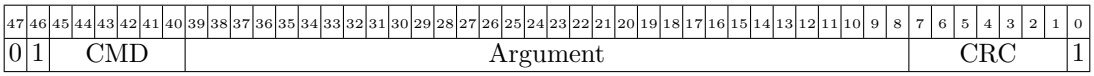

Figure 2: Command token format [4]

Commands are sent packaged in a 48 bit command token. Figure 2 shows the format of a command token. Tokens are sent over the command line with the most significant bit first and the least significant bit last. The first bit is called the start bit and is always 0, the last bit is called the end bit and is always 1. The second bit is the transmission bit, a value of 1 indicates that it is a command from the host. Next is the command index number and the argument. The format of the argument is different for each command. Commands that do not need an argument will ignore it. The CRC field contains a 7 bit CRC checksum that is used to detect transmission errors.

After each command the card will send a response. There are several different response types called R1, R2, R3, R6 and R7. The type of response depends on the command. R1 is the normal

Index	Name	Argument	Description
CMD0	GO_IDLE_STATE	None	Resets all cards to idle state
CMD2	ALL_SEND_CID	None	Asks any card to send the CID numbers on the CMD line
CMD3	SEND_RELATIVE_ADDR	None	Ask the card to publish a new relative address (RCA)
CMD7	SELECT/DESELECT_CARD	RCA	Toggles a card between the stand-by and transfer states
CMD8	SEND_IF_COND	Supply voltage	Sends SD Memory Card interface condition
CMD9	SEND_CSD	RCA	Card sends its card-specific data (CSD) on the CMD line.
CMD16	SET_BLOCKLEN	Block length	Sets the block length (in bytes) for all following block commands
CMD17	READ_SINGLE_BLOCK	Data address	Reads a block of the size selected by the SET_BLOCKLEN command
CMD18	READ_MULTIPLE_BLOCK	Data address	Continuously transfers data blocks from card to host
CMD24	WRITE_BLOCK	Data address	Writes a block of the size selected by the SET_BLOCKLEN command
CMD55	APP_CMD	RCA	Indicates to the card that the next command is an application specific command
ACMD41	SD_SEND_OP_COND	HCS	Sends host capacity support information (HCS) and asks the accessed card to send its operating condition register (OCR) content in the response.

Table 2: SD commands [4]

response, it is the response for most commands and it contains information about the card status. The other response types are only for specific commands and contain data from the card's internal registers. For example the R3 response contains the operating conditions register (OCR) and is only sent as a response to ACMD41 (SD_SEND_OP_COND).

Similar to commands, responses are sent packaged in a 48 bit token with the exception of the R2 response which is sent in a 136 bit token. Figure 3 shows the format of a 48 bit R1 response token. Just like the command token the start bit is always 0 and end bit is always 1. The second bit is the transmission bit, a value of 0 indicates that it is a response from the card. Next is the command index number of the command that the card is responding to. The card status field contains information about the current state of the card. This is used by the host to determine if the command was completed successfully or if there was an error.

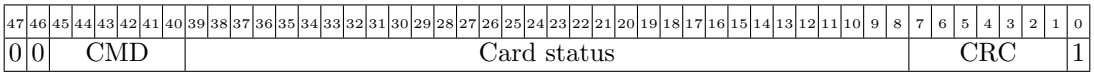

Figure 3: R1 response format [4]

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	1	OCR register																																1	1	1	1	1	1	1	1

Figure 4: R3 response format [4]

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	RCA																Card status bits																CRC							1

Figure 5: R6 response format [4]

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Voltage				Check pattern								CRC							1

Figure 6: R7 response format [4]

The SD-card works like a state machine. Depending on the current state it will behave differently and accept different commands. After receiving a command the card will usually move to another state. The state machine is divided into two modes: card identification mode and data transfer mode.

To use the card it must be initialized first. This is done in the card identification mode. The card is in this mode at power-on or after a reset. Figure 7 shows the state diagram for the identification mode. The following steps are required to initialize the card:

1. CMD0 is sent to make sure that the card is in idle state.
2. CMD8 is sent to test if the card is compatible with the host supply voltage. This command only works on v2.0 or higher SD-cards. Older cards or cards with incompatible voltage range will not respond and stay in idle mode.
3. ACMD41 is sent with as argument the supported voltage range of the host. The card responds with its operating conditions register (OCR). After this the card is in ready state. The OCR contains the voltage range supported by the card.
4. If desired the host can switch voltage using CMD11.
5. CMD2 is sent. The card responds with the card identification (CID) register. After this the card is in identification state. The CID register contains information such as the manufacturer ID, manufacturing date and serial number.
6. CMD3 is sent. In response the card sends a relative card address (RCA). It also goes into the stand-by state of data transfer mode. The RCA is needed as an argument for some commands to address the card. In case multiple cards are sharing the bus only the addressed card will respond. The RCA is always needed even when there is only one card. During the card identification mode all cards have a default RCA value of 0.

Figure 8 shows the state diagram for the data transfer mode. In the stand-by state of data transfer mode the host should first send CMD9. The card will respond with its card specific data (CSD) register. The CSD register contains information such as the data format and the card size.

From the stand-by state the host can select a card by sending CMD7 with the RCA as argument. This puts the card into transfer state. Read and write commands can only be performed while in this state. The card can also be deselected by sending CMD7 with the RCA as argument again. When the card is in the transfer state of the data transfer mode it is possible to send read or write commands. In response to a read or write command the card will send a R1 response over the CMD line. If there was no error the data will be sent over the data lines after the response.

