Spring 2014

DEVELOPING AN EMBEDDED SYSTEM SOLUTION FOR HIGH-SPEED, HIGH-CAPACITY DATA LOGGING FOR A SIZE-CONSTRAINED, LOW-POWER BIOMECHANICAL TELEMETRY SYSTEM AND INVESTIGATING COMPONENTS FOR OPTIMAL PERFORMANCE

Brandon Blaine Gardner
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Brandon Blaine Gardner

Entitled
Developing an Embedded System Solution for High-speed, High-capacity Data Logging for a Size-constrained, Low-power, Biomechanical Telemetry System and Investigating Components for Optimal Performance

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

THOMAS M. TALAVAGE
Chair

MARK C. JOHNSON

NIKLAS E. ELMQVIST

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): THOMAS M. TALAVAGE

Approved by: M. R. Melloch 04-14-2014
Head of the Graduate Program Date

DEVELOPING AN EMBEDDED SYSTEM SOLUTION FOR HIGH-SPEED, HIGH-CAPACITY DATA LOGGING FOR A SIZE-CONSTRAINED, LOW-POWER BIOMECHANICAL TELEMETRY SYSTEM AND INVESTIGATING COMPONENTS FOR OPTIMAL PERFORMANCE

A Thesis

Submitted to the Faculty

of

Purdue University

by

Brandon Blaine Gardner

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2014

Purdue University

West Lafayette, Indiana

This thesis is dedicated to my grandparents, who always believed in me.

ACKNOWLEDGEMENTS

Special thanks to my advisory committee Tom Talavage, Niklas Elmqvist, and Mark Johnson for taking the time to review this work, and to my roommates/labmates, Adi, Diana, and Jeff, who helped me see the telemetry sensor to field prototype stage.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Gardner, Brandon Blaine M.S.E.C.E., Purdue University, May 2014. Developing an Embedded System Solution for High-Speed, High-Capacity Data Logging for a Size-Constrained, Low-Power, Biomechanical Telemetry System and Investigating Components for Optimal Performance. Major Professor: Thomas M. Talavage.

The Purdue Neurotrauma Group (PNG) seeks to develop a biomechanical telemetry system capable of monitoring and storing athletes' head motions with the intention of identifying when a player may be at risk of neurophysiological damage, especially brain damage. A number of commercially-available systems exist with a similar goal; however, each of these systems discards information below an acceleration threshold. Research by PNG indicates that any acceleration may contribute to brain damage and that, because of this, an event-based model is insufficient for a proper understanding of an athlete's neurophysiological health. Continuous-time monitoring of head accelerations is therefore necessary. To facilitate the collection and storage of continuous telemetry data, a high-speed sensor system with a sufficiently large amount of memory storage is required. Additional requirements include low power consumption, low cost, and a small form factor. It has been concluded that a microSD card is the memory technology most capable of meeting these requirements, despite a number of drawbacks, most notably a relatively slow data write speed. An embedded solution requiring the use of large data buffers was developed to combat this drawback. Various microSD cards were tested to determine base read and write speeds and whether differences exist between card manufacturers, card sizes, or card speed ratings. It was found that the base performance was nearly identical in each test. Recommendations are made based upon the testing results, enabling production of operational prototypes for field evaluation.

1. INTRODUCTION

1.1 Background

The long-term effects of traumatic brain injuries (TBIs) are not well understood. The New York Times reports that retired professional football players are nineteen times more likely to develop dementia than the general population [1]. In a communication with Thomas M. Talavage, he notes that "this dementia has often been found to be associated with chronic traumatic encephalopathy, an Alzheimer's-like neurodegenerative disease." Additionally, evidence suggests that the level of neural tissue damage exhibited is not commensurate with the known history of diagnosed concussions. Recent research suggests that the increased level of damage is likely due to repetitive exposure to sub-concussive impacts [2].

Critically, the degenerative effects of these sub-concussive impacts are likely to go unnoticed [3], and thus represent a silent danger to the athlete. It is therefore no longer appropriate to evaluate and treat TBI only when a concussion is suspected. To effectively combat permanent brain damage, early detection methods must be developed. This also means that athletes not only of high-impact sports such as football, hockey, and boxing must be considered at-risk but also those of traditionally low-impact sports such as soccer, volleyball, and baseball.

The Purdue Neurotrauma Group (PNG) has been collecting data using the Head Impact Telemetry System (HITS or HIT System) [4] for many years with the eventual goal of developing predictive models to detect the likelihood of irreparable brain damage based upon a player's recorded impact history before the damage can occur [5]. There are a number of problems with the HIT System [6], however, and the accuracy of any models developed using HITS data is questionable. Recently, a number of similar yet still fundamentally flawed products have become available. The crux of each of these devices is that impacts are treated as discrete event windows outside of which no data is considered [7]. An event is marked by some time window in which head acceleration exceeds some arbitrary threshold. The inherent, unproven assumption underlying this approach is that no brain injury can occur below the threshold.

Thomas M. Talavage of the PNG cautioned in a personal communication that "past failure to accurately quantify and record all [accelerations] leading to an observed injury has likely led to improper attribution of [singular large accelerations]" – rather than the cumulative effect of all events leading up to and including the event in question – to the cause of TBI. He also offered the following illustration:

> Consider the "random incidence paradox" [8] which informs us that sub-concussive hits are not likely to be observed as the proximal cause of observable head injuries. Assuming that each individual possesses a fixed (but unknown) threshold of accumulated damage, beyond which clinically-observable symptoms will be present, this threshold is most likely to be exceeded by a larger, more damaging blow than by a smaller one. For example, if a player has an unknown symptom-producing threshold between 1 and 100 units, and experiences (in a random order) 50 blows producing 1 unit of damage, 3 blows producing 10 units of damage, and 1 blow producing 20 units of damage, the threshold is 50% likely to be crossed by one of the 4 blows producing 10 or more units of damage, even though these blows represent a mere 7.4% (4 out of 54) of the collision event history. Given the actual exponential-like distribution of the magnitude of blows observed in athletes (see [2]), the tendency for larger blows to occur at the time of crossing of the "concussion" threshold would be even greater than in this example. This is consistent with the concussion literature noted above, in which large blows are typically observed at the time of concussion, but with no clear relationship between magnitude and subsequent symptom severity.

It is clear that an event-based model of head trauma is not sufficient for a proper understanding of TBI. A continuous-time telemetry system must be developed to allow for proper formulation of predictive models.

1.2 Scope

The purpose of the Purdue Neurotrauma Group is to develop this continuous-time telemetry system in both hardware and software. This system must be simultaneously (1) small, so as to be worn in any sport with or without a helmet; (2) power efficient, so as to provide a device charge life of at least the length of an entire game of any commonly-played sport; (3) fast, so as to collect meaningful telemetry readings of impact events less than ten milliseconds in length; and (4) low-cost, so as to be affordable for the research group.

The earliest concepts of the device included wireless transmission of telemetry data to a nearby ground unit; however, the design challenges that would need to be overcome were notably steep, especially considering that two teams of athletes could be reasonably expected to be using the devices simultaneously. In the interest of producing a prototype device more quickly, PNG opted to settle for continuous data logging to an on-board memory device.

In addition to the design challenges mentioned earlier, the device now requires (1) a large memory capacity, so as to be able to log telemetry for at least the length of a battery charge, and (2) a computationally-light interface, so as to minimize the percentage of system processing resources required for data storage. From a device software standpoint, the challenges associated with the memory device have arguably been the most difficult to overcome.

The purpose of this thesis is to identify the design challenges encountered during the implementation of continuous-time data logging, focusing particularly on the software challenges, methods used, solution implemented, optimizations attempted, and results obtained.

1.3 Outline

Section two of this thesis describes the process used for selecting the memory device, a microSD card. It includes quantified project requirements and a comparison with NAND

Flash, the closest available candidate. This section also details the earliest failures to prototype the data logging functionality which led to the use of a microSD card in via serial peripheral interface (SPI).

Section three comprises the bulk of the document and introduces three alternative implementations are briefly examined, followed by a more detailed description of the implementation selected. Particular problem points encountered and the solutions required to address those problems are noted as well as attempts to optimize the solution for an embedded system. The final section describes a method of implementing basic file system capabilities to make use of one of SD's additional benefits.

Section four describes initial performance testing on a variety of microSD cards to cursorily determine if a particular card or subset of cards might have superior performance characteristics.

The members of the Purdue Neurotrauma Group immediately involved in the development of the biomechanical telemetry system include (1) Paul Rosenberger, primary developer of the earliest prototype device; (2) Jeffery R. King III, developer of wireless functionality; (3) Aditya Balasubramanian, primary hardware designer; (4) Brandon Blaine Gardner, primary software designer; (5) Thomas M. Talavage, primary advisor and neuroimaging expert; and (6) Eric A. Nauman, advisor and biomechanics expert.

