UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering,
Mathematics & Computer Science

The effect of writing and transmitting SD card data on the consistency of SD card write performance

Robbert Willem Kräwinkel

B.Sc. Thesis
June 2020

Supervisors:
prof. dr. ir. L. Abelmann
ir. M. Welleweerd
ir. E. Molenkamp

Robotics and Mechatronics Group

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

In this paper, we study the effect of writing data to an SD card over SPI and transmitting that wirelessly on the consistency in write performance. Our prime objectives are to determine if the performance while writing to an SD card can be predicted on both throughput and latency and what the effect is of running an HTTP SD web server in parallel to it using the ESP32 microcontroller dual core functionality. We demonstrate that there are peaks in SD card write time of at minimum $50\,\mathrm{ms}/100\mathrm{B}$ that are unpredictable in both height and occurrence and can increase write time by 1500%, it is also demonstrated how the general throughput can be increased through a variety of tests. Finally, it is shown that an HTTP SD web server running in parallel hurts write performance by 12% and how its wireless transfer speed can be maximized when a file write is happening simultaneously. These results can be directly applied to the engineering of a performance-optimized SD-based storage solution and an HTTP SD web server that can run separately or concurrently.

Contents

Chapter 1

Introduction

The MagOD2 measurement system is a magnetic optical density meter dedicated to the analysis of magneto-tactic bacteria. The system uses an ESP32 microcontroller with built-in WiFi connection and an SPI controlled micro SD card. Currently, measurement data is saved to the SD card, measurements are done in sections called recipes and are saved to the SD card as a file per recipe. To ease the operation of the MagOD2 system it would be ideal if the SD card would be accessible over WiFi. The ESP32 has a secondary processing core which could, in theory, take care of this task. However, the WiFi capabilities should not interfere with the measurements done, meaning the system should perform reliably and consistently. It is known that in general, flash storage has some caveats regarding consistent performance. However, the effect in this particular scenario was unknown. Therefore further research was required into the consistency of writing data to the SD card itself and the effect that running code in parallel on a secondary core has on the primary core its measurement task. Resulting in the question; what is the effect of writing data to an SD card and transmitting that wirelessly on the consistency in write performance? This reports discusses the measurement of performance of the SD card in various scenarios to attempt to identify, and isolate the largest inconsistencies in write performance of the SD card. Also, the effect of running parallel tasks on the ESP32 is quantified in both a bare calculation and more representative web server scenario.

1.1 Requirements

When designing an implementation for the WiFi accessible SD card for the MagOD2 measurement system there are design constraints to keep in mind. The most relevant options are listed in the MoSCoW diagram in table 1.1.

Table 1.1
MoSCoW diagram showing the importance of different requirements for a SD web server running on the ESP32 microcontroller

Must	Should	Could	Won't
Read measurement data over WiFi	Transmit data per recipe	Transmit live data	Work with MagOD1
Be able to read data from the SD card	Transmit data per day	Transmit data via a direct WiFi connection	Replace the functionality of the display
Work with current MagOD2	Not lose data due to transmission	Have a GUI	
Not affect the measurements in any way	Transmit data via network router (no WiFi direct)		
Perform consistently and reliably	Have a commandline like interface		
Be opensource	Have sufficient range to transmit the data wirelessly		

1.1.1 Datasize

In the current implementation of the MagOD2 system, the measurement data is saved locally on an SD card. The data for each measurement is saved as one or multiple `.csv` files(s), with each file containing one entry or line per measurement done. Every measurement consists of 13 different data points, or columns in the `.csv` file. Each data point is saved with a certain accuracy, depicted by the number of decimals. On average a 1000 line file is $82\,\mathrm{kB}$ in size, meaning that there are on average, $82\,\mathrm{B}$ used per line. That includes the data, all the required commas to separate the values and the characters for specifying the end of the line and starting the next (`CR, \r`, carriage return and `LF, \n`, line feed). In order to allow expansion of the resolution or the number of data types in the future. It will be assumed from now on that the data size of one measurement is saved as $100\,\mathrm{B}$.

1.1.2 Datarate

A collection of measurements with its settings is known as a recipe, takes about $2\,\mathrm{min}$ to execute. Measurements can be done at frequencies between $8\,\mathrm{Hz}$ and $860\,\mathrm{Hz}$ (as specified by the recipe), every measurement results in one line of data in the final data file. The size of a single line is $100\,\mathrm{B}$ so the wireless data rate should at minimum be equal to or higher than $86\,\mathrm{kB\,s^{-1}}$ when transmitting simultaneously to the measurements.

When transferring after an entire day, it is assumed that the $8\,\mathrm{Hz}$ measurement rate is used, that totals to about $7 \cdot 10^5$ measurements per day. If we wish to transfer that information within $10\,\mathrm{min}$ that would require a datarate of $117\,\mathrm{kB\,s^{-1}}$ (again assuming $100\,\mathrm{B}$ per measurement).

Combined this means that if the achieved data rate is above $117\,\mathrm{kB\,s^{-1}}$ all the required functionality could be fulfilled. According to the documentation of the ESP32 microcontroller, the maximum attainable speed when using the TCP protocol is $20\,\mathrm{Mbit\,s^{-1}}$ which equals $20/8 = 2.5\,\mathrm{MB\,s^{-1}} = 2500\,\mathrm{kB\,s^{-1}}$ [1]. Meaning this requirement should be attainable judging by the Wi-Fi connection.

Chapter 2

Theory

2.1 Datasize

The data that is saved to the SD card is not the same as what is measured or stored temporarily in the RAM of the microcontroller after the measurement. The microcontroller can save everything in a multitude of different data formats. The most important formats are listed in Table 2.1. The 13 data types that are saved on the SD card are also represented by these data formats. Specifically, the used data formats for the different data types can be found in Table 2.2.

The data can be sent in two different formats; as it is currently saved on the SD card, or as the raw data formats in which the data is stored. Both have advantages and disadvantages that are discussed below. But first, the differences between the two will be highlighted.

As an example, π is used as the value that is to be stored. Saving it in a `double` dataformat ($8\,\mathrm{B}$, $64\,\mathrm{bit}$), it can be represented as

$$\begin{aligned}\pi(\mathrm{double}) &= b\,01000000\,00001001\,00100001\,11111011\,01010100\,01000100\,00101101\,00011000\\ &= 0\mathrm{x}\,400921\mathrm{FB}54442\mathrm{D}18 = 3.141592653589793115997963468544.\end{aligned}$$

This is the most accurate representation of π using the `double` format that is possible. Saving this double to the SD card with let's say 7 digits would result in 3.141592 being saved.

Table 2.1
Datasizes of various dataformats on the ESP32

Type	Size (bytes)
`Char`	1
`Int`	4
`Short`	2
`Long`	4
`Float`	4
`Double`	8

Table 2.2
Data as it is stored on the SD card. In order to decrease the total table size the shorthand nx *is used to indicate that* n *variables with the same type are saved consecutively*

Name	Time	$6\mathrm{x}\ \mathrm{V}$	Temperature	OD	LED_type	Loop	$3\mathrm{x}\ \mathrm{V}_{\mathrm{fb}}$	**Sum**
Format	`unsigned long`	`6x double`	`double`	`double`	`int`	`int`	`3x double`	
Bytes	4	$6 \cdot 8 = 48$	8	8	4	4	$3 \cdot 8 = 24$	100

Numbers in text form are represented through the use of the ASCII table, in this table, each character is represented by a single byte. This byte can tell a computer which character to display and allows humans to more easily interpret data. After all, a binary string or a set of hexadecimal characters are not very easy to read. To save 3.141592 to the SD card in a readable format (through the ASCII table) requires $8\,\mathrm{B}$, 7 for the digits and 1 for the decimal point. The `double` data format also uses $8\,\mathrm{B}$ but has a much higher resolution of 30 digits compared to the 7 that are being saved now. However, if the binary string was written to the SD card without conversion through the ASCII table the following string would appear once the file is opened: `@[HT]!ûTD-[CAN]` (both `[HT]` and `[CAN]` are what are known as control characters, these are not actual characters). This happens because the computer will assume the file to be in the ASCII format and represent the bytes in such a way.

SD formatted data

The way the data is currently stored on the SD card is readable by simply opening the `.csv` files in any spreadsheet editor like Excel. This means that it is easy to interact with. However, by representing the values as readable text, either resolution is lost, or the file size increases significantly. An increase in file size would require a higher data rate to have the same amount of information transferred in the same amount of time.

Raw dataformats

By transferring the data that is located in ram on the microcontroller in a raw, non-readable format. The data can be represented in a more efficient manner, minimizing data size, while maximizing the resolution. The resolution that is transferred might even be higher than that of the sensors that are used, however in any case, no resolution is lost.

Chosen format

As the extra resolution that could be provided by the raw data formats was not required for the use of the MagOD2 system, the preference was given to ease of use that is brought by saving the data in readable CSV files.

Table 2.3
HTTP commands that the SDWebServer can receive by default

HTTP request	URL	Result
`GET`	`http://[address]/list ?dir=/data`	Returns the content in the folder specified by the dir argument
`GET`	`http://[address]/data /file1.txt`	Retrieves the file specified by the URI
`PUT`	`http://[address]/edit ?dir=/dataNew`	Creates a new folder or empty file with the name and location specified by the argument
`DELETE`	`http://[address]/edit ?dir=/data/file1.txt`	Deletes the file or folder specified by the argument
`POST`	`http://[address]/edit ?dir=/data/file2.txt`	Uploads a file specified in the body of the HTTP command (not listed here) to the file specified by the argument, creating or overwriting it if necessary

2.2 SDWebServer

In the example sketches that come with the ESP32 libraries for the Arduino IDE, there is an example called `SDWebServer.ino`. This is a simple HTTP server that can interface with a connected SD card to allow the contents of the SD card to be accessible through the WiFi connection. As this is a very good starting point for what is, in essence, a large portion of the original design assignment, the workings of this example will be explored to be able to grasp what is going on and why choices are made.

In the basis a simple HTTP server is used, an HTTP server can receive different types of requests; GET, PUT, POST and DELETE for example. Each of these types can be used while requesting a certain URI with any arguments added to it. The URI is anything after the initial web address or IP equivalent. So in the case that the entered URL is: `192.168.1.1/list`, the `/list` portion is what is known as the URI. Arguments can be added to the URL by appending a question mark followed by the argument, for example, `192.168.1.1/list?dir=/files`. The combination of these three options; request type, URI and arguments can be used to request a variety of things from the HTTP server. In this example, the URI is used to set a function and either the URI or the arguments are used to set which folder or file they should act upon. Finally, the type of request determines what is done to the specified file specifically. Some examples are listed in table 2.3.

A downside to this approach is the fact that only the HTTP GET commands can be executed by entering the URL in the address bar of a web browser, for the other types of HTTP requests another way to transmit data is required. This can be done through for example a simple MATLAB or Python sketch. In order to simplify operation, as many commands should be HTTP GET commands to allow basically full operation through the address bar only. To simplify even further, the URI should always be used to specify the location. This could both be the location that should be read from or the location in which something should be created. Everything else can be handled using arguments to specify the operation that is desired on the specified location. The only operation which is not possible this way is uploading a file, an HTTP POST request is required for this in which a file is attached to the request.