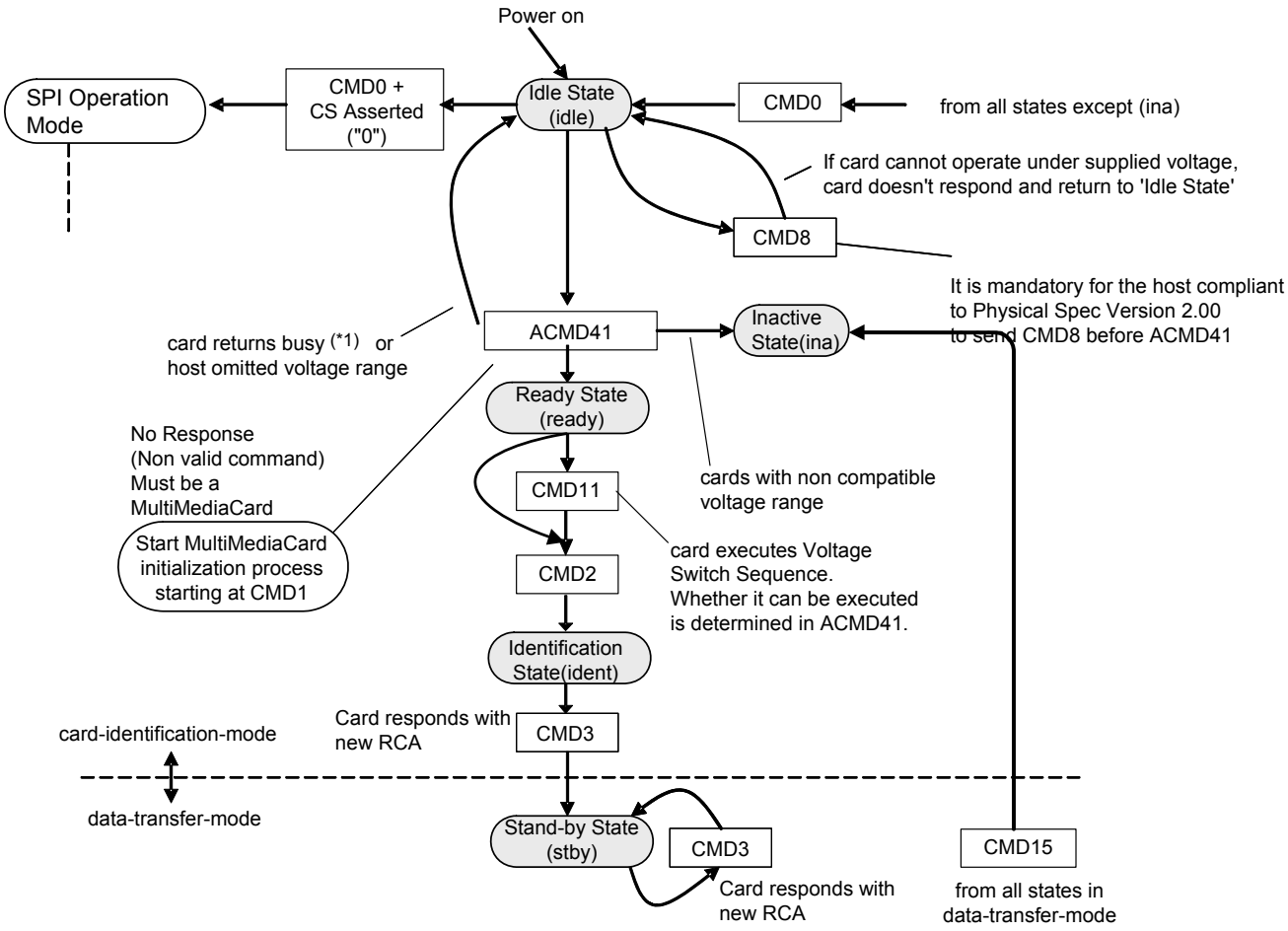

Figure 7: SD-card state diagram (card identification mode) [4]