2. SELECTION OF A MEMORY TECHNOLOGY

2.1 Memory Technology Requirements

2.1.1 Small Form Factor

To keep the device as transparent as possible to the user, space was one of the primary concerns for the memory technology. The ideal memory technology would take up zero extra space on the device. Realistically speaking, a device smaller than 0.75 inches by 1.5 inches and less than 0.25 inches thick was targeted [7]. The physical limitations guiding this requirement are entirely hardware related and thus out of the context of this document.

2.1.2 Low-Power

For a prototype version of the device, members of the Purdue Neurotrauma group (Tom Talavage, Eric Nauman) targeted four hours as the minimum battery life, as it would be suitable for most but not all sports. The ideal memory technology would consume zero extra power; however, this is clearly impossible. If a 1000 milliamp-hour battery was used, the memory device should consume no more than 200 milliamps to meet this requirement.

2.1.3 High-Speed

Early estimates required the memory technology to be able to store at minimum 52,000 bytes per second. The memory would need to be transferred either during device operation or during periods when the device was not performing any data collection. The following equation was used consistently throughout the development process to calculate the memory bandwidth required.

$$Bandwidth = (Bytes\ per\ data\ sample) \cdot (Sampling\ frequency)$$

2.1.4 Low-Cost

As the PNG is university affiliated and relies on grants for funding, a cost effective device was also important. A total device price of less than $100USD was desired. PNG also hoped to be able to commercialize the device, and this price point was determined to be competitive.

2.1.5 Large Capacity

To store a minimum of four hours of data at an estimated 52,000 bytes per second, the memory device was required to have a capacity of at least 714.2 megabytes[1] (748.8 million bits). Because PNG members (Tom Talavage, Eric Nauman) intend to use the devices to track the telemetry of many local sports teams, a much higher memory capacity was desired if possible. Doubling the memory would allow members to visit teams only every other game or practice session. Five times more memory would further reduce the visits to every week, which would require less PNG staffing; therefore, 3.5 gigabytes[2] (3.744 billion bits) was a more attractive capacity.

2.1.6 Computationally Light

The primary software requirement for the memory technology was that the interface should require little computation. This requirement was difficult to quantify and was mostly used to compare different memory technologies. A simple interface means that (1) the system spends less time managing memory and more time collecting data or conserving battery by entering a sleep state and (2) the development time spent implementing the interface would be minimal, allowing for rapid prototyping.

2.2 Selection of microSD via Serial Peripheral Interface

Based upon the above criteria, a microSD card operating in SPI mode was selected for the primary memory technology. Members of the PNG team (Aditya Balasubramanian, Jeff King, Brandon Gardner) compared all available non-volatile memory technologies, finding

---

[1] 1 megabyte equals 1048576 bytes
[2] 1 gigabyte equals 1073741824 bytes

that NAND flash and microSD were the only suitable candidates meeting the capacity requirements alone. A microcontroller with a sufficient amount of on-board Flash was the most desirable option; however, no microcontrollers found provided the available space.

Although there were only two available memory technologies, there were four options to consider. External control chips exist for both SD and NAND technologies that simplify the device interfacing. Table 2.1 summarizes the comparison between the four options.

Table 2.1. Comparison of Memory Technology Candidates

Device / Parameter	NAND Flash	NAND Flash with external controller chip	MicroSD in SPI mode	MicroSD with external controller chip
Size (LxW)	0.87” x 0.47”	0.87” x 0.8”	0.7” x 0.7”	1” x 0.7”
Max Current Consumption	50 mA	83 mA	200 mA (max) 80 mA tested	207 mA (max) 87 mA tested
Max (ideal) Bandwidth	6.25 MBps	50 kBps (insufficient)	3.125 MBps	3.125 MBps
Cost	2GB: $5-$30 (~$15)	2GB: $21-$47 (~$31)	2GB: $9-$10	2GB: $11-$12
Capacity	2 GB – 8 GB	2 GB – 8 GB	2 GB – 64 GB	2 GB – 64 GB
Computational Complexity	Dead sector maintenance required. Highest complexity.	Complexity estimated to be similar to microSD in SPI mode.	FatFs library available* (low development effort, but lots of code)	Is essentially NAND with external control chip.**

* Discussed in a later chapter

** High hardware complexity

At the expense of complexity, the first iteration of the prototype utilized NAND Flash without an external control chip. During the three month design cycle of this device, it was discovered that the complexity was much greater than anticipated. Dead sector maintenance was a huge software burden and was infeasible for rapid prototyping.

The next iteration opted to use microSD in SPI mode using the FatFs library [9]. FatFs was utilized in an example project from Code Composer Studio [10], allowing quick prototyping of the microSD interface, and although it was able to interface easily to the card, the library utilized too much processing overhead. In this iteration, the practical current consumption of two test cards was found to be a maximum of about 80 mA rather than the 200 mA maximum given by the SD card specification document [11].

For the third iteration, PNG members (Aditya Balasubramanian, Jeff King, Brandon Gardner) revisited the memory technology selection process from the beginning, arriving at the same results. The best option was to use a microSD card with an external interfacing chip; however, these chips were either too large or required too much hardware fabrication complexity for the team. Therefore, the third iteration also utilized the microSD card in SPI mode. The approach taken in this iteration was to implement code for the raw interfaces rather than use the FatFs library. This implementation constituted the primary difficulty of the project software and is covered in detail in section 3 below.

3. DESIGN OF A HIGH-SPEED DATA LOGGING INTERFACE TO AN SD CARD VIA A SERIAL PERIPHERAL INTERFACE

3.1 Interfacing Techniques

In a simple embedded system, there are generally three techniques available for implementing a serial peripheral interface (SPI) to a device like an SD card. The three techniques and the method used are discussed in the sections below. The analysis of these interfacing techniques was only valid for a loop-based program flow and not for a real-time operating system (RTOS). There are a number of advantages a RTOS holds versus a traditional loop-based program [12]; however, PNG team members (Brandon Gardner, Jeff King, Aditya Balasubramanian) were not aware of these benefits until late into the design phase.

3.1.1 Reentrant Loop-Based Technique

This technique relies on the main system to allocate a small amount of processing power periodically to operation of the SPI bus. Each time the system calls the SPI subsystem process a byte will be transmitted and received. For example, to send and/or receive eight bytes of data, the SPI process must be called eight times, one for each data byte. It is worth noting that this technique requires some method of counting the number of bytes to be transacted.

The primary benefit of this technique is that the main system is able to allocate processing time to the microSD card as it is available, meaning the system will always be able to collect data. The drawback to this method, however, is that if the main system uses too much processing time on other tasks and does not allocate enough to SPI transactions, the data logging bandwidth will fall below data collection bandwidth causing a buffer overflow condition. Additionally, detecting when a buffer overflow condition is likely to occur in the system is also very difficult using this method.

3.1.2 Interrupt-Based Technique

This technique utilizes system interrupts to transmit and receive bytes via SPI. The main system is able to instigate a multi-byte transaction with a single function call rather than multiple. The system interrupt is responsible for transmitting and receiving the proper data bytes. When an interrupt is triggered, the system collects the received byte and transfers a new byte to the bus. This method must also keep count of the number of bytes to be transacted, stopping when there are no more bytes to be transmitted.

Compared to the loop-based technique, performance of the SPI subsystem will not degrade as the main system's processing increases. The amount of processing used by each interrupt is comparable to the processing used by each call of the loop-based technique. The main system is not easily able to control when the SPI subsystem is able to perform processing, however. Additionally, although a good optimizing compiler might be able to reduce the function call overhead from the first technique, the interrupt execution latency and time required to execute a return from interrupt instruction cannot be optimized.

3.1.3 Direct Memory Access-Based Technique

This technique takes advantage of the system's direct memory access controller to control the transaction and reception of SPI data. Similarly to the interrupt-based technique, the main system is able to initiate a multi-byte transaction with a single function call. Instead of an interrupt routine, the DMA controller is able to transfer a byte with a single main system clock cycle; two cycles would be needed for each SPI transaction byte, one for transmitting and one for receiving. This technique relies on the availability of DMA channels in the system, an often limited resource.

Although the initialization required for this method is more complex than that of the interrupt-based routine, its operation uses fewer system resources, using only two clock cycles per byte transacted versus tens to hundreds for interrupt-based operation. This method is further improved by the lack of a need to keep count of the number of bytes to be transacted.

3.1.4 Selection of a Technique

The benefits of the loop-based technique are desirable; however, the primary drawback suggested the other techniques. It is clear that the DMA-based technique is superior, but the ease of set-up led to use the interrupt-based technique initially. This led to an unanticipated problem during development. At times, the main system was unable to process more than a few instructions, preventing the system from collecting new data. The interrupt routine was using a large percentage of the available instruction cycles, leaving the main system without adequate processing capabilities.