Uploading and downloading files to and from the SD card using the HTTP web server are done in a similar fashion. In particular, the reading from the SD card will be elaborated on. When the HTTP server receives a request of the correct format, in this case for a file download, it asks to stream the file:

```
server.streamFile(dataFile, dataType)
```

`server` is a variable of the `WebServer` type. Looking into the respective header file the `streamFile` function can be found:

```
template<typename T>
size_t streamFile(T &file, const String& contentType) {
    _streamFileCore(file.size(), file.name(), contentType);
    return _currentClient.write(file);
    }
```

At the very bottom, it returns the value from the write function, what is returned is actually the number of bytes that have been transmitted. `_currentClient` is of the type `WiFiClient` where the write function can be found in the respective header file:

```
size_t WiFiClient::write(Stream &stream)
{
    uint8_t * buf = (uint8_t *)malloc(1360);
    if(!buf){
        return 0;
    }
    size_t toRead = 0, toWrite = 0, written = 0;
    size_t available = stream.available();
    while(available){
        toRead = (available > 1360) ? 1360 : available;
        toWrite = stream.readBytes(buf, toRead);
        written += write(buf, toWrite);
        available = stream.available();
    }
    free(buf);
    return written;
}
```

Here $1360\,\mathrm{B}$ are preallocated, while the file is still available, or in other words, while the end of the file has not been reached yet. It reads either $1360\,\mathrm{B}$ or else how many of are left in the file from the SD card. Next, it writes the buffer to the HTTP stream. This way the entire file is sent in $1360\,\mathrm{B}$ long sections. Why $1360\,\mathrm{B}$ was chosen and if it is the optimal size for the fastest file transfer has not been determined.

File uploading to the SD card is handled in mostly the same way, with obviously the HTTP stream buffer being the input and the SD card being the output. In this case, it writes in blocks of $1436\,\mathrm{B}$.

2.3 Parallel processing

The ESP32 microcontroller has two separate cores and runs a modified version of the FreeRTOS operating system known as ESP-IDF FreeRTOS [2]. CPU 0 or PRO_CPU handles the WiFi, Bluetooth and peripherals like SPI, I2C and the ADC, while CPU 1 or APP_CPU can be used for application code [3]. Both cores share the same memory meaning that they can perform tasks interchangeably. The ESP32 can be programmed using the Arduino IDE, in which case by default only APP_CPU is used for executing code in the main loop and setup of the code. They are executed by what is known as the LOOPTASK, a task that is created by default through use of the Arduino IDE and is pinned to run on APP_CPU. Tasks can also be created manually, they allow a user to have the microcontroller execute multiple tasks on each processor core. Doing so it is also possible to assign a task to PRO_CPU essentially creating the possibility to run two tasks mostly independently.

Hypothetically it would make sense if tasks run on PRO_CPU would run slower than on APP_CPU assuming any of the features listed above are used in this or any other task. It would also make sense that when tasks are run on both cores at the same time both tasks would have a performance penalty as they are using the same memory.

When running a task pinned to a core, the CPU needs time to tailor to other tasks running on that core as well, one of these tasks is the IDLE task. If the IDLE task is not run within the set timeout period it will cause a watchdog timeout rest, causing the ESP32 to reboot. This is not an issue when running code in the main loop as it has the watchdog timer disabled by default when using the Arduino IDE. The watchdog timer is in place to make sure that one task is not holding up the entire system which may have more than one task running. To circumvent this there are multiple options, the function `vTaskDelay();` suspends the currently running task for a set amount of time. In this time other tasks such as the IDLE task can be run, preventing a reset. Another option is changing the priority of the tasks, the IDLE task has a priority of 0, the lowest possible. By also setting your task to have a priority of 0 the CPU can alternate between those two tasks. Which will inevitably slow down the custom task. There is also the `yield();` function, which will suspend the current task and allow a higher priority task to run. This will not work for the IDLE task as that will always have the lowest priority and therefore never be higher. In any case, by keeping the watchdog active, some performance impact will always be measurable by having to tailor to the IDLE task. In return, the watchdog will guarantee that the code is not stuck in a loop causing the program to halt.

2.4 SD transferspeed

SD cards are in essence flash chips with an integrated microcontroller managing the filesystem, the interface with the outside world and access to the flash chip. SD cards can be formatted using multiple filesystems; NTFS, exFAT and FAT32 to name a few on the Windows operating system. As FAT32 is the default for SD cards that are equal or smaller than $32\,\mathrm{GB}$, it will be assumed as the used filesystem for now. When formatting an SD card there is an option to change the 'allocation unit size' or 'cluster size', these refer to the sizes of memory blocks on the SD card. When reading or writing to a FAT32 system it is accessed per cluster, a single file can span just a single cluster or be spread over multiple clusters if the file is too big to fit in a single cluster.

SD interfaces

SD cards can be accessed in two different modes, the default SD mode and the alternate SPI mode. While the SPI mode is much easier to implement it does also have its downsides. For example;

"The disadvantage is the loss of performance of the SPI mode versus SD mode (e.g. Single data line and hardware CS signal per card)." [4]

and

"[In case of SDHC and SDXC cards in SPI mode] *partial block read/write operations are also disabled."* [4].

Both are referring to a reduction in performance when using the SPI mode. This can partially be explained by looking at the pinout of the two modes. Where in SPI mode the SD card has one dedicated pin for data input and one for output (which are never used simultaneously), SD Mode allows up to 4 pins for both input and output in parallel. Which if everything else is kept the same would allow a factor 4 higher throughput in comparison to SPI mode. Moreover, the speed of the SPI bus is determined by the clock speed that the used microcontroller, in this case, the ESP32, can reach on the SPI clock.

According to the SD card association Class 4 and 10 SD cards can reach $4\,\mathrm{MB\,s^{-1}}$ and $10\,\mathrm{MB\,s^{-1}}$ respectively [5]. However, from the SD card specifications: *"As opposed to SD mode, the card cannot guarantee its Speed Class. In SPI mode, host shall treat the card as Class 0 no matter what Class is indicated in SD Status."* [4]. Meaning that even though the classes of the different SD cards indicate the attainable performance in SD mode, this performance can not be guaranteed when the SD cards are being accessed over SPI.

From the SD card specifications it can be read that in SPI mode;

"In the case of SDHC and SDXC cards, block length is fixed to 512 bytes" [4]

In this case block length is the size of data which can be read in one operation.

By default when setting up the SD card using SPI the SPI bus speed is set to $4\,\mathrm{MHz}$, in the initialization a higher frequency can be given. The ESP32 will, in that case, use the closest lower frequency possible. It is limited by the onboard crystal and divisions of that frequency meaning that not simply all frequencies are possible. [6]

Cluster and buffer size

The buffer size is the size of the buffer variable on the microcontroller used to store the results of a read operation. If this is smaller, more read cycles are required to read a file with set size. The allocation size is the size of one memory block on the SD card, files can occupy just one block or span multiple depending on the file size.

Flash chips

Flash consists of planes, blocks, and pages (from large to small). Write and read operations happen on a page level but erase happens on a block level. Erasing also takes an order of magnitude more time than a read or write. Erasing also hurts the lifetime of the flash chip as it has a limited amount of erases and writes. This is why if a set of data in a page is to be changed, its newer version is simply written to another page and the old page is marked as invalid to be eventually cleaned up. [7]

The management of the flash chip for distributed writing and clean up is known as the flash translation layer (FTL). In short, the FTL can be seen as having the following tasks: [8]

1. Write updated information to a new empty page and then divert all subsequent read requests to its new address (logical block addressing)
2. Ensure that newly-programmed pages are evenly distributed across all of the available flash so that it wears evenly (wear levelling)
3. Keep a list of all the old invalid pages so that at some point, later on, they can all be recycled ready for reuse (garbage collection)

Logical block addressing (LBA) is used to make the change in the location of data on pages, it is invisible to the user. If the user requests dataset A, LBA will make sure to access the correct page on which this dataset is stored, which could be a different location than before without the user knowing.

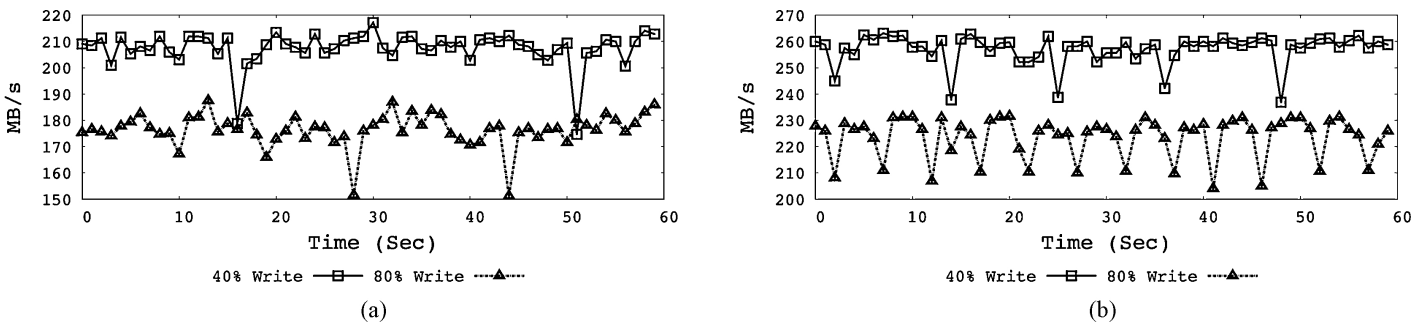
Wear levelling is the act of writing data to different blocks to spread out the load over the entire flash chip. The issue is with blocks that contain data that rarely changes. To make sure the chip wears evenly these should be moved such that their block can also be erased and reused. But moving data adds writes and with it more wear. More aggressive wear levelling thus causes more wear and has a higher performance impact, less aggressive wear levelling causes hot and cold spots. The balance between these two extremes is key to a good wear levelling scheme, it is up to the manufacturer of the flash controller to implement this.

Through the use of garbage collection, blocks containing invalid pages are erased. The non-invalid pages need to be moved to a different block before erasing as otherwise data is lost. There needs to be a balance between how many pages need to be relocated and how many are invalid. Moving pages requires writes, taking time and causing wear.

In addition, each flash die (or plane) can only perform one operation at a time. So while erasing (which takes a long time) no data can be read or written to the same plane. Operations will be queued. For multi-level cell (MLC) the read time is $\approx 50\,\mu\mathrm{s}$ and the erase time is $\approx 3\,\mathrm{ms}$. So read operations could be slowed by 60x randomly if an erase is taking place in between. Background garbage collection (BGC) happens without effect on I/O operations from the user, resulting in predictable I/O times. If more data is being changed than BGC can keep up with user I/O operations will have to wait for the background tasks to complete. This is known as active garbage collection (AGC) and is bad for latency and performance. Once the buffer zone of overprovisioned space is used completely the FTL will switch to AGC until enough buffer room is cleared again and BGC can kick in again. [9]

Figure 2.1

Write speed of a commercial MLC ***(a)*** *and SLC* ***(b)*** *SSD at two different write loads using* $512\,\mathrm{kB}$ *files. Both SSDs exhibit a pattern of dips in write speed whose frequency increases as the load on the drive increases. [11]*



However, even if no erases occur between reads or writes;

"Some cards may require long and unpredictable times to write a block of data" [4]

as

"The responsiveness of flash memory cells typically changes over time as a function of the number of times the cells are erased and re-programmed." [10].

In conclusion, a flash chip has a lot of tasks happening in the background which are not visible for the user. Performance is dependent on both foreground tasks and background tasks meaning one can not simply be ignored when comparing performance.

Looking at the write performance of commercial single-layer cell (SLC) and MLC type SSDs in figure 2.1, the effect of the FTL becomes obvious. Rhythmic dips in write speed indicate the background processes that are running and have to run more frequently if data is written to the drives faster.

Worst case

In order to approximate the maximum time a write of a set amount of bytes would take, it is assumed that for every byte written a block erase has to take place. If $100\,\mathrm{B}$ are written and a block erase takes $\approx 3\,\mathrm{ms}$, this would take $300\,\mathrm{ms}$ for just the erases to happen. In this case, it is assumed that all of the pages were invalid and none of them have to be moved before the erase. Without knowing the number of pages within a block it is impossible to calculate what the actual worst-case would be. Also, the write time is assumed to be negligible in comparison to the erase time.