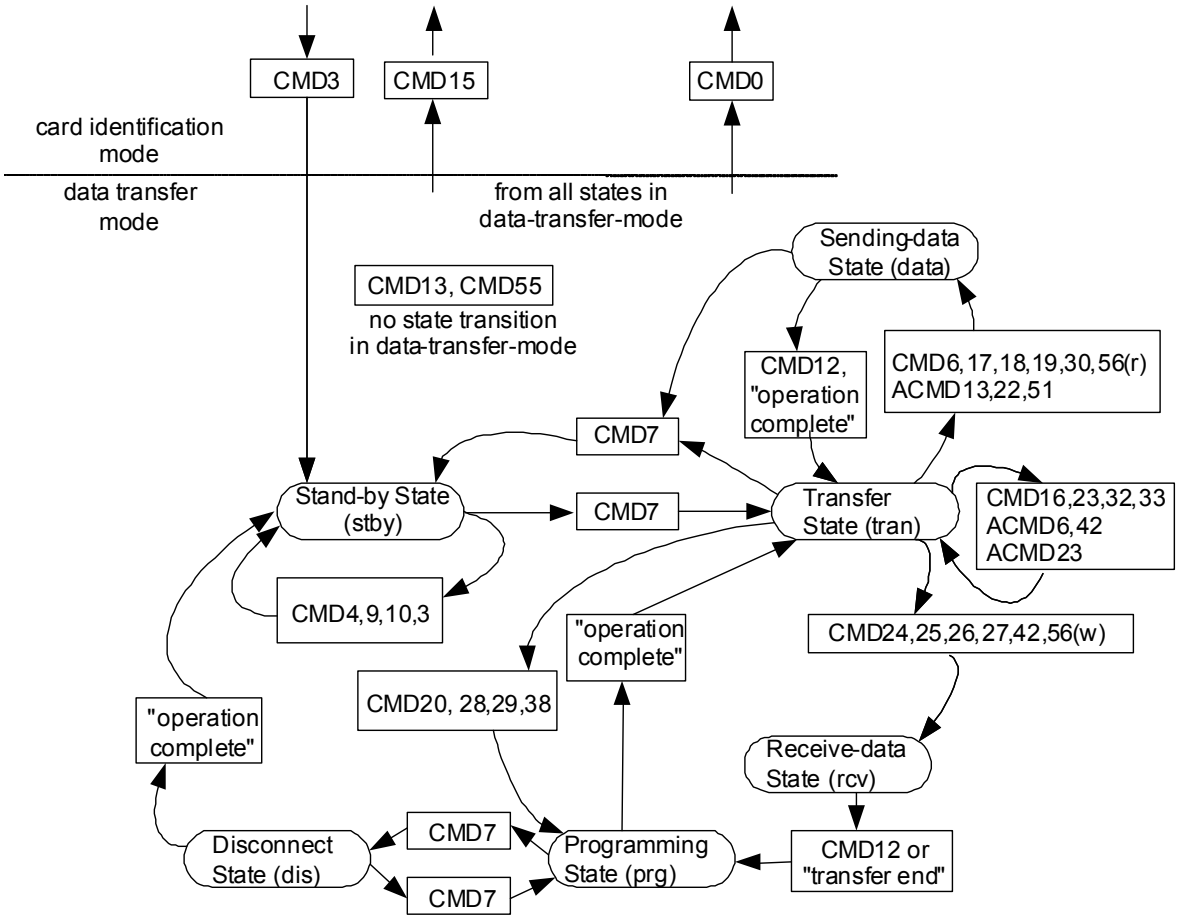

Figure 8: SD-card state diagram (data transfer mode) [4]

2.2 OMAP MMC/SD/SDIO Controller

The SD bus protocol is implemented by hardware in the MMC/SD/SDIO controller on the BeagleBoard. The MMC and SDIO functionality will not be used in this project so from now it will be referred to as SD controller. The SD controller supports both the 1 bit and 4 bit mode of the SD protocol. The controller formats the data, adds start and end bits and performs all the CRC checks [3]. It is therefore not needed to implement this in software.

Like other peripheral devices on the BeagleBoard the SD controller is programmed using memory mapped I/O. Memory mapped I/O means that the hardware can be accessed in the same way as memory because the memory and device registers are in the same address space. The SD controller can be controlled and monitored by reading from or writing to these registers.

There are registers to configure the controller and read the current status. There are also registers for various functions such as sending commands and reading responses. Sending an SD command can be done by simply writing the command to the command register and the argument to the argument register. The SD controller will then send the command to the card and place the response in the response register.

The hardware registers of the controller are specified by the manufacturer in Section 22.7 of [3]. This is a summary of the important registers that were used in this project:

BLK This register is used to set the block size and block count for a read or write operation.

ARG The argument of the command that is going to be sent should be written to this register.

CMD After a write to this register a command will be sent to the SD-card. The register has fields to set the command index and other command options such as the expected response type, the data direction for read or write operations and enabling or disabling DMA.

RSP There are 4 response registers. R2 uses all of them because it is a 136 bit response. Other responses use only the first one. The registers will contain the response data after a command. CRC checksum, start and end bits are checked by the controller hardware and are not placed in these registers.

DATA This is the register that data is written to on a write operation or read from on a read operation. The buffer read ready (BRR) or buffer write ready (BWR) bits of the STAT register should be checked before accessing this register.

STAT The status register contains fields for various errors and the status of commands and transfers. For example command complete (CC) or transfer complete (TC).

IE This register controls which status bits in the STAT register are enabled. If a status bit is disabled it will never be set.

2.3 SD-card driver implementation

The first thing that had to be done to create a driver for the SD controller was getting access to the hardware registers. The registers are at a specific memory address but because S.M.A.C.K. implements virtual memory it is not possible to directly access them. Virtual memory means that each process has its own separate virtual address space. Physical memory is then dynamically mapped into that address space where needed. These separate address spaces prevent processes from interfering with each others memory but it also prevents direct access to hardware registers. This is why the virtual memory system provides the function `vm_map_physical()`. This function maps a given physical address into the process's virtual address space and returns a virtual address through which the physical memory can be accessed. The registers for the SD controller start at the memory location 0x4809C000 [3]. Each register is 32-bit wide and the total size is 512 bytes. Using the `vm_map_physical()` function this address range is mapped into the driver's address

space. To easily access the registers a struct is created in which each register is represented by a 32-bit unsigned integer.

After it was possible to access the registers the controller had to be initialized. This is done by resetting the controller and writing default initialization values to the configuration registers to make sure that the controller is in a known state. Resetting the controller is done by writing a 1 to the SOFTRESET bit in the SYSCONFIG register. Then the SYSSTATUS register is polled. The RESETDONE bit becomes 1 when the reset is completed. After the reset is done the default settings are written to various configuration registers.