Consider the simplified example in Figure 3.1. From the figure, it is easy to see that the primary system processing is left less with than 50 percent of the available instruction cycles. Figure 3.2 uses the same scenario to illustrate the instruction cycle usage of the DMA-based approach, finding that only two cycles out of every 16 are used for SPI transactions. This leaves 87.5 percent of the available instruction cycles are available to the main system. In testing, this has been an acceptable efficiency.

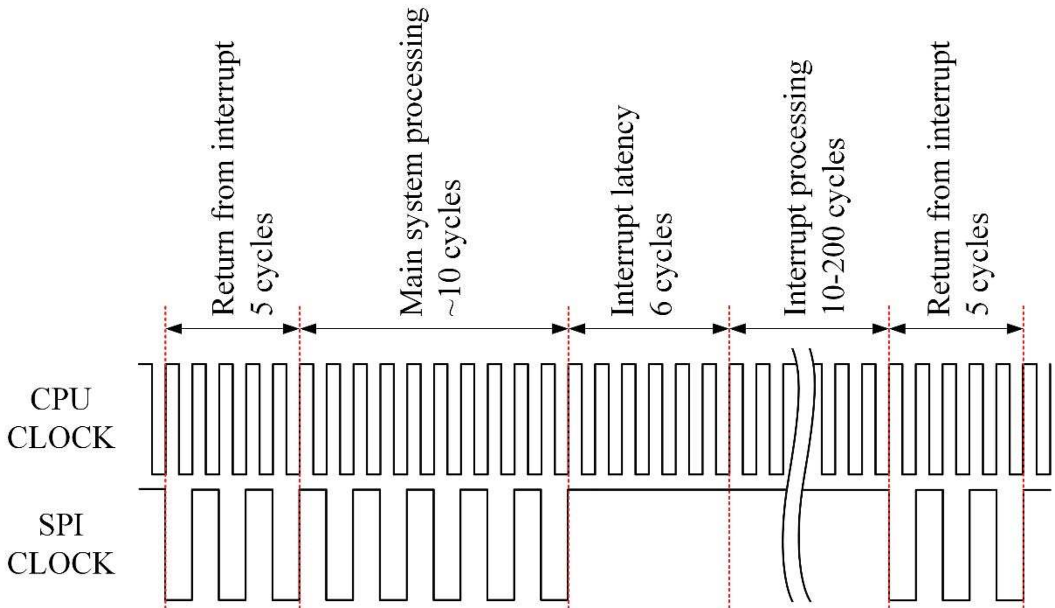

Figure 3.1. Example scenario demonstrating CPU clock cycle usage during an interrupt-based SPI transfer of one full and one partial byte. SPI clock frequency is half that of the CPU clock frequency. Comments indicate which system process utilizes the indicated CPU clock cycles.

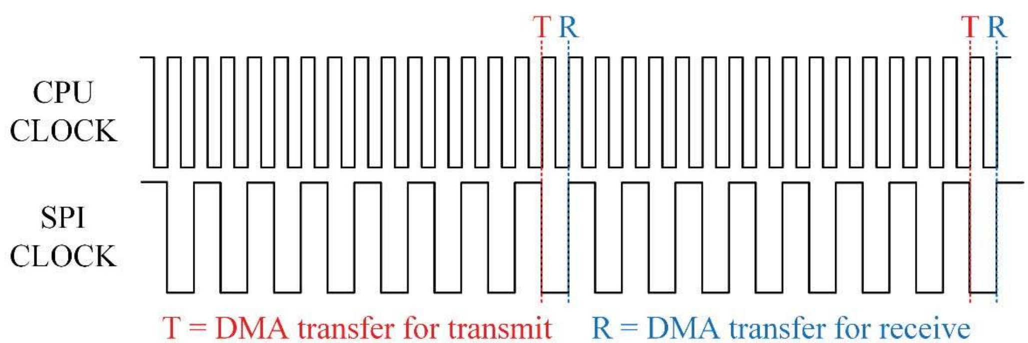

Figure 3.2. Example scenario demonstrating CPU clock cycle usage during a DMA-based SPI transfer of two bytes. SPI clock frequency is half that of the CPU clock frequency. A blue or red marker indicates that a particular clock cycle is used for a DMA transfer.

3.2 Development of an SD Control Process

3.2.1 Communication Scheme

The FatFs library [9] was used as a guide for designing the microSD card control software, and the SD specification [11] was constantly referenced to ensure accuracy. When information from the SD specification was vague or non-existent, a SanDisk SD card product manual [13] was used to supplement the information. Communication to the card includes both commands and data transfers and follows a command and response format, shown in simplified form in Figure 3.3. The "Clock Control" section of the SD specification document describes the requirement of eight extra clock cycles in a number of circumstances during SD card communication; these eight clock cycles are implemented as a "processing byte" during which data must be 0xFF hexadecimal.

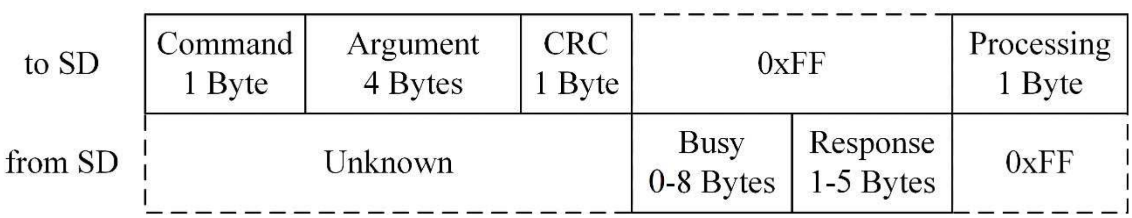

Figure 3.3. SD card simplified command and response format.

Before implementing communication with the card, the commands that would be necessary for a data logging application were identified, ignoring any command that was unnecessary or extraneous. This first began with identifying the commands required for device initialization. For data logging, writing to a card was the most important feature. Reading from a card was also necessary in order to facilitate the ability to determine where in a microSD card's memory data should be placed so that data from previous device sessions was not overwritten. The ability to read and write multiple blocks was deemed more useful than the ability to read and write only a single block. The SD commands required are summarized in Table 3.1. With these commands, it was possible to read from or write to any 512-byte block of a microSD card. Reading from or writing to any specific subset of the block was impossible, instead requiring the entire block to be accessed.

Table 3.1. Minimal commands required for data logging.

Command	Response	Usage
CMD0	R1	Card reset (Initialization)
CMD8	R7	Voltage check (Initialization)
CMD58	R3	Read operation condition register (Initialization)
CMD55	R1	Application command (Used before ACMD commands)
ACMD41	R1	Activate card initialization
CMD18	R1	Read multiple blocks
CMD12	R1b	Stop transmission of data blocks (following CMD18)
CMD25	R1	Write multiple blocks

In addition to the device command format, an SD card follows a format for data transmission shown in Figure 3.4. For high-capacity SD cards, the data block length is fixed to 512 bytes, but for standard-capacity SD cards, the block length can be specified. The design used the fixed default block length of 512 bytes for maximum compatibility and ease of implementation.