Chapter 3

Methods

3.1 Test setup

All tests were done on the same setup consisting of a breadboard mounted ESP32 with an SPI connected micro SD cardholder. The setup can be seen in figure 3.1. The connections between the ESP32 and the SD card are described in table 3.1. Everything is powered over the USB connector on the ESP32, the $5\,\mathrm{V}$ from the USB connector is stepped down to $3.3\,\mathrm{V}$ using an onboard regulator, this same $3.3\,\mathrm{V}$ line powers both the ESP32 and the SD card.

Figure 3.1

Test setup showing the ESP32 microcontroller and micro SD card holder attached and interconnected on a breadboard

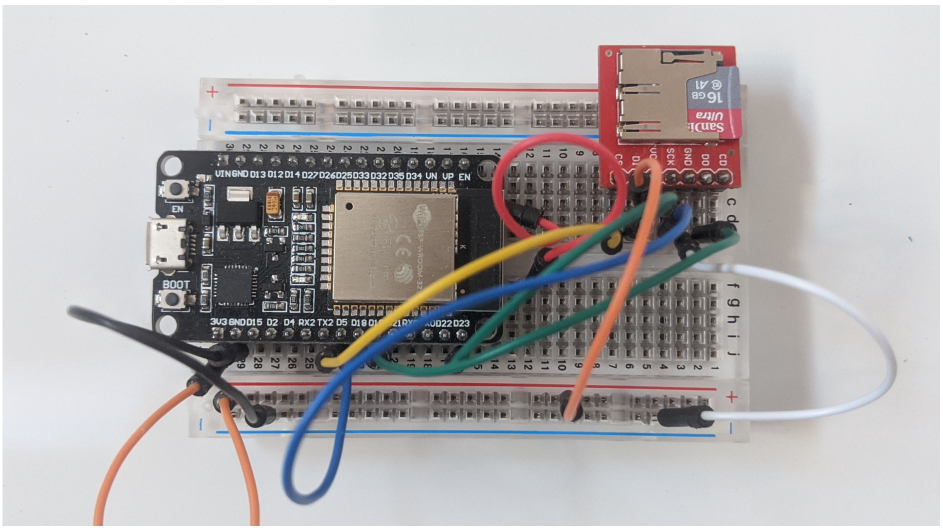


Table 3.1
Description of the connections between the ESP32 microcontroller and the SD card

ESP32	SD card
3v3	Vcc
GND	GND
D5 (CS)	DAT3 (CS)
D18 (SCK)	CLK (SCK)
D19 (MISO)	DAT0 (DO)
D23 (MOSI)	CMD (DI)

3.2 Parallel performance (impact)

To investigate the effect of using both cores a test script was designed that can do simple calculations not using any of the special features mapped to PRO_CPU. The execution time can be compared between both cores and both cores simultaneously. Separately a different test file was created to benchmark the performance of SD and WiFi access on both cores. The effect of using the SD card on both cores at the same time will also be investigated.

3.2.1 Basic performance

By doing a repeatable calculation the performance of both cores can easily be evaluated. The code that was ultimately used can be found below:

```
float t1;
int t2;
int t3;

void artificialLoad () {
  for (long i = 0; i < 1000; i++) {
    for (long  j = 1; j < 1000; j++) {
      t1 = 5000.0 * i;
      t2 = 150 * 1234 * i;
      t3 = j % 554 ;
    }
  }
}
```

It performs fairly simple operations with type conversions but does so 1 million times. The time it took to execute this function was timed using the `millis()` function, this returns the number of milliseconds passed since the start of the program, by comparing this number before and after the execution time can be deducted. The function `xPortGetCoreID()`

returns the core on which the code is running, using this it was possible to determine on which core the calculation was actually run. And finally `xTaskCreatePinnedToCore` could be used to create a new task separate from the LOOP task that was pinned to a certain core, in this case, that would be CPU 0 or PRO_CPU.

3.2.2 Application specific performance

Next in order to stimulate the access of the SD card and the WiFi communication, the `SDWebServer.ino` sketch that was discussed in section 2.2 was taken as a basis. As a repeatable test a $1\,\mathrm{MB}$ file stored on the SD card was downloaded through MATLAB 10 times and the transfertime per file as reported by MATLAB was saved to compare. As a baseline test the bare `SDWebServer.ino` sketch was used. For the measurements where any of the tasks was pinned to a core a `vTaskDelay();` of $5\,\mathrm{ms}$ was inserted to prevent the watchdog timeout discussed in section 2.3. Next a simulation of the MagOD2 system was designed, the key functionality that could have an effect on the performance of the WiFi transfer is the periodic writing to the SD card of the measurement data. With a frequency of $8\,\mathrm{Hz}$ a file on the SD card is opened, a single line of measurement data is added before the file is closed again. In the simulation this single line is $100\,\mathrm{B}$ in size and built up in the following way:

Recorded write time	$2\,\mathrm{B}$
5 constant characters written 16 times	$80\,\mathrm{B}$
16 commas to separate the values	$16\,\mathrm{B}$
2 characters to indicate the end of the line (`\r\n`)	$2\,\mathrm{B}$
	$100\,\mathrm{B}$

1000 lines are written in a file before the next file is created, this repeated indefinitely. On boot up, all previous files are deleted and files are created from scratch to make sure that comparisons are fair. In the MagOD2 system the file on the SD card is opened and closed on every cycle when writing measurement data. The advantage of doing this is that if a power outage or error occurs, the newest measurement is always saved properly. If power fails while a file is still open, the data that has been written to that file without closing is lost. The expectation is, however, that opening and closing the file every time will have a significant effect on the transfer speed of the HTTP server running on the other core. So as a test this will be compared to only opening and closing a file once the 1000 line limit has been reached. Performance on both cores will also be compared.

3.3 SD transfer speed

To be able to investigate the SD performance properly, different to be tested parameters have to be determined based on the hypothesis from the currently known facts. The expectation is that the performance has something to do with:

- ESP32 operating system background tasks
- Block size of the SD card
- File size
  - Bytes per line
  - Lines per file
- Amount of files in a folder
- Time between writes
- How bytes are written (open and close file or keep it open)
- Wear levelling
- Garbage collection

In order to be able to test all these aspects criteria for a test were determined:

- Be able to vary settings for the write test
  - Bytes per line
  - Lines per file
  - Files per folder
  - Time between writes
  - Open and close file for every write or keep open per file
- Be able to measure the power draw of the SD card
- Be able to easily test multiple scenarios to speed up testing

3.3.1 SD consistency function

In order to accurately and reproducibly test these parameters, a single test function was designed. In the basis, the script creates a variable of a set size that will contain the data that will be written to the SD card with every write cycle. This variable is filled with the same character on every test to prevent possible CRC functions from having an effect. Also, a temporary folder is created to which the temporary data will be written. Next (if the to be written file is to be opened and closed on every write), the current time is recorded with the `micros();` function which returns the time since bootup of the microcontroller in microseconds. The file is opened, the variable is written to the file and the file is closed again. Next, the current time is saved again using the same function. Both the start and end time are now saved to a separate variable and the next cycle starts. This is repeated for the set amount of times that the data should be written to the file. This can be repeated over for a set amount of files in the same folder or, by calling the consistency function multiple times, in multiple folders too. Once all the test writes have been completed the data write times that were recorded are saved to a different file and folder on the SD card together with all the settings for the test. By saving them separately the test data that is written is always consistent.

The complete test function can be found in Appendix A.

3.3.2 Energy consumption measurement

In order to measure the energy consumption of the SD card, the Vcc line was cut up and a $1\,\Omega$ resistor was placed in between. The resistance was verified using a Fluke 289 multimeter with a maximum error of 0.05% or $50\,\mathrm{m}\Omega$. Next, the voltage over this resistor was measured using two voltage probes ($V1$ and $V2$) and a Siglent SDS1102x Oscilloscope. The scope samples the voltages at $f_s = 10\,\mathrm{kHz}$ with a resolution of $8\,\mathrm{bit}$ on a $1.2\,\mathrm{V}$ range (offset by $-3.3\,\mathrm{V}$), meaning the maximum error in a voltage measurement is $4.7\,\mathrm{mV}$.

The current draw is calculated using $I = \frac{U}{R} = \frac{V1-V2}{1\,\Omega}$ next the power draw is calculated using $P = U \cdot I = V2 \cdot (V1 - V2)$ and finally the energy consumption is calculated using $E = P \cdot \Delta t$, $\Delta t = \frac{1}{f_s} = 0.1\,\mathrm{ms}$. In order to visualise the peaks in energy usage better the energy consumption was summed over $10\,\mathrm{ms}$.

Chapter 4

Results

For the figures in the report the measurement points are indicated by dots and dotted lines are used to interconnect these dots. The lines are there to guide the reader and make it easier to see relations, they do not represent the (in this case linear) behaviour between the points.

4.1 Parallel processing

By doing a repeatable calculation and varying the core to execute the calculation on the parallel performance can be evaluated. These results can be seen in table 4.1.

Table 4.1
Results of a test comparing the processing speed of the two cores of the ESP32 by running the same calculation in all scenarios. Taking the loop as a baseline performance, Core 1 performs identically and Core 0 actually performs better. Running the calculation on both cores at the same time comes with a 15% penalty. Running the same calculation twice on the same core simply doubles the execution time.

Code location	**Execution location (core)**	**Execution time (ms)**	
		Core 0	Core 1
Loop	1		823
Core 1	1		823
Core 0	0	820	
Core 0,1	0,1	968	968
2x Core 0	0	1640	
2x Core 1	1		1638

Raw computation takes 0.36% less time on core 0 (PRO_CPU) than it does on core 1 (APP_CPU) this can be assigned to the fact that the flash and ram do not need to be shared between the two cores. As core 0 also has the background tasks of the peripherals running it is the only core accessing the memory. The computational time is 18% higher if the same calculation is run on both cores at the same time, in comparison to just running the same calculation twice on the same core, which can also be pointed towards the fact that both cores share the same memory.

Knowing the performance impact of running code on two cores simultaneously, the performance on a real-world example was investigated next. Comparing the performance of the SD web server discussed in section 2.2 running across both cores and adding a consistent SD card write simulating the MagOD2 on the other core. The results can be found in table 4.2.

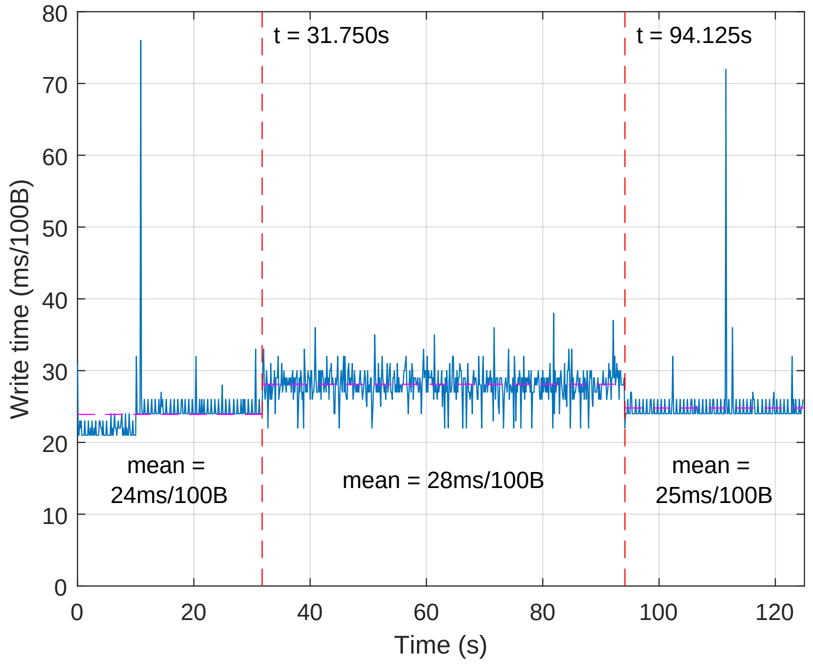
Looking at the performance of the HTTP file server, the download speed is decreased by 17.3% by moving the HTTP server from core 1 to core 0. It is known that tasks such as WiFi and SPI are handled on core 0, so the performance decrease by also moving the handling of the server to core 0 is expected. If the file that is being written to is left open, the effect on the download speed is only 4.1%, compared to 13.4% when the file is opened and closed every time.