Now that the controller could be initialized the next step was to create a function to send SD commands to the card. This function is called `omap_mmc_send_cmd()`. Commands are sent by writing the command index number, response type and some other settings to the CMD register. Before a command can be sent the correct interrupt and controller configuration must be set. These settings are different for each command. Because of that a struct was made for each command that contains all the required register settings. The parameters of `omap_mmc_send_cmd()` are the command struct and command argument. When the function is called it will first wait until the SD command line is free. It then resets the STAT register by writing 1 to all fields. This is done to make sure that all status or error bits of previous commands are cleared. After that the argument is written to the ARG register and the data from the command struct is written to the corresponding configuration registers. The CMD register is written last because writing to it triggers the controller to send the command to the card. After the command is sent the STAT register is polled to check if the command is completed or if there is an error. If the command was successful, the card's response will be in a RSP register.

With the ability to send commands and receive responses it was possible to implement everything else because all functions of the SD-card are performed by simply sending one or more commands to the card (see Section 2.1).

The read function first waits until the data lines are free. Then it sends CMD17 (READ_SINGLE_BLOCK). If the command succeeds it polls the STAT register until the buffer read ready (BRR) bit is set. If the BRR bit is set the read data is available in the DATA register. When all data is copied from the DATA register it polls the STAT register to verify if the transfer is completed or if there was an error. It is important that the clock settings of the controller are configured correctly. Initially the clock settings were left untouched on the default values. This caused a lot of strange problems where SD commands would sometimes timeout or fail and the data sent from the SD card was often invalid. It took a while to realize that these problems were caused by the clock settings and a lot of time was spent on verifying all the code. After correcting the clock settings these problems went away and everything worked as intended.

The write function works very similar. It also waits until the data lines are free but then sends CMD24 (WRITE_BLOCK). If the command succeeds it polls the STAT register until the buffer write ready (BWR) bit is set. If the BWR bit is set the data can be written to the DATA register. After the data is copied it polls the STAT register to verify if the transfer is completed or if there was an error.

After the read and write functions were implemented and working correctly it only required a small modification to the boot parameters to be able to boot S.M.A.C.K. with the SD-card mounted as the root file system instead of the RAM disk.

To use a device in S.M.A.C.K. a driver for it must be registered in the device system. The device system allows all storage devices to be accessed in a standard way. To register a device it is necessary that certain functions are implemented. These functions are:

`read()` Reads data from the device.

`write()` Writes data to the device.

`seek()` Sets the position at which data will be read or written.

`open()` Called before the device is accessed.

`close()` Called when the device will not be accessed anymore.

`ioctl()` Used to access other functionality of the device besides read and write.

For the SD-card driver `open()`, `close()` and `ioctl()` have no function. When the driver is registered it will appear in the file system as the device file `/dev/sd`. There are also device files for each partition which have the same name but with the partition number appended. These device files can be used to mount the partitions on the sd card like any other device.

3 DMA

Direct memory access (DMA) is a technique used in computers to access memory independent of the main processor. It can be used to transfer data between devices and memory or between different memory locations. Because the memory is accessed by a separate controller independently of the CPU the CPU is free to do other useful work during the transfer. During a normal data transfer the CPU has to spend all its time on copying by executing load and store instructions.

The goal is to add DMA support to the SD-card driver to improve performance. Most of the tasks of the SD-card driver consist of copying data from the SD-card to memory or from memory to the SD-card. This copying is done by the CPU. If the copying could be done by the DMA controller the CPU can do other work during that time. Another way DMA can increase the performance is by using multi block transfers from the SD-card. With multi block transfers the amount of read commands that have to be sent to the SD-card is reduced and more data can be copied in a single DMA transfer. Section 5 presents experiments to evaluate the effectiveness of DMA transfers on the BeagleBoard.

3.1 OMAP DMA Controller

The BeagleBoard has a DMA controller which can copy data between memory locations. Because most hardware uses memory mapped I/O it is also possible to use the DMA controller to copy data from and to devices. The DMA controller has 32 channels. Each channel can be configured independently to do a DMA transfer. It is also possible to chain channels together to automatically perform multiple transfers.

Like the SD controller the DMA controller can be accessed using memory mapped I/O. The controller has some configuration and interrupt registers and every one of the 32 channels has its own set of channel configuration registers.

To perform a DMA transfer a channel has to be configured. This is done by writing information such as the source address, destination address and transfer size to the channel registers. If everything is configured the channel can be enabled to start the transfer. Enabling a channel can be done manually by software or automatically by a hardware event. The SD controller is connected to the DMA controller and is able to trigger DMA transfers. If a channel is set up correctly it will automatically start the transfer when the buffer of the SD controller is ready.

Every DMA transfer is a block that consists of a number of frames and each frame consists of a number of elements. The element size can be 8, 16 or 32 bits and the frame size can be anything. It is also possible to define a packet size as a number of elements. Data is always copied element by element but the channel can be configured to copy one block, one frame or one packet when the transfer is started. It is also possible to generate an interrupt after each block, frame or packet.

When an interrupt is generated it requires immediate attention. Whatever process is currently running will be interrupted and a special function called an interrupt handler is called. This function will check why the interrupt was generated and processes it accordingly.

The hardware registers of the controller are specified by the manufacturer in Section 9.7 of [3]. This is a summary of the important registers that were used in this project:

IRQENABLE This register enables or disables the interrupts for each channel. Each bit of this register represents a DMA channel. 1 enables and 0 disables interrupts for that channel.