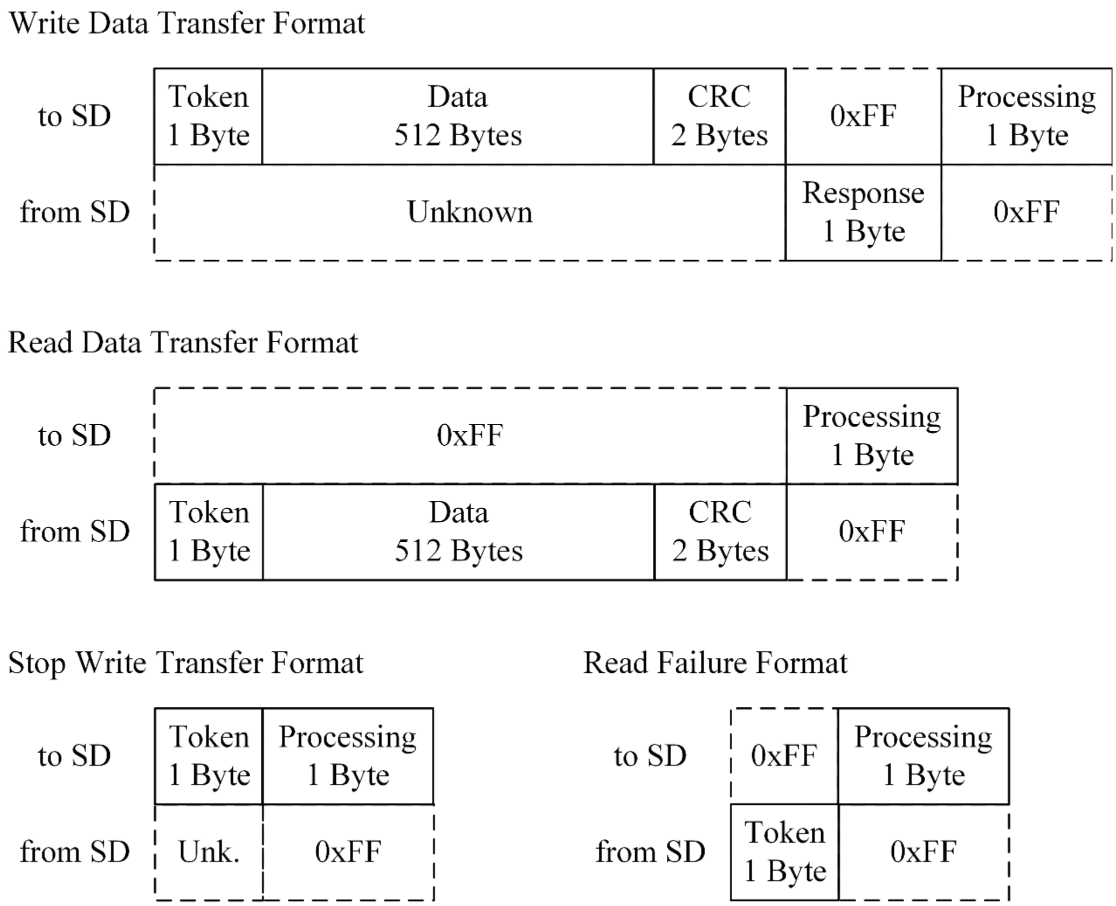

Figure 3.4. SD card simplified data transaction format.

After the necessary communication formats to an SD card were enumerated, the command and data communication methods and formats were designed. Although the most direct method would be to send a command and then query a response from the card following transmission of the command (see Figure 3.5), this would use valuable system processing to request and re-request a card response. Instead, the design utilized a method that would transmit the command and read the response with a single large SPI transaction at the expense of up to eight extra byte transmissions following the card's response (See Figure 3.6). On average, this approach wastes time in the extra transmissions; however, the control scheme is simplified by eliminating the need to request and re-request responses from a card. Although this method was not certain to improve performance, it was likely that response logic would use more system processing cycles overall than would be saved by eliminating extra transmissions.

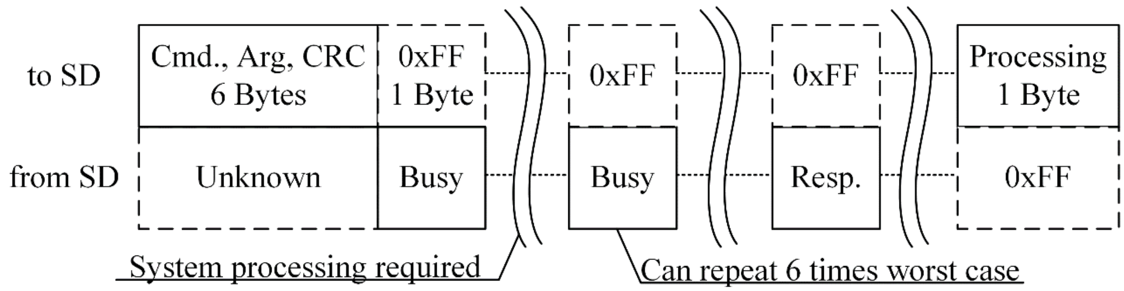

Figure 3.5. SD command transaction using response test and retest method. System processing overhead makes this method slower than ideal.

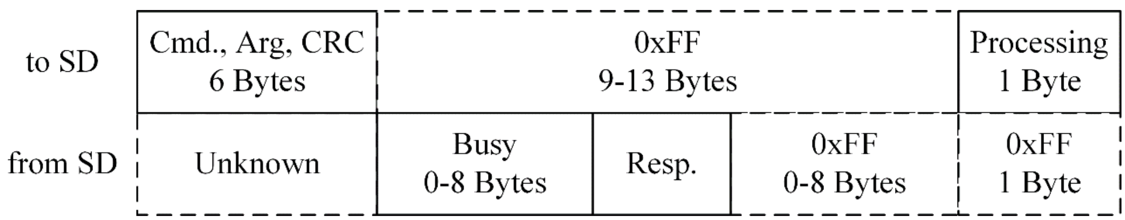

Figure 3.6. SD command transaction using a single large transaction. Up to eight extra bytes are used for each transaction, but system processing overhead is eliminated compared to response test and retest method.

To maximize performance, a hybrid scheme (See Figure 3.7) employing the benefits of both methods was considered, but extensive testing would have been required, and it was uncertain whether the method would be applicable to all microSD cards or even the same card for all operating conditions. This method was therefore abandoned.

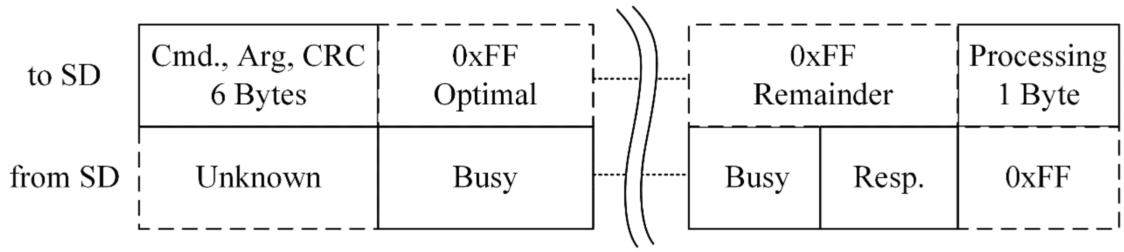

Figure 3.7. SD command transaction using a hybrid method enjoying the benefits of both the response test and retest and the single large transaction methods. Abandoned due to extensive testing required to determine optimal test duration and uncertainty of the testing's universal applicability.

This control method posed a specific challenge for reading data blocks. Consider the scenario posed by Figure 3.8. Also consider the scenario posed by Figure 3.9 which was found to occur much more often (almost exclusively) in testing. The first scenario dictates that data may exist within the command and response transaction. The second scenario dictates that data may not yet be ready in the transaction. Although it would be possible to wait 100 milliseconds before reading data following the second scenario, this would waste precious time waiting when data may be available. Instead a method of requesting and detecting the data token was devised. For unknown reasons, testing showed that a microSD did not respond with a data token until two busy tokens were sent by the card, even when waiting longer than 100 milliseconds to request the data token (See Figure 3.10). This oddity led to the development of a three-byte data token request. Appendix A shows the multi-block read process in its entirety.

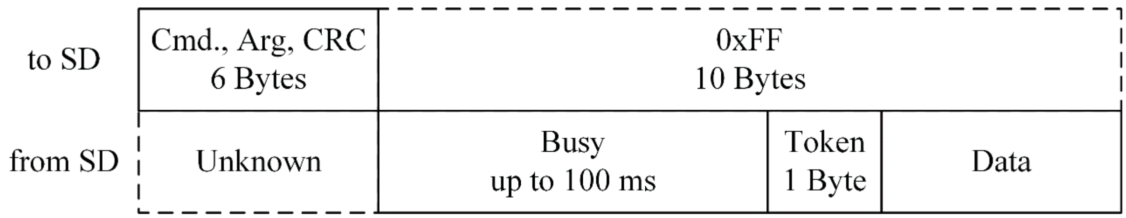

Figure 3.8. Example scenario demonstrating the presence of a data read token and read data in the response following a multi-block read command.

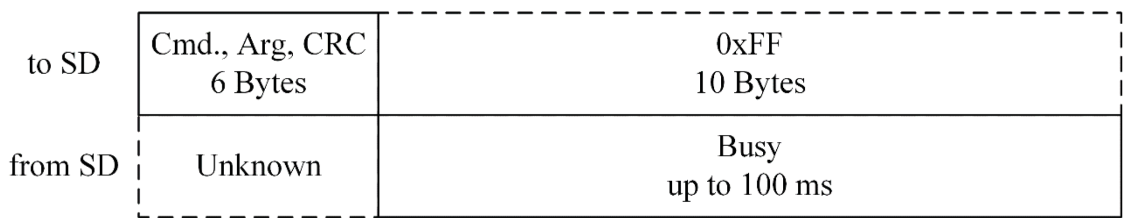

Figure 3.9. Example scenario demonstrating the absence of a data read token in the response following a multi-block read command. This scenario was more prevalent in testing.

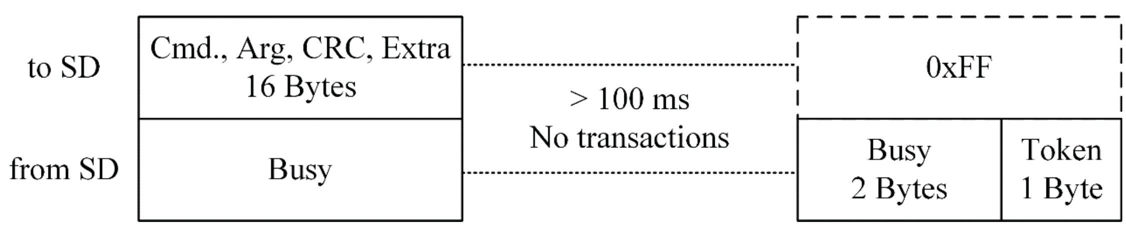

Figure 3.10. Oddity detected during testing of multi-block read process. Even after waiting much longer than the maximum wait time (100 ms), two busy (0xFF) bytes were always sent by the microSD card before the read data token. The cause was unknown.