Table 4.2
Test comparing the download speed of 10, $1\,\mathrm{MB}$ files in multiple scenarios. Once again taking the main loop as the baseline, the performance stays fairly consistent with the server running on Core 1. Moving it to core 0 has a significant performance penalty. Writing to the SD card simultaneously on core 1 hurts performance even more if the file is opened and closed on every write. If the file is kept open, the parallel writing has an almost negligible performance impact on the transfer speed.

Test	**Details**	**Download speed ($\mathrm{kB\,s^{-1}}$)**		
		Avg	Min	Max
1	Baseline test, SD card HTTP server running in main loop	232.7 ± 7	175.5	247.2
2	SD card HTTP server running on core 1	229.2 ± 6	183.0	246.0
3	SD card HTTP server running on core 0	189.4 ± 4	155.3	202.9
4	$8\,\mathrm{Hz}$ file write on core 1 opening and closing the file on every iteration, SD card HTTP server on core 0	164.0 ± 3	134.4	172.2
5	$8\,\mathrm{Hz}$ file write on core 1 but keeping the file open, SD card HTTP server on core 0	181.6 ± 5	155.5	196.5

Figure 4.1

Write time of 1000, $100\,\mathrm{B}$ lines to a file. The area marked between the dashed red lines is where a file download is occurring simultaneously. Judging by the write time the transfer took $62.375\,\mathrm{s}$. Outside of the window where the transfer was happening there are two random spikes in write time higher than during the transfer and over $70\,\mathrm{ms}$ in height. This and the other apparent events happening around the transfer will be discussed in section 4.2. The average write time (indicated by the dashed magenta lines) in the section where the transfer is happening is $3\,\mathrm{ms}$ higher compared to the steady-state after the transfer.



SD card performance

As was mentioned before, the write time for the $100\,\mathrm{B}$ is saved as part of those $100\,\mathrm{B}$, or actually, the previous write time is saved. This was done to be able to analyse the effect on the write time if files were downloaded simultaneously. The write time of one 1000 line file can be seen in figure 4.1.

Immediately obvious is the inconsistency and increase in write time during the file transfer. Judging from the write time the transfer took $62.375\,\mathrm{s}$, which is close to the $64.02\,\mathrm{s}$ that were measured by MATLAB to complete the file transfer. What is much more interesting however are the strange peaks in write time to over $70\,\mathrm{ms}$. This will be further elaborated on in section 4.2. The average write time is 12% higher during the file transfer compared to the steady-state after the transfer.

4.2 SD transferspeed

Unless mentioned otherwise in the caption of the figure the $\mathrm{Buffersize} = 512\,\mathrm{B}$ *and the* $\mathrm{Allocationsize} = 8192\,\mathrm{B}$ *for all measurements.*

Firstly different types of SD cards were compared to get a baseline measurement, both the allocation unit size and buffer size were kept constant throughout the tests. The results can be seen in figure 4.2.

When operating in SPI mode SD cards are unable to guarantee their specified transfer speeds, meaning even lower class 4 cards can outperform class 10 cards by as much as 6.8%.

Figure 4.2
Read speed of various SD cards at different file sizes. $\mathrm{Allocationsize} = 4096\,\mathrm{B}$. *There is a obvious dip in read speed independent of the used SD card at a file size of* $5\,kB$*, the class 4 SD card also consistently outperforms the class 10 card.*

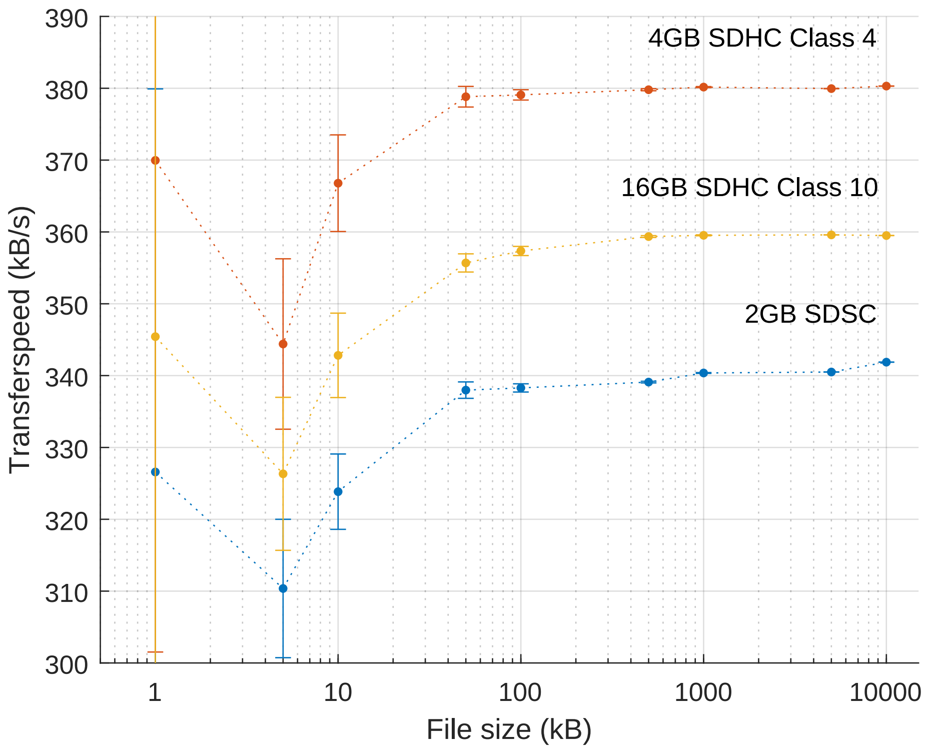
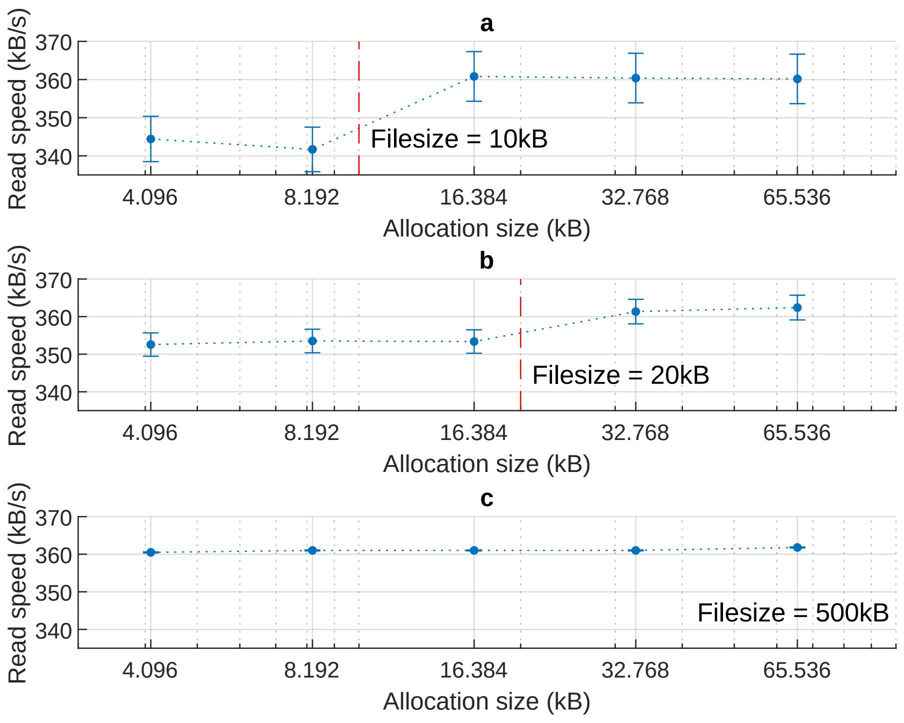


Figure 4.3

Read speed of the $16\,\mathrm{GB}$ Class 10 SDHC card versus the allocation size at various file sizes. ***(a,b)*** *Read speed decreases if the chosen allocation size is lower than the file size if the file comparable in size to the chosen allocation size.* ***(c)*** *If the file is much larger than the allocation size the impact becomes negligible.*

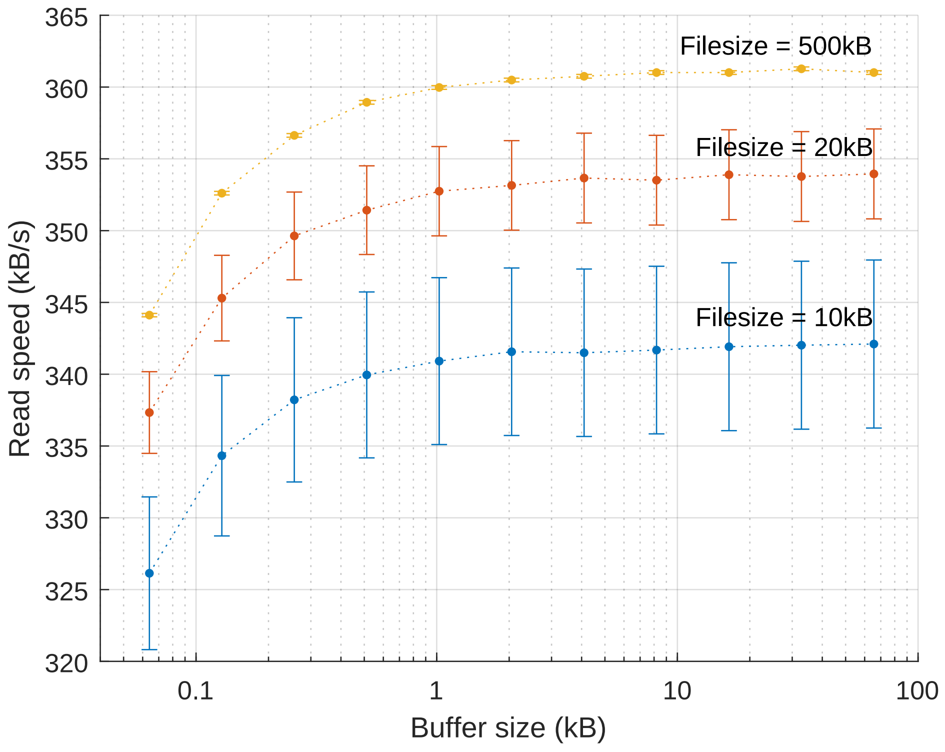


Looking into the dip in read speed at a file size of $5\,\mathrm{kB}$ was done by analysing the effect of the allocation size. At various allocation unit sizes for the different files sizes the effect becomes obvious, see figure 4.3

The formatted allocation size of the SD card can hurt the read speed by as much as 5.6% if the filesize is chosen very close to the allocation size. This is due to the fact that only entire sections of the size set by the allocation size can be read. Meaning that if the file is just slightly larger than one of those sections, two sections have to be read. If the file is much larger than the allocation size, many sections have to be read anyway. Meaning the impact on performance becomes negligible.