CICR This is the channel interrupt register. It controls on what events the channel should generate an interrupt.

CCR Channel control register. This register is used to set the read and write options for a DMA channel.

CSDP This is the channel source destination parameters register. It is used to configure parameters of the data such as element size and endianness.

CEN Channel element number. This register is used to set the number of elements within a frame to transfer.

CFN Channel frame number. This register is used to set the number of frames within a block to transfer.

CSSA Channel source start address. This register is used to set the source address of the data.

CDSA Channel destination start address. This register is used to set the destination address for the data.

3.2 Implementation

The start address of the DMA controller registers is 0x48056000 [3]. Each register is 32-bit wide and the total size is 4 KB. This range is mapped into the driver's address space using `vm_map_physical()`. To easily access the registers a struct is created in which each register is represented by a 32-bit unsigned integer.

Initialization of the controller is done by writing default values to the configuration registers.

After the DMA controller was initialized a DMA channel had to be set up to read data from the SD controller's DATA register. DMA channel 2 was used because the example in the documentation also used that channel [3] but any of the 32 channels could have been used. The channel is configured to use 32-bit elements in little endian mode for both the source and destination. The source address mode is set to constant, this means that data is always read from the same address. Destination address mode is set to post increment which means that the address is incremented by 1 element after each write. The transfer will be done with 1 frame and the end of frame interrupt is enabled. The number of elements in the frame depends on the total transfer size. The packet size is set to the same size as the SD data buffer which is 512 bytes. The channel is set to be triggered by the SD controller and transfer one packet each time. The source address is set to the address of the SD DATA register and the destination address is set to the buffer of the process that requested the data.

Some modifications were needed to the SD driver to make use of DMA. Instead of reading data one block at a time with CMD17 (READ_SINGLE_BLOCK) data will be read using CMD18 (READ_MULTIPLE_BLOCK). CMD18 causes the card to keep sending blocks. The number of blocks to read is specified in the BLK register of the SD controller. Directly after sending CMD18 the process will be blocked. This means that the process will be stopped until something unblocks it. In the mean time other processes are allowed to run.

When sending CMD18 the DMA enable bit in the CMD register will be set. This causes the SD controller to signal the DMA controller when the buffer is full. Because the packet size of the DMA channel is the same as the buffer size of the SD controller it will copy all data out of the buffer. The card will then automatically fill the buffer again and the process repeats until all blocks are transferred. After all blocks are copied the DMA controller generates the end of frame interrupt.

To handle this interrupt an interrupt handler function had to be created. This function will verify that there were no transfer errors and then resumes the blocked process.

4 Buffer cache

4.1 Caching

A cache is a system that transparently stores data for faster access. Caches are used in many situations where the access time of data is relatively high. For example most CPUs have a built-in cache because accessing data on the same chip is faster than accessing external RAM. Web browsers use a cache because accessing data locally is faster than accessing it remotely over a network. Another situation where a cache is often used is for hard drives and other storage devices. Disk access is very slow compared to memory access. The difference can be many orders of magnitude [8] and because of that most operating systems use some sort of disk cache. This cache stores data that is read from disk into memory. When the data is needed again it can be accessed directly from the cache. This way the cache will improve the I/O performance by minimizing the amount of disk access.

For this project a buffer cache was implemented. A buffer cache is a type of cache that stores data in buffers with a certain block size. When data is requested the buffer cache will look up the corresponding block. If the required block is found the data will be returned from the buffer cache. This is called a hit. If the block is not found it will be read from disk and added to the buffer cache before it can be returned. This is called a miss. Misses should be avoided as much as possible because they take a lot more time. Using a cache means that there are two copies of the data. One in the cache and one on disk. When data is modified in the cache the changes must also be written to disk. This can be done either immediately when the change is made or the cache can keep track of all changes and write them at a later time.

There is usually a lot less memory available than there is disk space so it is generally not possible to cache an entire disk. At some point the cache is full and blocks need to be removed from the cache to make room for new ones. Removing blocks from the cache can increase the number of misses because there is the possibility that you remove data that will be needed again. Deciding which blocks will be removed while minimizing the number of misses is done by a cache algorithm, also called replacement algorithm. An optimal replacement algorithm will remove the data for which the next access is the furthest in the future. Because it is impossible to look into the future this algorithm can not be implemented. There are several other algorithms that are possible to implement such as the following [7, 8]:

Random replacement (RR) Randomly replace a block.

Least recently used (LRU) Replace the block whose most recent access was earliest.

First-in, first-out (FIFO) Replace the block that has been in memory longest.

Last-in, first-out (LIFO) Replace the block most recently moved to memory.

Least frequently used (LFU) Replace the block that has been accessed the least.

Some of these algorithms are only useful in certain situations. For example LIFO works best in situations where old blocks are accessed frequently. LIFO only replaces the block that is added last. All blocks that were added before the last one will remain in the cache forever and accessing any other data will cause a miss. A similar thing can happen with LFU if the cache is filled with blocks that all have been accessed multiple times. If a new block is added to the cache it will always have a lower access count than the older blocks, so it will always be the new block that is immediately replaced. For the average case LRU and FIFO are generally good algorithms [7].