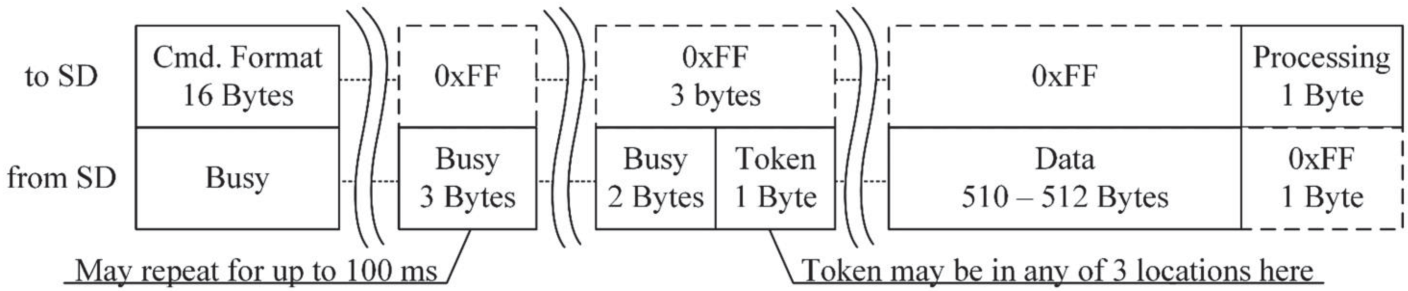

Figure 3.11. Multi-block read timeline.

In development, the command used to halt a multi-block read process demonstrated unexpected behavior. The byte prior to the response byte often appeared corrupted. FatFs's documentation reported that the byte immediately following the command was a "stuff byte" [14]. This information was not corroborated by the SD specification document. The SanDisk product manual offered the critical insight: the busy time after the command was reported to be two to 64 clock cycles [13]. Since all communications were required to be byte aligned, this implied that the first byte would be invalid because the first two bits were invalid. This translates to a busy time of one to eight bytes for this command rather than zero to eight bytes for other commands.

Writing data blocks to the card was a much simpler, following the process shown in Figure 3.12. Although the specification document specifies that the busy time between blocks may be as much as 250 milliseconds, this time was determined to be less than 200 microseconds in preliminary testing. The busy time following a stop transmission token was found to often last several milliseconds.

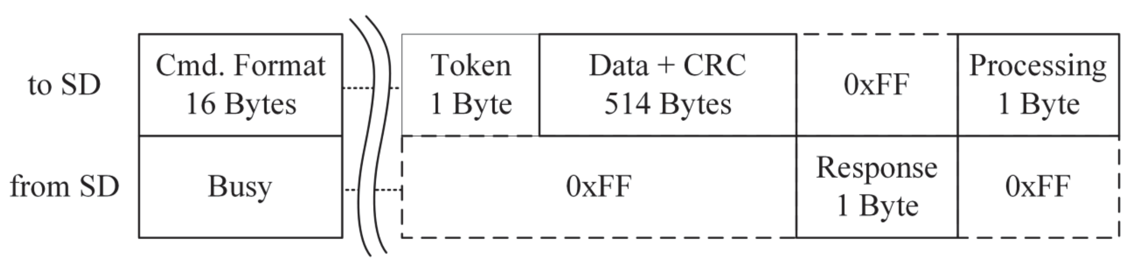

Figure 3.12. Multi-block write timeline.

Detecting the busy state of an SD card is as simple as reading a byte from the card with no data transmitted. If the returned byte is not equal to 0xFF (hexadecimal), the card is non-busy. The "Clock Control" section of the SD specification document states that "the host shall provide a clock edge for the card to turn off its busy signal" [11]. With this statement as inspiration, PNG team members (Brandon Gardner) tried to optimize detection of the busy state by simply toggling the microSD card's clock signal from high to low and then back to high, effectively providing one clock to allow the card to turn off its busy signal.

Although this optimization was sufficient for the card to turn off its busy signal, this caused card errors. These errors were assumed to be due to the violation of the requirement that "every command or data block is built of 8-bit bytes and is byte aligned to the CS signal" [11]. A similar optimization technique was investigated which directly toggled the card's clock signal, providing the eight required clocks without initiating a byte transaction via SPI. The number of system processing cycles used for this method was compared to the number of system processing cycles used for sending a byte via SPI. The former method required approximately 220 cycles, whereas the latter method required approximately 350 cycles. This optimization therefore reduced the number of cycles required by approximately 37 percent.

To successfully log telemetry data during periods where a microSD card was in a busy state, logged data was buffered. The busy state could last up to 250 milliseconds, meaning a buffer size of at least 250 milliseconds was necessary. For a system generating 52,000 bytes of data per second, this equated to a minimum of 13,000 bytes for data buffers. This does not include any overhead needed for transmitting data to a card or for the buffer control logic. This buffer size proved effective in testing, but a larger buffer size of 500 milliseconds or more was preferable to maintain data integrity in the eventuality of extreme corner cases that might not have manifested in testing.

3.2.2 Control State Machines

To properly interface with an SD card, it is necessary to know what commands were sent to the card previously. Certain commands are only valid after certain other commands, and the response given by the card must be put into context given the command history. Based upon this knowledge, it was determined that a state machine control scheme was appropriate. Before any data could be read from or written to the card, it first needed to be initialized based upon the flowchart given in the SD specification document (See Figure 3.13). Version 1.x SD memory cards and non-SD memory cards (left side of chart) support only up to 2 gigabytes of capacity. Because microSD easily supported the ideal 3.5 gigabyte capacity and much greater capacities, the PNG team decided to report initialization failure for these low-capacity cards. Once an SD card is initialized, the initialization procedure is not needed again; therefore, designing the initialization procedure to be reentrant was unnecessary. For initialization, the flowchart neglects to specify that once an SD card receives power, the host is required to send at least 74 clocks to the card with the chip select line held high to enter SPI mode. The flowchart also neglects to mention setting a timeout of "more than [one] second to abort repeat of issuing ACMD41 when the card does not indicate ready" [11].

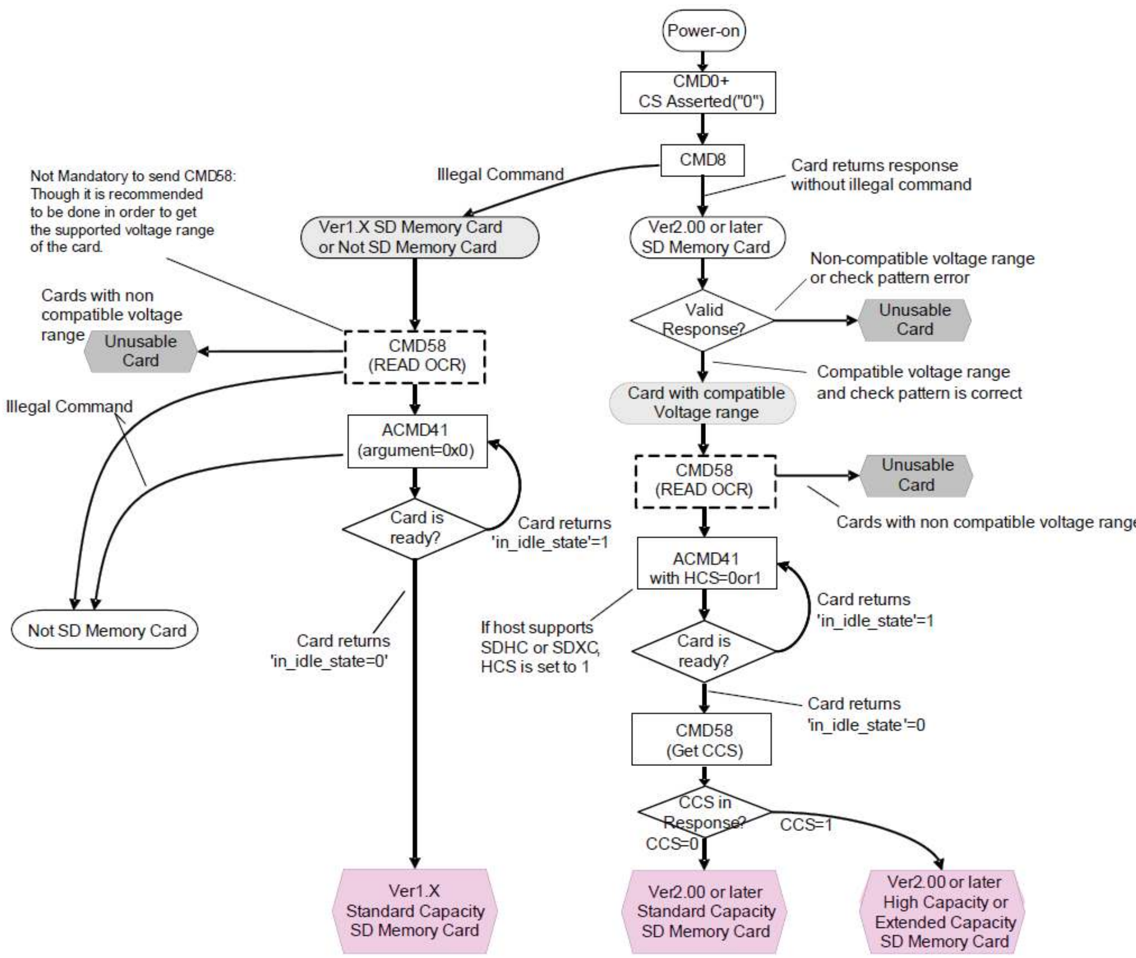

Figure 3.13. SD card initialization flow chart [11].