Figure 4.4

Read speed of the $16\,\mathrm{GB}$ *Class 10 SDHC card versus buffer size for various file sizes. The buffer size has an effect on read speed, with performance plateauing from around* $512\,B$*. Read speed is also higher for larger file sizes.*

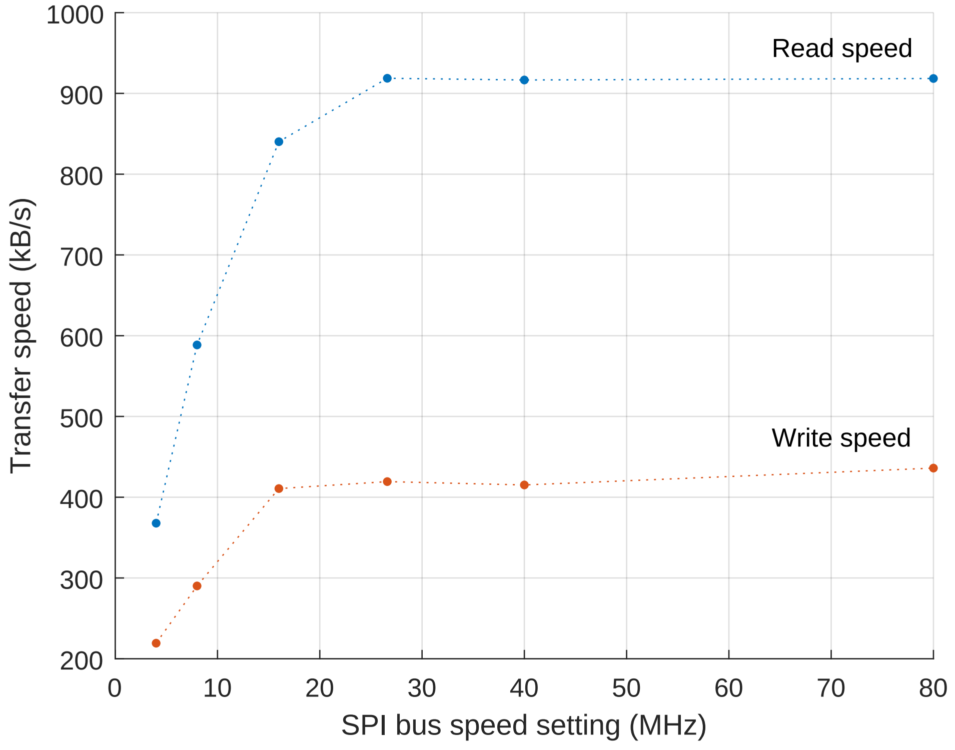


In figure 4.4 the effect of the buffer size can be seen for three different file sizes. The file size also has a significant effect it seems, with the constant allocation unit size of $8192\,\mathrm{B}$ at least.

Looking at the size of the buffer on the ESP32 side when initiating a file read it can be observed that from $\mathrm{Buffer\ size} = 512\,\mathrm{B}$ and higher the read speed plateaus. Knowing that over SPI at maximum $512\,\mathrm{B}$ can be transmitted per transfer. It makes sense that a lower buffer size would require extra read operations and therefore have a lower transfer speed. And that a larger buffer size would be very comparable to just emptying the buffer more often as more transfers are necessary to fill the buffer in that case. Increasing the buffer higher than $512\,\mathrm{B}$ has at maximum a 0.7% increase in performance, most likely due to the fact that the buffer has to be emptied less often.

Figure 4.5

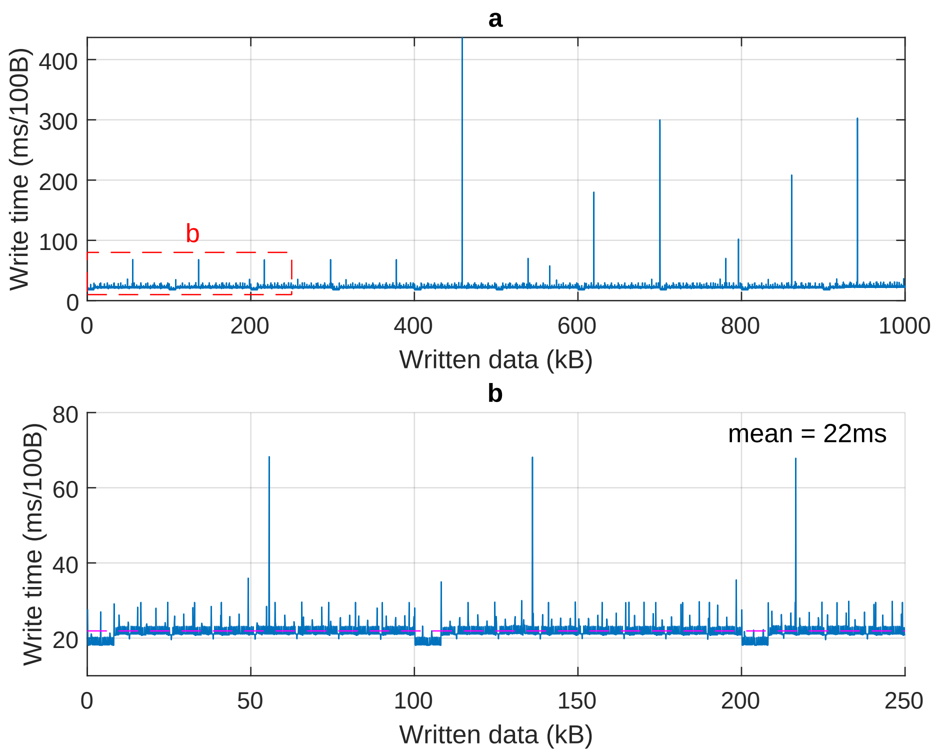
Read speed of the $16\,\mathrm{GB}$ *Class 10 SDHC card at various SPI bus frequency settings*
$\mathrm{Filesize} = 100\,\mathrm{kB}$*,* $\mathrm{Allocationsize} = 4096\,\mathrm{B}$*. The higher the SPI bus frequency setting the higher the attained read speed up to a limit at* $26.6\,\mathrm{MHz}$*.*
Note: These are the frequency settings not the actual measured frequencies.



The effect of the SPI bus speed on the transfer speed can be seen in figure 4.5.

The frequency of the SPI bus has a large effect on the transfer speed, from the default $4\,\mathrm{MHz}$ the read speed can be improved by 150% by increasing the frequency to $26.6\,\mathrm{MHz}$.

Figure 4.6
Write time of 1000, $100\,\mathrm{B}$ lines to 10 files stored in the same folder, closing the file between writes, versus the amount of written bytes. ***(a)*** *Peaks higher than $100\,\mathrm{ms}$ occur seemingly randomly but on mostly the same interval as the $70\,\mathrm{ms}$ peaks.* ***(b)*** *Write time is lower for the first few bytes of a new file, peaks of around $70\,\mathrm{ms}$ are repetitive with about $80\,\mathrm{kB}$. Average write time is $22\,\mathrm{ms}$ (dashed magenta line)*

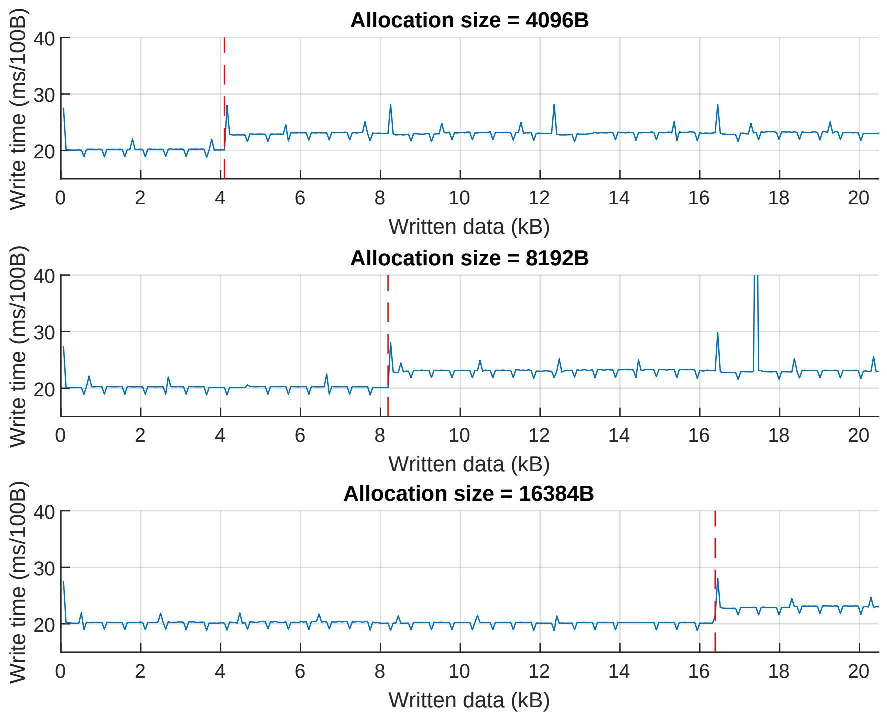


Using the SD consistency function described in section 3.3, a first test was done that mimics the MagOD2 system. Writing 1000, $100\,\mathrm{B}$ lines to files stored in the same folder every $100\,\mathrm{ms}$, the file was closed between every write. The results for this test can be found in Figure 4.6.

When writing continuously in sets of $100\,\mathrm{B}$, peaks in write time of around $70\,\mathrm{ms}$ and higher occur mostly on a $80\,\mathrm{kB}$ pattern.

Figure 4.7

Write time of $2048, 64\,\mathrm{B}$ *lines to a single file versus the amount of written bytes. Write time is lower until the total file size becomes larger than the set allocation size.*

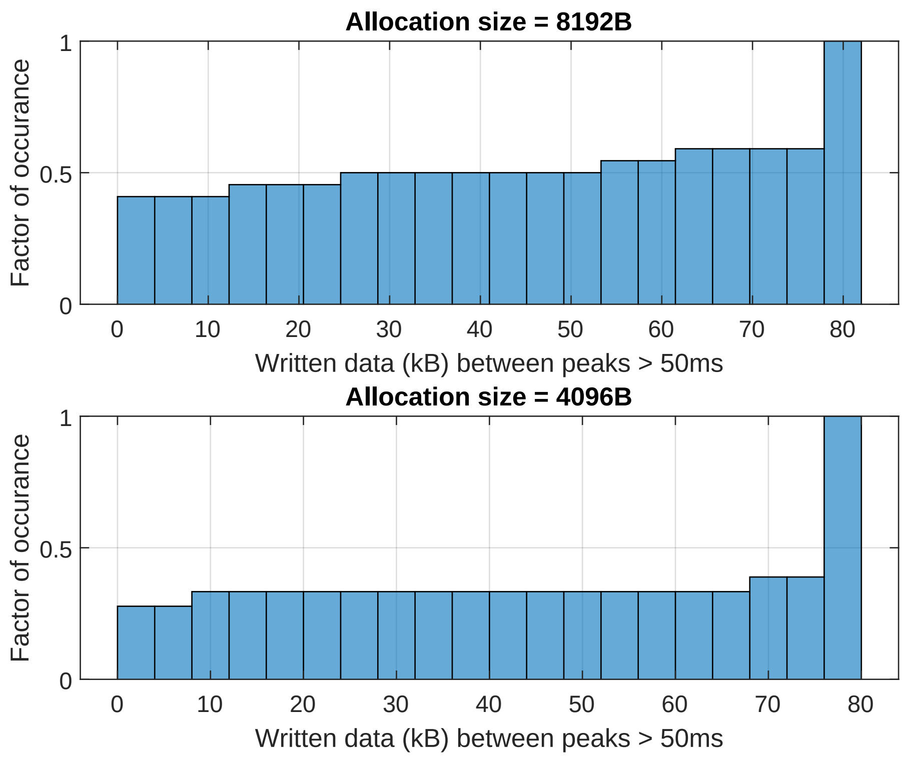


Looking into the small step at the start of a new file shows its dependency on the chosen allocation size in Figure 4.7.

When writing to a new file the write time of a set amount of bytes is about 13% smaller until the file is equal or larger than the set allocation size.