Many UNIX-like operating systems provide two methods to access files. The first method is through I/O system calls like `read()` and `write()`. The other method is memory mapping with `mmap()`. Memory mapping means that the contents of a file are mapped directly into a process's address space. Most operating systems used to implement both methods separately and both had their own cache. The I/O system uses the buffer cache which allocates buffers of memory in the kernel address space. Data is read from disk and stored in these buffers. When a process requests data it is copied from the buffer to the process. The page cache is used for memory mapping. It stores file data in memory pages managed by the virtual memory system. These pages can be mapped into a process's address space. With two separate caches it can happen that data is cached twice. Once in the buffer cache and once in the page cache. To solve this duplication many operating systems now use a unified buffer cache (UBC) [5]. The purpose of a UBC is to eliminate the double caching. This is done by re-using the virtual memory pages of the page cache as buffers instead of allocating separate buffers. The same data in memory can then be used for both memory mapping and I/O calls.

Because S.M.A.C.K. does not support memory mapping of files it was decided to implement a normal buffer cache as it is a simpler design compared to a unified buffer cache. The block size will be the same as the page size so it will still use the virtual memory system to allocate pages but the pages are only used as blocks for the buffer cache. The file system drivers will need to be modified to access the cache instead of the device drivers. The buffer cache will then handle the requests from the file system drivers. On a cache miss it will request data from the device drivers. The buffer cache sits between the file system drivers and the device drivers. Every time a file system needs access to disk it has to go through the buffer cache. In the case of a read operation the buffer cache will try to find the requested disk blocks in memory. If they are found the data is copied from memory. Otherwise the blocks are read from disk and stored in memory first. In the case of a write operation the data will be written to memory. The buffer cache will then later write the data to disk. Because processes are blocked during I/O this will save time. The process can already continue after the data is written to the buffer cache and does not have to wait until it is written to disk. Delaying the writing of data to disk brings the risk of data loss because all data in RAM is gone after a crash or power down. Modified data will be periodically written do disk to prevent that. This is called a cache flush.

4.2 Implementation

Each device is identified by a major and a minor number. The major number is used by the device system to determine which driver is used to access the device. The minor number can be used by the driver for various functions. Usually it is used to indicate the partition number on the device. The device drivers only work at block level and are unaware of any file systems. The virtual file system provides file access through several file system drivers which access the device drivers. To implement a buffer cache the file system drivers were modified to access the buffer cache instead. The buffer cache then uses the device drivers if data from disk is needed.

The buffer cache has a maximum size of 16MB and uses a block size of 4096 bytes which is the same size as a virtual memory page. This makes it easy to allocate a block using the virtual memory system. The maximum number of blocks is equal to the maximum cache size divided by the

block size. For each block the cache has a block descriptor. The descriptor contains information about the data stored in the block such as the device major and minor and the status of the data. They do not contain the cached data itself, only a pointer to the memory page which contains the data.

For performance reasons it is important that block descriptors can be accessed quickly. This is why the buffer cache uses an AVL-tree for each device minor to store the block descriptors. An AVL-tree is a self-balancing binary tree for which insertion, deletion and searching all take $O(\log n)$ time. In the AVL-tree the block descriptors are ordered by block offset. With a given device, minor and offset it is possible to quickly find the corresponding block descriptor. When it is needed to loop through all the block descriptors an AVL-tree is less efficient. This is why the buffer cache also contains two linked lists. A page descriptor can be in a list and in an AVL-tree at the same time. One linked list contains all the used block descriptors and the other linked list contains all the unused block descriptors.

A read request to the buffer cache contains the device, minor, offset and size. When a request comes in the buffer cache will calculate in which cache block the requested data should be. It then uses the AVL-tree of the requested device to look up the block descriptor. If the block descriptor is found, the data can be copied from the memory page that the block descriptor was pointing to. If the block descriptor is not found it means that data must be read from disk. A block descriptor is taken from the linked list that stores all unused descriptors. Then a free memory page will be mapped and filled with data read from disk. The block descriptor will be updated with information about the newly read block and added to both the device's AVL-tree and to the end of the linked list of used pages.

For write operations all modifications are done only in the cache block, not directly on disk. If there is a write to a block that is not in the cache it will be added first. Finding or adding blocks is done in the same way as for a read operation. When data in a cache block is modified the status in the block descriptor will be set to dirty. Every 30 seconds all dirty blocks are written to disk.

When the buffer cache is full the linked list of unused block descriptors is empty. If a new block is read an old block has to be removed from the cache. This is done using the LRU (least recently used) algorithm. Every time a cache block is accessed the descriptor for that block will be moved to the end of the list that stores the used blocks. This means that the list will be sorted by last access time. When it is necessary to remove a block the first one in the list of used blocks will be picked which is the least recently used. The data of that block will be written to disk, the page that contains the data is unmapped and the descriptor is moved to the unused list so it can be re-used for the new block.

5 Experimental Evaluation

In the previous sections two ways of improving I/O performance were discussed: DMA transfers and a buffer cache. This section will describe some experiments that were conducted to investigate the influence of DMA transfers and a buffer cache on S.M.A.C.K..

When DMA is disabled every read from the SD-card is divided into 512 byte blocks which is the maximum read block size for SD-cards. For each block a separate read command must be sent to the SD-card before the data of that block can be copied. The copying of data from card to memory is done by the CPU. With DMA enabled a multi block read command will be sent to the card. This causes the card to read all the required blocks with a single command. Instead of the CPU the DMA controller will then handle the copying of data from the card to memory. It is expected that the real time performance with DMA enabled is better than the performance with DMA disabled because less commands have to be sent to the SD-card. Additionally another

performance increase may be caused by the fact that the DMA controller is copying the data and not the CPU. The CPU can do other work in the meantime. When the buffer cache is disabled all reads will be directly from the SD-card. If the buffer cache is enabled data will only be read from the SD-card if it is not available in the cache. This should improve performance by reducing the amount of reads from the SD-card.