Once an SD card has been initialized, the clock speed may be raised to a maximum of 25 megahertz. This can result in outstanding read and write speeds; however, the clock frequency may be limited by the hold time of the card (14 nanoseconds max) and the setup time of the device controlling the card [13], [14]. The maximum clock frequency follows the following equation.

$$f_{CLK(\max)} = \frac{1}{2 \cdot (14\,ns + t_{su})} \quad , t_{su} \text{ is the minimum setup time of the host's MISO pin}$$

State machines for reading and writing multiple blocks were not provided in documentation and had to be designed. The commands used to initiate a read or write process, the data transmission and reception methods, and the method used to finish read and write processes are each entirely distinct. This led to the creation of two distinct state machines, one for reading multiple data blocks and one for writing multiple data blocks.

The multi-block write process was designed first and was reduced iteratively to only three states. The multi-block read process was designed second and was also reduced to three states. The external-facing interface to the state machines were designed for flexibility rather than ease of use, allowing an external program to have explicit control of when the SD card should be prepared for reading or writing, when the card should transmit or receive data blocks, and when the card should finish reading or writing. The state machines are shown in overview in Figure 3.14 and Figure 3.15. The state machines are shown in full detail in appendix A.

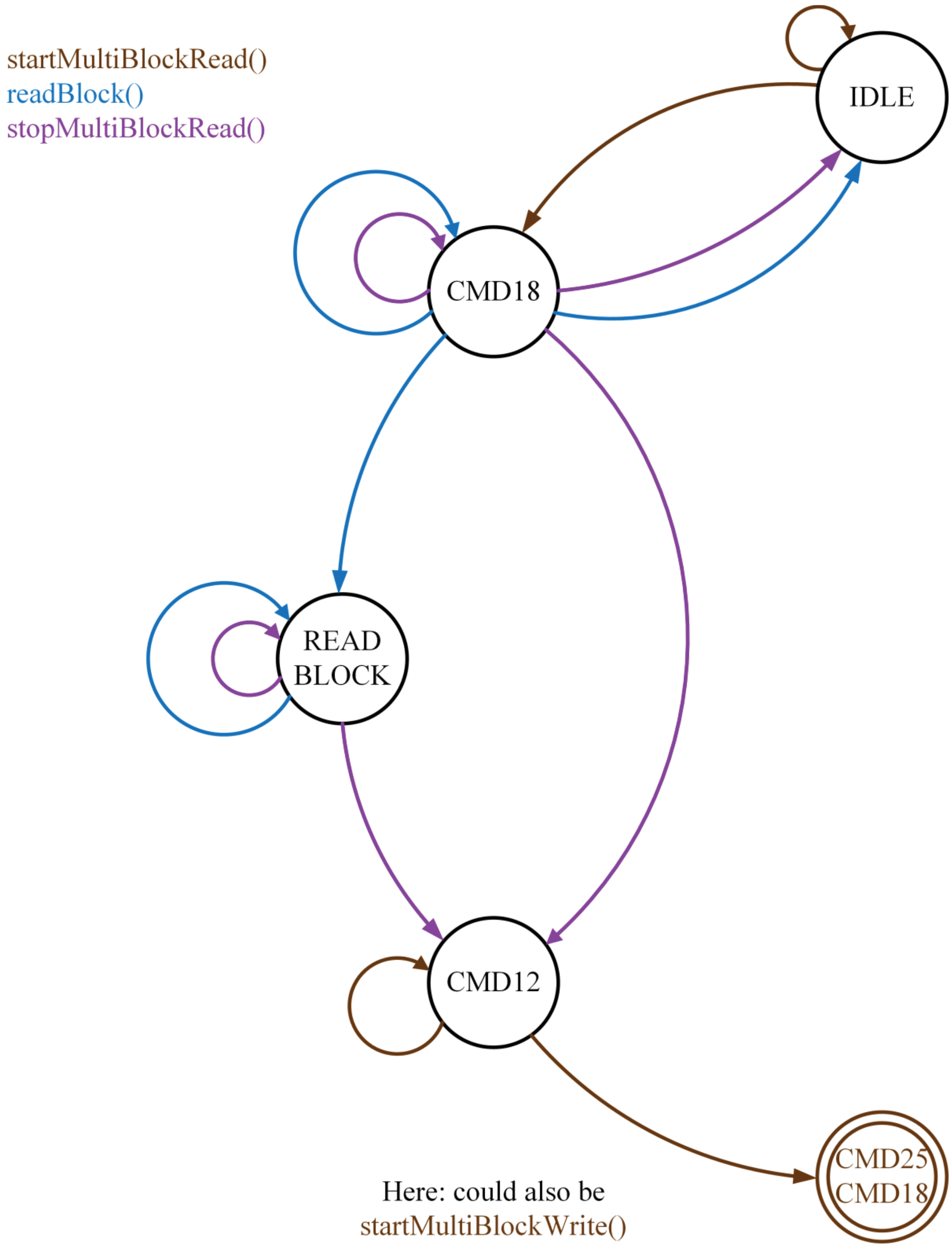

Figure 3.14. Overview of multi-block read state machine. Each arrow color represents a state change as a result of a corresponding function call. Function calls are listed in the top-left.