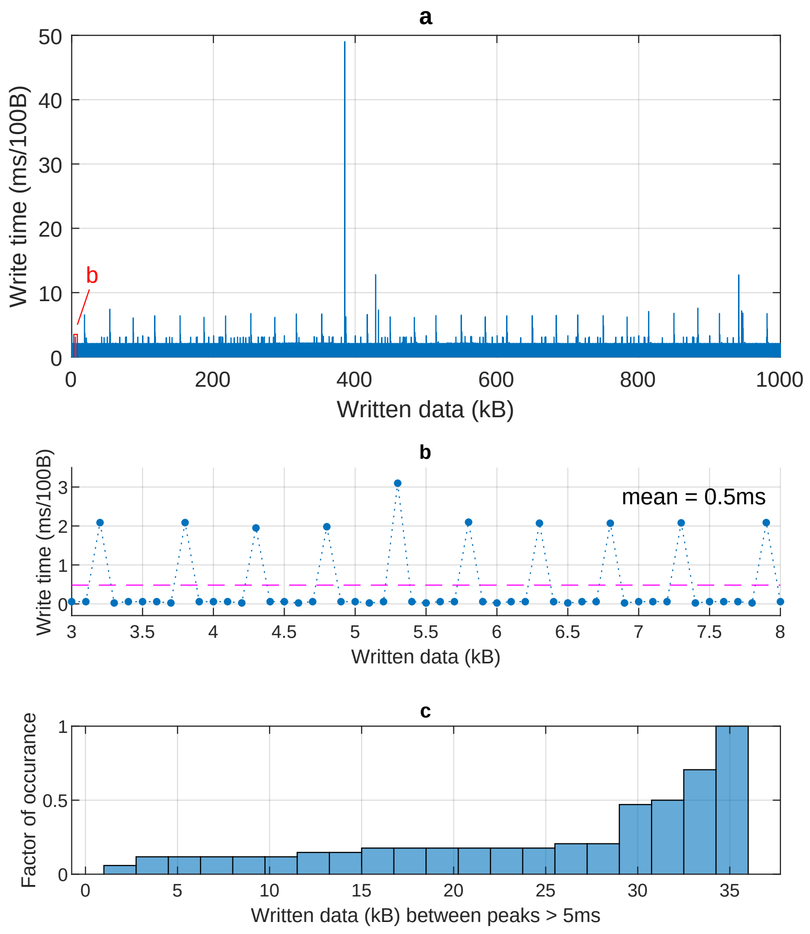
Figure 4.8

Written bytes between peaks in Figure 4.6 higher than $50\,\mathrm{ms}$ *plotted as a cumulative distribution for two different allocation unit sizes. The amount of written bytes between peaks is not significantly dependent on the allocation size, the majority of peaks still occur every* $80\,\mathrm{kB}$



Analysing Figure 4.6 using a cumulative distribution and comparing that with the same test with a smaller allocation size of $4096\,\mathrm{B}$ shows that the $80\,\mathrm{kB}$ steps are not dependent on it. This can be seen in figure 4.8. The previously discovered $80\,\mathrm{kB}$ period is independent of the allocation size.

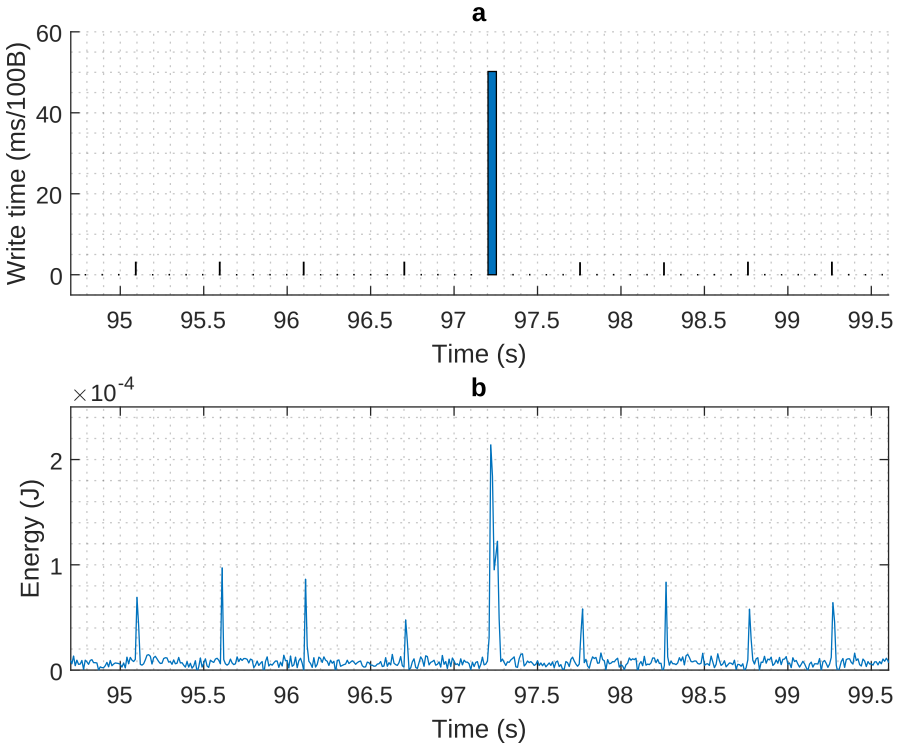
Figure 4.9
(a) *Write time of 1000, $100\,\mathrm{B}$ lines to 10 files stored in the same folder, keeping the file open between writes, versus the amount of written bytes. The write time in general is lower compared to figure 4.6, due to the file not being opened and closed on every write.* ***(b)*** *In the zoomed in plot there are portions where the write time is orders of magnitude lower between apparent $2\,\mathrm{ms}$ peaks. Either 4 or 5 of these writes occur between the peaks, there are 8 sections between each 5 write block. Average write time is $0.5\,\mathrm{ms}$ (dashed magenta line)* ***(c)*** *About half of the peaks larger than $5\,\mathrm{ms}$ are more than $30\,\mathrm{kB}$ apart.*



So far the tests have simulated the MagOD2 system closely, however analysing what happens if the file is kept open instead of being closed between every write proved to be interesting as can be seen in figure 4.9.

If the file that is being written to is kept open between writes it can be observed that the write time is; lower in general, has peaks periodic over about $30\,\mathrm{kB}$ and has periods where the write time is in the $\mathrm{\mu s}$ range instead of the ms range.

Figure 4.10
***(a)** Write time of 1000, $100\,\mathrm{B}$ lines to 10 files stored in the same folder every $100\,\mathrm{ms}$, keeping the file open between writes, versus the elapsed time. Note the very small dots of $\mu\mathrm{s}$ write time between the $3\,\mathrm{ms}$ bars **(b)** Compared to the energy usage of the SD card ($\Delta t = 10\,\mathrm{ms}$). The small $2\,\mathrm{ms}$ and large $50\,\mathrm{ms}$ peaks line up with the energy usage of the SD card while in the flat portions of the write time plot the energy usage is much closer to $0\,\mathrm{J}$. The energy usage is higher if the write time is longer.*



In order to analyse the very low write time in the zoomed-in portion of figure 4.9 a energy usage measurement was done on the VCC line of the SD card, showcasing a very low energy consumption in those portions of figure 4.10.

The peaks in write time are very bad from a consistency point of view, the write time is increased by 1500%. The origin of these inconsistencies has been traced back to the SD card as spikes in energy consumption can be seen during these peaks in write time. It can't be said for certain what it is in the SD card that takes such a long time to process a mere $100\,\mathrm{B}$ write. By looking at the energy consumption, there is basically no energy consumed in the periods where the write time is in the $\mu\mathrm{s}$ range.

To look at the effect of storing a large number of files in a single folder, a lot of small files were written to the same folder the effects are shown in figure 4.11.

When storing a lot of files in the same folder, the lower limit of the write time becomes progressively larger for every $16\,\mathrm{kB}$ that is stored in the folder in steps of $4\,\mathrm{ms}$ in most cases, the difference between the lowest and highest write time also grows on the same interval. If these parameters are dependent on for example file size, write size or allocation size was not investigated further.

Figure 4.11

(a) *Write time of* $10,100\,\mathrm{B}$ *lines to* 1000 *files stored in the same folder, closing the file between writes, versus the number of written bytes. The write time and the write time deviation increase as the number of written bytes and with it the number of files in the folder increase. The orange arrows indicate 3 points where the staircase like steps are larger than in all other areas, coinciding with a small peak. The very large spikes seem to be on a sort of interval like was analysed in figure 4.8.* ***(b)*** *There are 16 steps between the two larger steps.* ***(c)*** *The larger steps are about twice the size of the regular steps, both are consistent across all the steps.* ***(d)*** *Every step is about* $16\,\mathrm{kB}$ *in width*

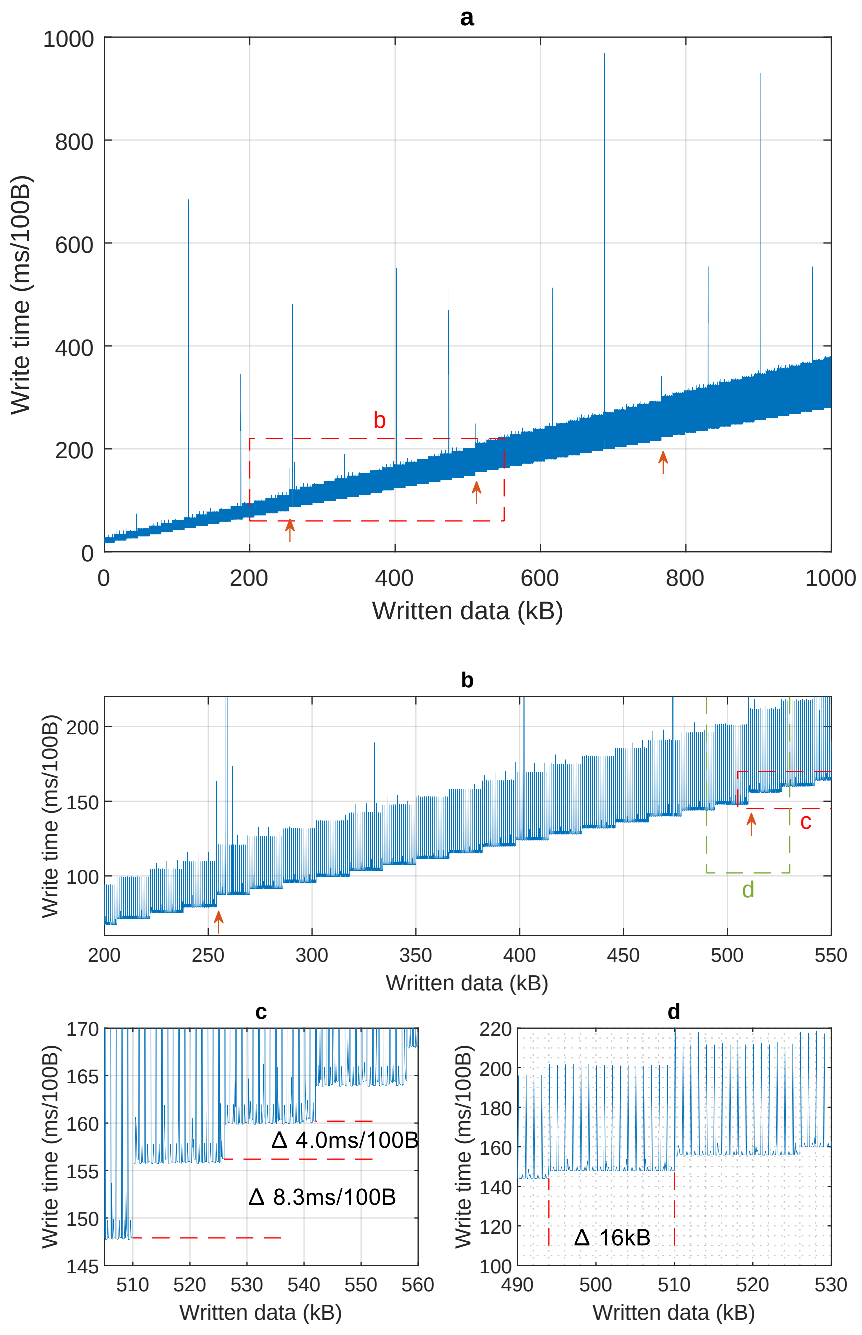
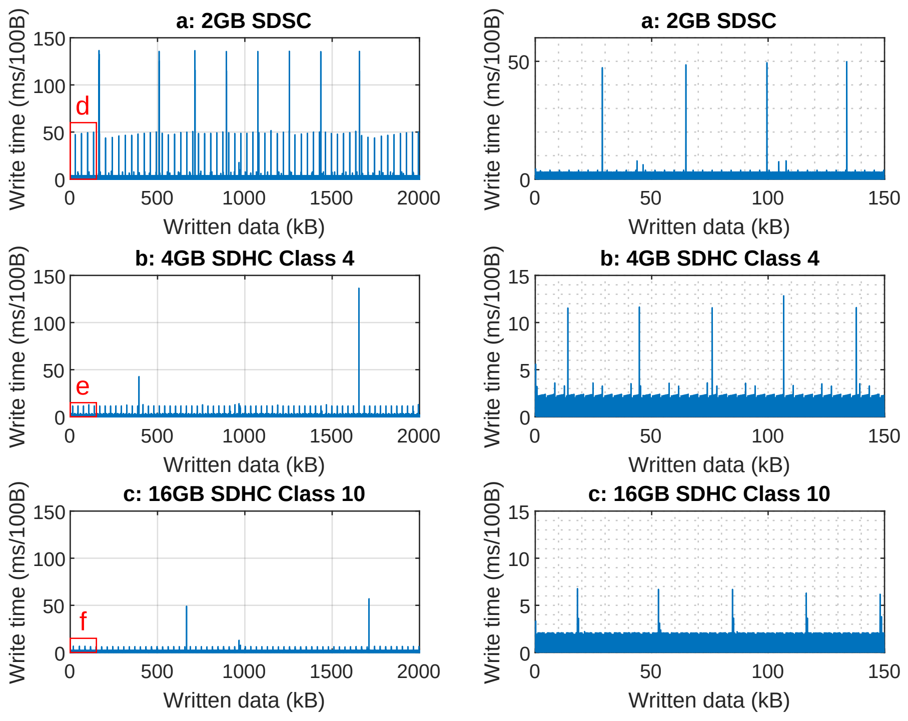