Performance is measured in the amount of time it takes to read a file. The ARM processor on the BeagleBoard contains a built-in cycle counter [6] which was used to measure time. The CPU runs at 720 MHz which means that one cycle is $\frac{1}{720 \times 10^6}$ seconds. To prevent an overflow of the 32-bit cycle counter a divider of 64 is used, this means that only every 64th clock cycle is counted, so in the results below one cycle is $\frac{64}{720 \times 10^6}$ seconds.

The counter measures the total number of cycles, not only the cycles of the process that is used to read the test files. This means that the results could contain some overhead caused by other processes. To minimize this overhead the only other process besides the test program is a shell.

5.1 Tests

To test the performance a simple test program was created. This program reads a file from SD-card to memory and counts the CPU cycles. The normal `read()` system call in S.M.A.C.K. has a limited buffer size. Because of this the test program uses a specially made system call that will allocate a buffer the same size as the test file. This allows the file to be read with a single `vfs_read()` call. The overhead of starting the process and allocating memory is not counted. With this program the following tests were performed:

DMA disabled, buffer cache disabled This test measures the standard read performance without DMA or buffer cache. For each file it was measured how many CPU cycles it took to fully read the file from the SD-card.

DMA enabled, buffer cache disabled This test measures the performance of DMA transfers. For each file it was measured hoe many CPU cycles it took to fully read the file from SD-card. The number of cycles that the process is blocked during the DMA transfers is also measured.

DMA disabled, buffer cache enabled This test measures the performance of the buffer cache. For each file it was measured how many CPU cycles it took to fully read the file with an empty cache and then again with data still in the cache.

DMA enabled, buffer cache enabled This test measures the combined performance of the buffer cache with DMA enabled. For each file it was measured how many CPU cycles it took to fully read the file with an empty cache and then again with data still in the cache. It was also measured for how many cycles the process was blocked during the DMA transfers.

For each test the same 4 test files of 1 MB, 8 MB, 16 MB and 24 MB were used. The tests were repeated 5 times and the average was calculated for the results. In test where the buffer cache is enabled the size was set to 16 MB.

5.2 Results

Table 3 shows the number of CPU cycles it takes to read a file from the SD-card without using the buffer cache. Comparing the normal and DMA columns show that the total time increases when DMA is enabled. It was expected that DMA would be faster than normal because multi block transfers are used so less commands have to be sent to the card. After each multi block transfer the controller has to send a CMD12 (STOP_TRANSMISSION) to the card in order to stop the transfer. This is not controlled by software but the software still has to wait for this when checking

the controller status before resuming the process. It could be that this causes enough overhead to be slower than normal transfers. It is also possible that the DMA controller just can not copy data as fast as the CPU can. The blocked column shows that the process is blocked most of the time while the DMA controller is handling the transfer. During this time the CPU is free to run other processes. So even though in this case the performance of a single transfer is worse, it is likely that the overall performance will improve in a situation with multiple processes that need CPU time.

If DMA is disabled all reads are single blocks of 512 bytes. With DMA enabled all reads are multi block reads. If the buffer cache is enabled the read size is always 4 KB which is the block size of the buffer cache. When the buffer cache is disabled the read size depends on the file system cluster size which was 8 KB during the tests. When the read size is smaller, more commands have to be sent in order to read the same amount of data which causes more overhead.

File size	Normal	DMA	Blocked	System
1 MB	128×10^5	148×10^5	146×10^5	1×10^5
8 MB	1029×10^5	1178×10^5	1167×10^5	11×10^5
16 MB	2022×10^5	2356×10^5	2333×10^5	23×10^5
24 MB	3030×10^5	3534×10^5	3499×10^5	35×10^5

Table 3: CPU cycles on uncached reads.

Normal Total number of cycles it takes to read the file with DMA disabled.

DMA Total number of cycles it takes to read the file with DMA enabled.

Blocked Number of cycles the process was blocked while waiting for a DMA transfer to complete. During this time other processes are allowed to run on the CPU.

System The time the process was active which is the total number of cycles with DMA enabled minus the number of cycles that the process was blocked.

File size	Normal	DMA	Blocked	System
1 MB	502×10^5	788×10^5	412×10^5	376×10^5
8 MB	4779×10^5	6899×10^5	3282×10^5	3617×10^5
16 MB	10526×10^5	15063×10^5	6561×10^5	8503×10^5
24 MB	18665×10^5	25352×10^5	9842×10^5	15510×10^5

Table 4: CPU cycles on cold cache reads.

File size	Normal	DMA	Blocked	System
1 MB	46×10^5	47×10^5	0×10^5	47×10^5
8 MB	372×10^5	377×10^5	0×10^5	377×10^5
16 MB	18722×10^5	22797×10^5	6561×10^5	16235×10^5
24 MB	28444×10^5	34507×10^5	9842×10^5	24664×10^5

Table 5: CPU cycles on warm cache reads.

Table 4 shows the number of CPU cycles it takes to read a file from the SD-card with the buffer cache enabled when the cache is still empty (cold cache) and table 5 shows the number of CPU cycles it takes to read a file from the SD-card with the buffer cache enabled when the file has been read before (warm cache).