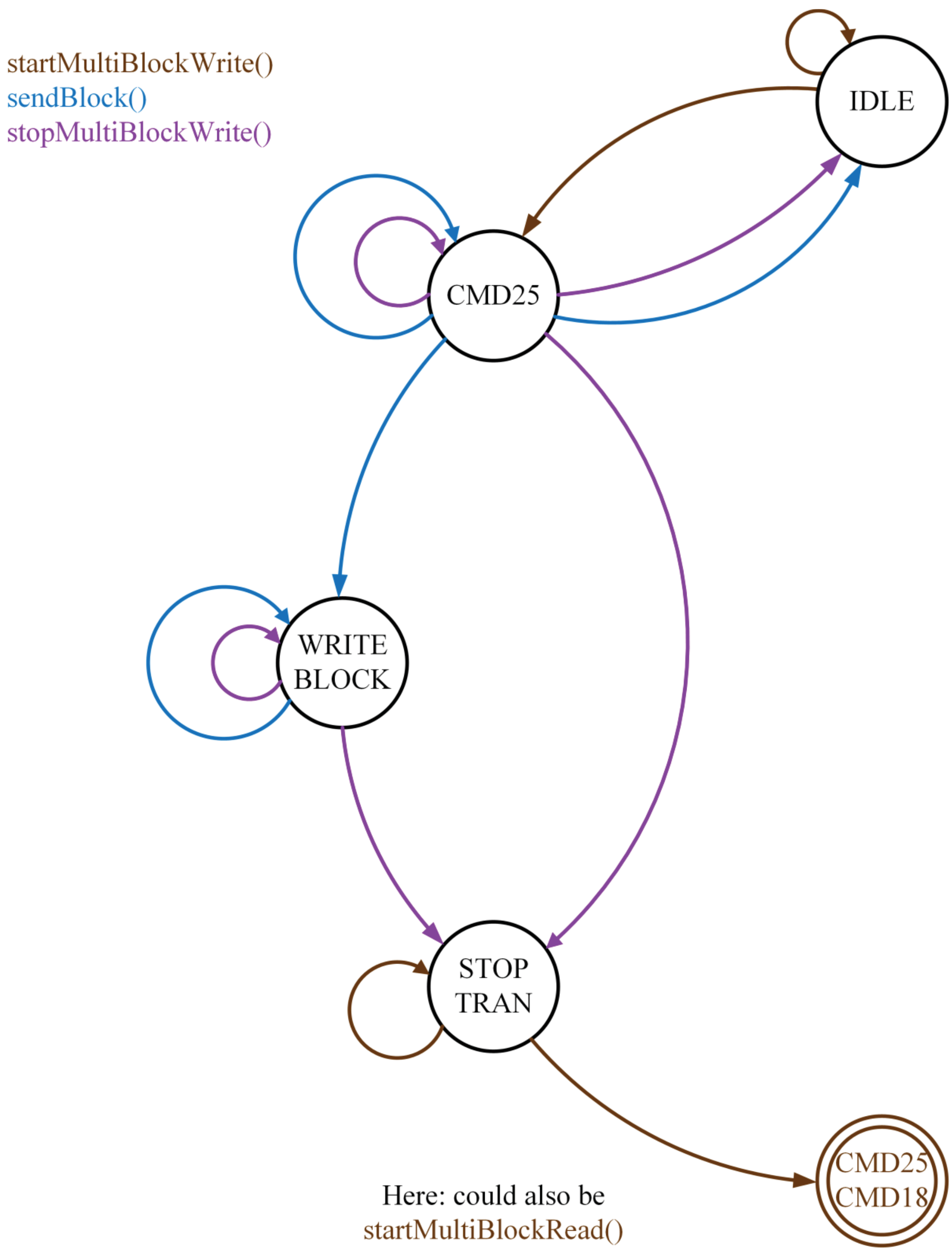

Figure 3.15. Overview of multi-block write state machine. Each arrow color represents a state change as a result of a corresponding function call. Function calls are listed in the top-left.

3.3 Taking Advantage of the Prevalence of microSD

One of the additional benefits microSD has is the fact that it can be accessed easily via a computer. In testing, this meant that it was possible to easily verify the data read from or written to the card. For real world data logging, this meant that it was possible to retrieve telemetry data without the need to implement an interface to a computer. To easily access the data from a computer, a file system was needed on the card. Access without a file system, although possible, was limited to standalone applications. With a file system, any application with basic file capabilities could access the files.

Although there are many methods of implementing a file system on the card in such a way that both the data logging device and computer could access the data, the method implemented was designed in such a manner that the telemetry device could be unaware of the existence of a file system save for the requirement that it avoid writing to microSD card blocks belonging to the master boot record and file system structures. These structures would ideally be located at the beginning of the disk so that all blocks after them would be available.

First, a card was formatted using the SD Association's SDFormatter utility. Next, the card was filled up by copying (one file at a time) as many text files to the card as would fit. These files were given a file name on the card indicative of the order in which they were copied, e.g., from 1 to N. A large file size allows the volume to have a smaller file directory, found in testing to be initially 1023 entries. To prevent the directory from growing beyond its initial size, a good rule of thumb was found to be: use a file size at least as many megabytes as the listed card capacity in gigabytes (e.g., a 16 megabyte or greater files were used with a 16 gigabyte card). It was also necessary for the file to be a multiple of 32 kilobytes[1], as this is the maximum allocation unit size for FAT32.

Following these guidelines, each of the files were consecutive on disk for a number of microSD cards tested. This means that file K was preceded by file K-1 and was followed

---

[1] One kilobyte equals 1024 bytes

by file K+1 for all files. More importantly, the files were contiguous, with no unused space between them. For cards between 4 gigabytes and 16 gigabytes, the first file's data began at block number 16448; this number was different for 2 gigabyte and 32 gigabyte cards. Telemetry data written to the card began at block 16448 and continued incrementally as logging progressed. After data collection, the telemetry data was able to be accessed by simply reading the files on the card. Data from the first file was retrieved first, followed by data from the second file, followed by the third, etc. Because the files were consecutive on disk, they were ordered chronologically, and because the files were contiguous, no data was lost.

This method allowed for easy retrieval of the logged data without losses and with a minimal set up effort. The microSD card setup process was easily automated using standard scripting tools. This method also provided the added benefit of being able to simply swap used cards with fresh cards in telemetry units when PNG members (Tom Talavage, Eric Nauman) were ready to collect the telemetry data rather than needing to connect each device to a computer for downloading.

4. PERFORMANCE ANALYSIS

Purdue Neurotrauma Group members (Brandon Gardner) tested the base performance of the implementation by measuring the time taken to read or write 32 megabytes from/to a small assortment of microSD cards of different brands, capacities, and speed class ratings. This measurement was taken while the system was idle, performing only the logic necessary to control the microSD cards. The timing was measured using an oscilloscope and was determined to be accurate to within 0.2 seconds.

Four variables could influence the read/write speed of the system: (1) manufacturer, (2) capacity, (3) speed class, and (4) silicon variance[1]. Cursorily measuring the first three variables was trivial. Determining the effects of silicon variance was much more difficult and required testing a large, statistically valid collection of cards. PNG did not have the resources to undertake this testing. This also means that the results found are not statistically valid; this testing was intended only to suggest trends that might be found.

Three common, name-brand manufacturers were chosen for testing: SanDisk, Kingston, and Transcend. Class four microSD cards were used to test for capacity variance, as it was easier to find cards in a wider variety of capacities than other classes. Exceptions to this include the two gigabyte card, which had no class rating and the 32 gigabyte card which had a class ten rating. The effects of speed class were tested using 16 gigabyte class four and class ten cards from both SanDisk and Transcend, as a 16 gigabyte class 4 Kingston card could not be found.

The system used for testing was an MSP430F5659 microcontroller with a main clock frequency of 20 megahertz and an SPI bus clock frequency of 12 megahertz. The cards

---

[1] Silicon variance refers to the variance between microSD cards due to silicon manufacturing tolerances.

were prepared with random data to be read, and blocks of 0xFF were written to the card to prevent the write operations from being a simple erase of each block. Table 4.1 summarizes the results.

Table 4.1. Experimental time taken to read or write 32 megabytes of data from/to the described microSD card. Speed was calculated as the number of bytes read/written divided by the time elapsed. Time accuracy was 0.2 seconds.

Card Description	Read Time	Write Time	Read Speed	Write Speed
Transcend 16 GB class 4	29 sec	26.8 sec	1.1 MBps	1.19 MBps
Transcend 16 GB class 10	29 sec	27 sec	1.1 MBps	1.18 MBps
Kingston 16 GB class 10	29 sec	27 sec	1.1 MBps	1.18 MBps
SanDisk 16 GB class 10	29 sec	28.4 sec	1.1 MBps	1.12 MBps
SanDisk 8 GB class 4	29 sec	26.6 sec	1.1 MBps	1.20 MBps
SanDisk 4 GB class 4	29 sec	27 sec	1.1 MBps	1.18 MBps
SanDisk 32 GB class 10	29 sec	28 sec	1.1 MBps	1.14 MBps
SanDisk 2 GB no class	29 sec	26.6 sec	1.1 MBps	1.20 MBps
SanDisk 16 GB class 4	29 sec	26.8 sec	1.1 MBps	1.19 MBps

The results obtained were unexpected. There was not a significant difference between any of the microSD cards tested. Additionally, it was expected that write times would be slower than read times, which was not the case in these tests. The theoretical max read/write speed for the test system was 1.43 megabytes per second (MBps[1]); the efficiency achieved was therefore approximately 77 percent. From these results, it is likely that the primary factor determining the read/write speed was the software implementation rather than limitations of the individual cards. If the system used for testing were able to use a 25 megahertz clock speed, it is likely that the performance differences between the cards would be more apparent. No conclusions can be drawn from these results.

Figure 4.1 shows the read/write speed of two SD cards and one multimedia card (MMC) tested by the developer of FatFs. These results are useful at a glance, but the test methods are largely unknown except for the system used, an LPC2368 microcontroller running at 72 megahertz with an SPI bus speed of 18 megahertz. The results shown reflect the expectation that read speed is faster than write speed. The efficiencies compared to the bus speed for the eight gigabyte Kingston card tested are 62 percent for writing and 84 percent for reading. A bus speed of 9000 kilobytes per second, however, implies one of two possibilities: (1) the bus speed was closer to 72 megahertz or (2) the bus provided four bits of data rather than a single bit at each clock edge. The latter possibility is more likely since the maximum clock speed for SD is 25 megahertz or 50 megahertz for some cards [11]. It is therefore inappropriate to compare these results to PNG results, but they are included due to the fact that no other SD performance tests were found elsewhere.