Figure 4.12
(a,b,c) *Write time of* $20000,\ 100\,\mathrm{B}$ *lines to a single file, versus the amount of written data. it can be seen that the higher class cards have fewer and or lower peaks in the write time.*
(d,e,f) *Zoomed in versions of the left side of the figure, the* $\approx 35\,\mathrm{kB}$ *separated peaks that appear on all three cards are lower for the higher class cards. Note the varying y-axis to make the peak height readable.*

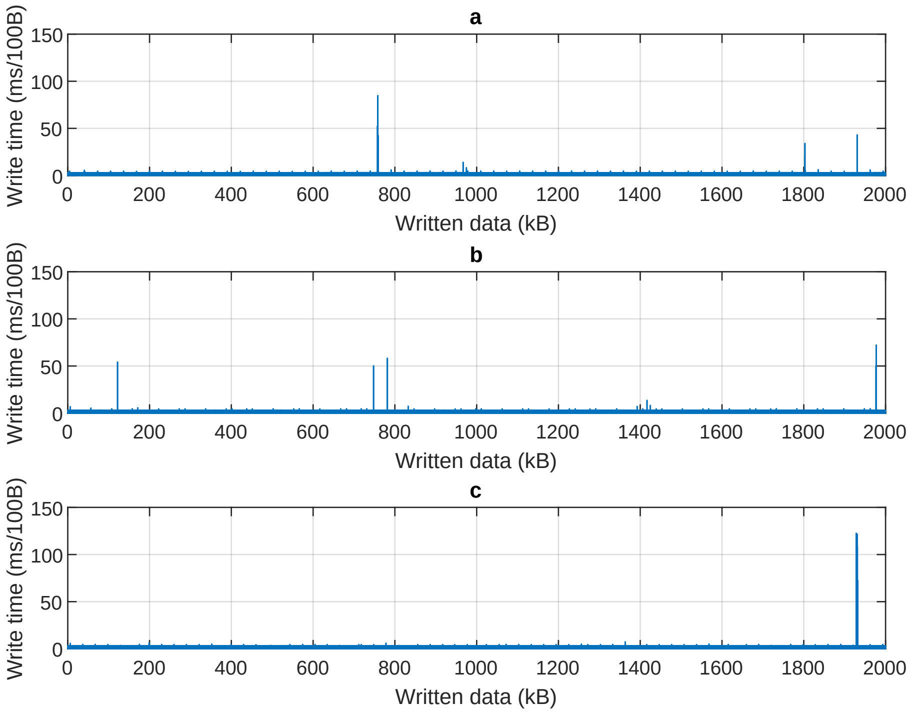


To get a better understanding of the write time spikes, three different SD cards (the same three cards from figure 4.2) were written to continuously for $2\,\mathrm{MB}$ in $100\,\mathrm{B}$ sections.

It can be seen that regular peaks in write time do occur and that the peaks of the class 4 card are 70% higher than those of the class 10 card.

Figure 4.13

(a) *Write time of* 20000*,* $100\,\mathrm{B}$ *lines (*$2\,\mathrm{MB}$ *total) to a single file with* $100\,\mathrm{ms}$ *delay between each write, versus the amount of written data. The file is kept open between writes. Even though the occurrence rate is low, there are still peaks to over* $50\,\mathrm{ms}$ *but not periodic.* ***(b)*** *Write time of* 400*,* $100\,\mathrm{B}$ *lines (*$40\,\mathrm{kB}$ *total) to* 50 *files for a combined total of* $2\,\mathrm{MB}$ *with* $100\,\mathrm{ms}$ *delay between each write. The files are kept open between writes, the opening and closing of the files is not included in the write time measurement. Peaks still reach to about* $75\,\mathrm{ms}$*.* ***(c)*** *Write time of* 80*,* $100\,\mathrm{B}$ *lines (*$8000\,\mathrm{B}$ *total) to* 250 *files for a combined total of* $2\,\mathrm{MB}$ *with* $100\,\mathrm{ms}$ *delay between each write. The files are kept open between writes, the opening and closing of the files is not included in the write time measurement. The write time is very consistent in the area where the* $120\,\mathrm{ms}$ *peak is not located*



As a test to get around the high write time peaks three different philosophies were tested. A $2\,\mathrm{MB}$ file written to in bursts of $100\,\mathrm{B}$ with $100\,\mathrm{ms}$ delay between every write without closing the file. The idea is to allow for garbage collection to occur in between writes and therefore prevent the peaks due to garbage collection. Next files are written to a maximum size of $40\,\mathrm{kB}$, judging by the distribution of figure 4.8 the peaks are usually $80\,\mathrm{kB}$ apart meaning those should be undercut in this case. Once again the file will be kept open when writing to the same file. The time of opening and closing the file in not included in the measurement. Finally, files are written up to a maximum size of $8000\,\mathrm{B}$ undercutting the allocation size of $8192\,\mathrm{B}$. Knowing the effect of storing many files in a single folder, every file was written to its own folder.

When leaving $100\,\mathrm{ms}$ between each write and keeping either the file open or closing it every time, peaks in write time still occur. Meaning that garbage collection is at least not the sole reason for the peaks. That garbage collection can not run in the background in SPI mode or that $100\,\mathrm{ms}$ is simply not enough. As another possible way to get rid of the write time peaks a test was done with a filesize of $40\,\mathrm{kB}$ and $8000\,\mathrm{B}$. However, neither of these options proved to be a solution to the peaks. It can be said however that the peaks in write time occurred at a seemingly lower rate for the smaller file size.

4.3 Overview

In order to give an overview of the results, they will be presented below in two tables in order to better condense the information and ease comprehension. Firstly the parallel transmission results will be presented in table 4.3. And the SD transfer results will be presented after in table 4.4

Table 4.3
Overview of results related to parallel computation and web server performance

Description	**Result**
Raw computation on core 0 compared to core 1	0.36% faster
Simultaneous computation on both cores compared to computing twice on one core	18% longer
Change in download speed by moving webserver from core 1 to core 0	17% slower
Change in write time on core 1 due to simultaneous file transfer on core 0	12% longer
Change in download speed due to simultaneous file write, when opening and closing the file for every write	13% slower
Change in download speed due to simultaneous file write, when keeping the file open between writes	4.1% slower

Table 4.4

Overview of results related to SD file transfers

Description	Result
Lower class 4 cards can outperform class 10 cards	6.8% faster
Write time peaks of class 4 cards are larger than the peaks of the class 10 card	70% higher peaks
Increasing the default SPI bus speed from $4\,\mathrm{MHz}$ to $26.6\,\mathrm{MHz}$ improves read speed	150% faster
By increasing the buffersize on the ESP the read speed can be improved	0.7% faster
By choosing a filesize very close to the chosen allocation size read speed is decreased	5.6% slower
When writing data to a new file the write time is initially lower until the filesize surpasses the allocation size	13% faster
Likeliest amount of bytes between write peaks of over $70\,\mathrm{ms}$	$80\,\mathrm{kB}$
Increase in write time at highest write time peaks due to the SD card (as seen by the aligned energy consumption)	1500% slower
Keeping a file open between writes lowers write time in comparison to opening and closing between writes	4300% lower
If a file is kept open for writing there are either 4 or 5, $100\,\mathrm{B}$ fast writes between the slow writes. The slow writes are much slower than the fast writes	3500% slower
Likeliest amount of bytes between write peaks of over $5\,\mathrm{ms}$ when the file is kept open	$30\,\mathrm{kB}$
If a lot of files are written to the same folder the lower limit in write time becomes larger in constant steps	$\text{step height} = 4\,\mathrm{ms}$, $\text{step length} = 16\,\mathrm{kB}$

Chapter 5

Discussion

Knowing the performance of an HTTP web server that is running in parallel, the required performance of $117\,\mathrm{kB\,s^{-1}}$ can be reached independently of the application that was tested. In every scenario, the minimum data rate was reached. However, if implementing such a web server one should make sure that when writing files to the same SD on the other core that the file is kept open between writes as that greatly improves the performance of the webserver.

When writing to data to SD cards over SPI special care has to be taken in selecting the SD cards that will be used. Looking at the performance between different cards from a transfer speed and consistency perspective tells a different story for both factors. Meaning the best card in one scenario might not be the best in all scenarios, depending on if average transfer speed or performance consistency is more important.

Continuing on the topic of consistency, many attempts have been made to circumvent the large peaks in write time, however, no solution has been found. It has been verified that the origin of the delays most likely resides in the SD card itself, indicated by the increased energy consumption during said peaks. It is known that flash chips require a form of garbage collection, wear levelling and other practices to guarantee the lifetime of the flash chips. However, the exact practices used on SD cards are not known as such methods are not required by the SD card specification documents. Meaning in theory a manufacturer could produce SD cards without any wear levelling techniques, severely limiting the possible lifetime of the flash chip inside. In general, it is usually possible for garbage collection like tasks to be performed in the background, however as there are no requirements for this a manufacturer could opt to only implement this in the regular SD operation and not in SPI mode. From the SD card documentation, it is known that the SPI communication protocol is more limited in both features and transfer speed compared to the regular SD protocol.