When the buffer cache is enabled the initial reads take more CPU cycles than when the buffer cache is disabled. This can be explained by the fact that it takes extra time to store the data into the buffer cache. Reading a file sequentially on a cold cache will cause a cache miss for every

block. On a cache miss new memory has to be allocated before reading the data from disk. This will take more time than directly reading from disk.

Cached reads are much faster. The buffer cache size is 16 MB so the 1 MB and 8 MB file can be completely stored in memory. Reading these back from a warm cache is very fast as no disk access is needed. The differences between normal transfers and DMA transfers is almost nothing because DMA transfers are only used for disk access. The results show this because the process is never blocked. However, cached reads on files bigger than the buffer cache are a lot slower. The number of blocked cycles is almost the same for a cold cache and a warm cache. This means that the DMA transfers take about the same amount of time. However, the total amount of cycles for DMA is higher. This means that there is extra overhead that is not caused by the DMA transfer. The buffer cache removes the least recently used data first when the cache is full. If a file that is larger than the buffer cache is read sequentially the beginning of the file is already removed from the buffer cache by the time the end is reached. If the file is read sequentially from the beginning again it will cause a cache miss for every block. The overhead is likely caused by the allocating and deallocating of memory on all the cache misses.

	Normal	Copy	DMA	Blocked	System
Cycles	2051×10^5	645×10^5	635×10^5	635×10^5	895
Seconds	18.23413	5.73318	5.64467	5.64459	7.95556×10^{-5}

Table 6: 16 MB raw read.

Table 6 shows the results of a 16 MB sequential raw read. A raw read means that disk blocks are read directly, ignoring any file system. When reading a file the file system can break up a read into several smaller reads depending on the file system cluster size. With a raw read this does not happen. The full 16 MB can be read with only a single CMD18 (READ_MULTIPLE_BLOCK) and CMD12 (STOP_TRANSMISSION) to the SD-card followed by a single DMA transfer. Compared to the previous tests this is much faster. The transfer with DMA enabled takes just over 5.64 seconds. Most of that time the process is blocked. The number of system cycles is very low. The only thing that the CPU has to do is to set up a single DMA transfer and then block the process until it is done. Without DMA the transfer takes over 18.23 seconds. The copy column shows how much of that time the CPU spends on copying data. The rest of the time is the overhead of sending read commands to the card. The time that the CPU spends on copying is very close to that of the DMA transfer. Apparently most of the performance increase comes from the use of the CMD18 (READ_MULTIPLE_BLOCK) command for DMA transfers instead of using multiple CMD17 (READ_SINGLE_BLOCK) commands for normal transfers.

As a speed comparison a test was done with the same SD-card on a MacBook running Mac OS X using a USB SD-card reader with the following command:

```
dd if=/dev/disk1 of=/dev/null bs=$((16*1024*1024)) count=1
```

The time it took was 4.06631 seconds which is comparable with the raw read with DMA enabled on the BeagleBoard.

6 Conclusion

The first goal of this project was to create an SD-card driver for S.M.A.C.K. and to be able to mount a partition on the SD-card and boot from it. This is now possible which makes S.M.A.C.K. easier to use. It is no longer necessary to use special tools to add data to a disk image which was loaded by the boot loader. Data can now be placed on a standard FAT16 partition on the SD-card which is supported by almost any operating system.

The second goal was to add DMA support and a buffer cache to improve the performance. On raw reads a DMA transfer makes a big difference. It is over 3 times faster than a normal transfer.

However, on a file system read DMA is slightly slower than a normal transfer. During normal use of S.M.A.C.K. only short file system reads are done so DMA will not be an improvement for read performance. Using DMA transfers does free up the CPU for most of the time during the transfer. In a multi process situation this should increase the overall performance even if an individual transfer takes slightly more time.

Using a buffer cache also affects the performance. It can be either positively or negatively. On a cache hit the performance is significantly better than without a buffer cache but on a cache miss the performance is much worse. The testing method with large might not be representative for normal use. The full operating system including all applications are under 2 MB and would easily fit in the buffer cache. As the tests have shown, reading files which fit completely in the buffer cache is very fast.

References

[1] BeagleBoard.org `http://beagleboard.org/` (Retrieved: June 6, 2014)

[2] Matthijs van Drunen, *Porting S.M.A.C.K to the x86 architecture*, 2012, Bachelor thesis, Universiteit Leiden.

[3] Texas Instruments, *OMAP35x Applications Processor Technical Reference Manual*, SPRUF98V–April 2010–Revised January 2012

[4] SD Group (Panasonic, SanDisk, Toshiba) and SD Card Association, *Physical Layer Simplified Specification Version 3.01*

[5] Chuck Silvers, *UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD*, Freenix 2000 USENIX Annual Technical Conference Paper, Pp. 285–290 of the *Proceedings*

[6] ARM Limited, *Cortex-A8 Technical Reference Manual*

[7] Sleator, Daniel D. and Tarjan, Robert E. *Amortized Efficiency of List Update and Paging Rules* Communications of the ACM, Vol.28(2) (February 1985), pp.202-208 [Peer Reviewed Journal]

[8] Ramakrishna Karedla and J. Spencer Love and Bradley G. Wherry, *Caching Strategies to Improve Disk System Performance* IEEE Computer, vol. 27, no. 3, pp. 38-46, 1994