The spikes in write time may be caused by one of the following reasons, sorted on most likeliness top to bottom.

- Foreground garbage collection
  - As background GC is not implemented in SPI mode
  - As background GC is not implemented in the SD SPI library for the ESP32
- Wear levelling
  - Pages requiring multiple tries to be written correctly

Further research is required to be able to determine what in the SD card causes the significant peaks and to be able to find a possible way to mitigate these peaks.

If the current implementation of SPI SD cards is to be used anyway there are some ways to make sure to maximize its performance. For starters, not closing the to be written file unnecessarily has the largest speed advantage while risking data loss, this is a design choice. Increasing the SPI bus speed has a large positive effect too. Also, making sure that the to be saved file is not very close in size to the formatted allocation size of the SD card will help in this regard. Finally storing many files in a folder not a good idea as seen in figure 4.11, the assumption is that the slow down is caused by having to go over all files in a folder before a new file can be written. Meaning splitting up the files over different folders solves this issue, no significant extra delay was seen over the writes in figure 4.6 where 10 files were saved in the same folder compared to the 1000 files in figure 4.11.

Next, there are also aspects where their effect on write time is unknown. In figure 4.9 at maximum 5 writes of $100\,\mathrm{B}$ with the $\mu\mathrm{s}$ write time occur between the ms peaks. As no significant energy is consumed in this time it is most likely that the data is merely written to a buffer on the ESP32, this buffer appears to be slightly larger than $500\,\mathrm{B}$ (judging by the mostly 4 and sometimes 5 fast writes between the slow writes) meaning $512\,\mathrm{B}$ is a logical option. It is unknown if this buffer can be changed in size and if that has an effect on the transfer speed. Something can also be said of the implementation of the SD card in general, why is it that the microcontroller is halted while the SD transfer is happening if it could in theory just be handled by the SPI interface once the task has been given. It is understandable that during a file read the microcontroller would have to wait for the transfer to complete. However, during a write there seems to be no particular reason other than keeping the process sequential and with it predictable. Perhaps there could be a workaround to this using a different SD or SPI library for example.

Summarizing for the MagOD2 measurement system, if the mainboard is not to be changed and the SPI bus can not be altered the best option it to save measurements after the complete measurement cycle (recipe) is completed. This is with the current knowledge the only way to guarantee that possible peaks in write time won't have an effect on the consistency of the measurement. The measurements could be saved to an internal buffer first and after the recipe is completed the data could be flushed to the SD card. In which case the time it takes to complete this transfer is much less important. If the file write is no longer happening continuously that would also be beneficial for an HTTP web server implementation as it would have free access to the SD card while the measurement is happening, possibly reducing its effect even further. The fact that the transfer will be slower when the measurement data is flushed to the SD card should not impact the performance very much. Knowing that if the measurement file is not closed midway through the writing the impact on the wireless transfer speed is limited.

Chapter 6

Conclusions

We have investigated the effect of writing data to an SD card and wireless transmission of that data on the consistency of the write performance. It can be concluded that inconsistencies due to the SD card cause peaks in write time of over 1500% that are unpredictable in both heights and in occurrence. The webserver running in parallel hurts the write performance by about 12%.

General write performance can be increased by 4300% if the to be written file is kept open between writes, 150% by increasing the SPI bus speed to $26.6\,\mathrm{MHz}$, 6.8% through choosing the correct SD card and 5.6% by choosing the allocation size an order of magnitude lower than the file size. Though none of these methods lowers the write time peaks below $50\,\mathrm{ms}$. Webserver download performance is decreased 17% by moving the server to core 0 and a further 13% if the other core is writing data and closing the file between writes or only 4.1% if the file is kept open.

In conclusion, the write time to an SD card over an SPI interface using the ESP32 microcontroller is unpredictable due to the at minimum $50\,\mathrm{ms}$ SD card induced peaks in write time that occurred in all scenarios. The effect of a webserver on the write time can be minimized through careful planning of the implementation

Bibliography

[1] “Wi-Fi Driver - ESP32 - Wi-Fi Throughput.” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#esp32-wi-fi-throughput

[2] “ESP-IDF FreeRTOS.” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html

[3] “Overview of features ESP32.” [Online]. Available: https://www.exploreembedded.com/wiki/Overview_of_ESP32_features._What_do_they_practically_mean%3F

[4] “SD Specifications Part 1 Physical Layer Specification Simplified Specification,” Tech. Rep., 2020. [Online]. Available: https://www.sdcard.org/downloads/pls/index.html.

[5] “Speed Class - SD Association.” [Online]. Available: https://www.sdcard.org/developers/overview/speed_class/

[6] “SPI Master Driver - ESP32.” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_master.html

[7] “Understanding Flash: Blocks, Pages and Program / Erases,” 6 2014. [Online]. Available: https://flashdba.com/2014/06/20/understanding-flash-blocks-pages-and-program-erases/

[8] “Understanding Flash: The Flash Translation Layer,” 9 2014. [Online]. Available: https://flashdba.com/2014/09/17/understanding-flash-the-flash-translation-layer/

[9] “Understanding Flash: The Write Cliff,” 11 2014. [Online]. Available: https://flashdba.com/2014/11/24/understanding-flash-the-write-cliff/

[10] S. Anatolievich Gorobets, “MANAGING HOUSEKEEPING OPERATIONS IN FLASH MEMORY ,” SANDISK CORPORATION, Edinburgh, Tech. Rep., 5 2007. [Online]. Available: https://patentimages.storage.googleapis.com/16/a9/1f/52d10b1a2a49bc/US20080294813A1.pdf

[11] J. Lee, Y. Kim, G. M. Shipman, S. Oral, and J. Kim, “Preemptible I/O Scheduling of Garbage Collection for Solid State Drives,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 2, pp. 247–260, 2013.

Appendix A

SD consistency function

```
void SDwriteConsistencyTest(uint16_t bytesPerWrite,
    uint16_t linesPerFile, uint16_t totalFiles,
    uint16_t delayBetweenWrites, bool openCloseFile) {

    // Variables to store results in
    uint32_t startTime[totalFiles][linesPerFile];
    uint32_t endTime[totalFiles][linesPerFile];
    uint32_t startWriteTime;
    uint32_t endWriteTime;

    char bytesToWrite[bytesPerWrite-1];
            // Two characters less as the return and new line character will be attached one extra to add \0 to end the character array
    uint32_t startTestTime;
    uint32_t endTestTime;

    uint16_t nFiles;
    uint16_t nLines;
    uint16_t nBytes;
    String fileName;
    File dataFile;

    // Create folder to store the temp data in
    fileName = "/writeTimeTemp/test_" + (String)nTests;
    createDir(fileName.c_str());

    Serial.print("[Test]␣Starting␣test␣"); Serial.println(nTests);

    Serial.println("[Setup]␣Creating␣variable␣to␣write");
    memset(bytesToWrite, '~', sizeof(bytesToWrite)-1);
        // Fill array with the set amount of ~ characters (total minus 2)
    *(bytesToWrite + sizeof(bytesToWrite) - 1) = '\0';
        // Set final byte to /0 to close of the array

    Serial.println("[Test]␣Writing␣files");
    if (openCloseFile == true) {
        // If opening and closing the file on every write (placed here and not in loop to eliminate extra if statement that could have effect on execution speed of the loop)
      startTestTime = micros();
      for (nFiles = 0; nFiles<totalFiles; nFiles++) {
          // Loop over all to be written files
        fileName = "/writeTimeTemp/test_" + (String)nTests + "/temp_" + (String)nFiles + ".txt";
            // Set file name to write to

        for (nLines = 0; nLines<linesPerFile; nLines++) {
            // Loop over all lines that should be written to the file
          startWriteTime = micros();
              // Log the start time of the write in us
          dataFile = SD.open(fileName, FILE_APPEND);
              // Open file now
          dataFile.println(bytesToWrite);
              // Write bytes to file
          dataFile.close();
              // Close file now
          endWriteTime = micros();
              // Log the end time of the write in us

          startTime[nFiles][nLines] = startWriteTime;
              // Save the start and end time of the file write
          endTime[nFiles][nLines] = endWriteTime;
          delay(delayBetweenWrites);
              // Delay the set amount of time
        }
      }
      endTestTime = micros();
    } else {
        // If not opening and closing the file on every write
      startTestTime = micros();
      for (nFiles = 0; nFiles<totalFiles; nFiles++) {
          // Loop over all to be written files
        fileName = "/writeTimeTemp/test_" + (String)nTests + "/temp_" + (String)nFiles + ".txt";
            // Set file name to write to

        dataFile = SD.open(fileName, FILE_APPEND);
            // Open the file now

        for (nLines = 0; nLines<linesPerFile; nLines++) {
            // Loop over all lines that should be written to the file
          startWriteTime = micros();
              // Log the start time of the write in us
          dataFile.println(bytesToWrite);
              // Write bytes to file
          endWriteTime = micros();
              // Log the end time of the write in us

          startTime[nFiles][nLines] = startWriteTime;
              // Save the start time
          endTime[nFiles][nLines] = endWriteTime;
              // Save the end time
          delay(delayBetweenWrites);
              // Delay the set amount of time
        }
        dataFile.close();
            // Close file now
      }
      endTestTime = micros();
    }

    Serial.println("[Test]␣Test␣complete,␣saving␣data...");

    //Count the amount of files already in the folder
    dataFile = SD.open("/writeTimeResults");
    dataFile.rewindDirectory();

    File entry;
    for(nFiles = 0; true; nFiles++) {
        entry = dataFile.openNextFile();
        if (!entry) {break;}
        entry.close();
    }

    dataFile.close();

    // Set new save data filename based on the amount of files in the folder (don't overwrite old files)
    fileName = "/writeTimeResults/results_" + (String)nFiles + ".csv";
    Serial.print("[Info]␣Saving␣results␣as:␣"); Serial.println(fileName);

    // Open results file
    dataFile = SD.open(fileName, FILE_APPEND);

    // Save test information
    dataFile.println("Test␣information:");
    dataFile.print("Bytes␣per␣line,"); dataFile.println(bytesPerWrite);
    dataFile.print("Lines␣per␣file,"); dataFile.println(linesPerFile);
    dataFile.print("Amount␣of␣files,"); dataFile.println(totalFiles);
    dataFile.print("Delay␣between␣writes,"); dataFile.println(delayBetweenWrites);
    dataFile.print("Open␣and␣close␣file␣between␣writes,"); dataFile.println(openCloseFile);
    dataFile.print("Test␣start␣time␣(us),"); dataFile.println(startTestTime);
    dataFile.print("Test␣end␣time␣(us),"); dataFile.println(endTestTime);
    dataFile.print("Text␣string,"); dataFile.println(bytesToWrite);
    dataFile.println("␣");

    // Save test results
    dataFile.println("Test␣results␣(us):");
    for (nFiles = 0; nFiles<totalFiles; nFiles++) {
            // Create header for save data
            dataFile.print("File␣");
            dataFile.print(nFiles);
            dataFile.print("␣start,");
            dataFile.print("File␣");
            dataFile.print(nFiles);
            dataFile.print("␣end,");
        }
        dataFile.println("␣");

        for (nLines = 0; nLines<linesPerFile; nLines++) {
                // Loop over the different lines
            for (nFiles = 0; nFiles<totalFiles; nFiles++) {
                    // Loop over the files
                dataFile.print(startTime[nFiles][nLines]);
                        // Write the results
                dataFile.print(",");
                        // Add comma for .csv seperation
                dataFile.print(endTime[nFiles][nLines]);
                        // Write the results
                dataFile.print(",");
                        // Add comma for .csv seperation
            }
            dataFile.println("␣");
                    // Print a new line
        }
        dataFile.close();
                // Close results file

        Serial.println("[Test]␣Data␣saved␣succesfully!");

        nTests++;
                // Increase test counter

        return;
}
```