---

[1] MBps = megabytes per second = 1048576 bytes per second

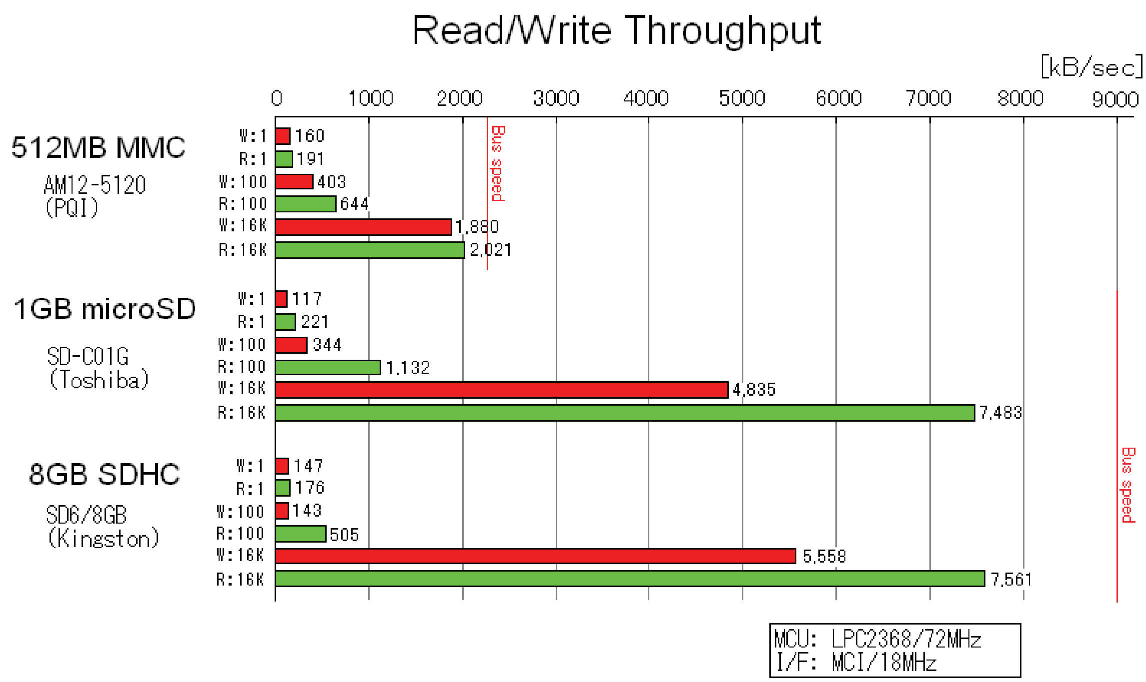

Figure 4.1. Experimental results obtained by the developer of FatFs [15]. A 9000 kB/sec bus speed is impossible unless four bits are written on each clock edge versus one; therefore, these results cannot be directly compared to Purdue Neurotrauma Group results.

5. CONCLUSION

This thesis has presented a simple software design for an embedded data logging system using a microSD card as the memory technology. This design's primary benefit is a large memory capacity that continues to grow as larger and larger capacity microSD cards enter the market [16]. MicroSD's wide acceptance also means a low cost per byte ratio, and for its capacity, it is remarkably small and power efficient.

This has allowed Purdue Neurotrauma Group members(Aditya Balasubramanian, Jeff King, Brandon Gardner) to develop prototype continuous-time, biomechanical telemetry sensors for their study of the effects of head impacts—especially sub-concussive impacts—on neurophysiology. The results of this effort have given the PNG (Tom Talavage, Eric Nauman) access to a quality of data that will allow them to develop predictive models of brain damage, a key component for enabling early detection of permanent threats to brain health.

5.1 Future Work

The design presented by this thesis was considered successful in meeting the Purdue Neurotrauma Group's needs; however, room for further optimization exists. As mentioned in 3.1, real-time operating system (RTOS) functionality was not considered for this design. A RTOS could enable the system to be more extensible for future needs both anticipated and unanticipated. With RTOS support, the reentrant loop-based technique might be a more appropriate communication method. The hybrid command scheme could also be tested to determine if a shorter, optimal response time exists. Additionally, a number of minor optimizations within the SD control code have been identified but not yet implemented or tested; implementing these optimizations could save tens of instruction cycles, releasing more cycles to the main system.

LIST OF REFERENCES

LIST OF REFERENCES

[1] A. Schwarz, "Dementia Risk Seen in Players in N.F.L. Study," *The New York Times*, 29-Sep-2009.

[2] E. L. Breedlove, M. Robinson, T. M. Talavage, K. E. Morigaki, U. Yoruk, K. O'Keefe, J. King, L. J. Leverenz, J. W. Gilger, and E. A. Nauman, "Biomechanical correlates of symptomatic and asymptomatic neurophysiological impairment in high school football," *J. Biomech.*, vol. 45, no. 7, pp. 1265–1272, Apr. 2012.

[3] T. M. Talavage, E. A. Nauman, E. L. Breedlove, U. Yoruk, A. E. Dye, K. E. Morigaki, H. Feuer, and L. J. Leverenz, "Functionally-Detected Cognitive Impairment in High School Football Players without Clinically-Diagnosed Concussion," *J. Neurotrauma*, vol. 31, no. 4, pp. 327–338, Feb. 2014.

[4] "Simbex: HIT System Research." [Online]. Available: http://www.simbex.com/hit-system1.html. [Accessed: 12-Mar-2014].

[5] "Purdue University - Purdue Neurotrauma Group," *Purdue Neurotrauma Group*. [Online]. Available: http://www.purdue.edu/research/png/. [Accessed: 12-Mar-2014].

[6] R. Jadischke, D. C. Viano, N. Dau, A. I. King, and J. McCarthy, "On the accuracy of the Head Impact Telemetry (HIT) System used in football helmets," *J. Biomech.*, vol. 46, no. 13, pp. 2310–2315, Sep. 2013.

[7] A. Balasubramanian, "DEVELOPING A HARDWARE PLATFORM FOR A LOW-POWER, LOW-COST, SIZE-CONSTRAINED BIOMECHANICAL TELEMETRY SYSTEM," Master's, Purdue University, West Lafayette, Indiana, 2014.

[8] O. C. Ibe, *Fundamentals of applied probability and random processes*. Burlington, MA; London: Elsevier Academic Press, 2005.

[9] "FatFs - Generic FAT File System Module," *The Electronic Lives Manufacturing - presented by ChaN*. [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html. [Accessed: 13-Mar-2014].

[10] "Code Composer Studio (CCStudio) Integrated Development Environment (IDE) v5 - CCSTUDIO - TI Tool Folder," *Texas Instruments*. [Online]. Available: http://www.ti.com/tool/ccstudio#Technical Documents. [Accessed: 12-Mar-2014].

[11] SD Card Association, *SD Specifications Part 1 - Physical Layer Simplified Specification Version 4.10*. 2013.

[12] Nick Lethaby, “Why Use a Real-Time Operating System in MCU Applications.” 2013.

[13] SanDisk Corporation, “Sandisk SD Card Product Manual Version 2.2 - Document No. 80-13-00169.” Nov-2004.

[14] “How to Use MMC/SDC,” *The Electronic Lives Manufacturing - presented by ChaN*. [Online]. Available: http://elm-chan.org/docs/mmc/mmc_e.html. [Accessed: 12-Mar-2014].

[15] “Read/Write Test 2,” *The Electronic Lives Manufacturing - presented by ChaN*. [Online]. Available: http://elm-chan.org/fsw/ff/img/rwtest2.png. [Accessed: 12-Mar-2014].

[16] SanDisk Corporation, “SANDISK INTRODUCES WORLD’S HIGHEST CAPACITY microSDXC MEMORY CARD AT 128GB,” *SanDisk*, 24-Feb-2014. [Online]. Available: http://www.sandisk.com/about-sandisk/press-room/press-releases/2014/sandisk-introduces-worlds-highest-capacity-microsdxc-memory-card-at-128gb/. [Accessed: 20-Mar-2014].

APPENDICES

A. SD CONTROL STATE MACHINES

Notes:

1) The IDLE state is shared between each state machine
2) If a state transition is not shown, it can be assumed to be an invalid command

MULTI-BLOCK READ STATE TRANSITIONS

1. startMultiBlockRead(): IDLE → IDLE
   a. Send CMD18 failure → Return: SPI error
2. startMultiBlockRead(): IDLE → CMD18
   a. Send CMD18 success → Return: OK
3. readBlock(): CMD18 → CMD18
   a. SPI busy → Return: Busy
   b. SPI not busy
      i. R1 response = 0
         1. Data token = 0xFE
            a. Read block failure → Return: SPI error
         2. Data token not present
            a. Request data token failure → Return: SPI error
4. stopMultiBlockRead(): CMD18 → CMD18
   a. SPI busy → Return: Busy
   b. SPI not busy
      i. R1 response = 0
         1. Send CMD12 failure → Return: SPI error
5. stopMultiBlockRead(): CMD18 → IDLE
   a. SPI not busy
      i. R1 response ≠ 0 → Return: Restart multi-block read
      ii. R1 response = 0
         1. Data token = error → Return: Restart multi-block read
6. readBlock(): CMD18 → IDLE
   a. SPI not busy
      i. R1 response ≠ 0 → Return: Restart multi-block read
      ii. R1 response = 0
         1. Data token = error → Return: Restart multi-block read
7. readBlock(): CMD18 → READ_BLOCK
   a. SPI not busy
      i. R1 response = 0
         1. Data token = 0xFE

            a. Read block success → Return: OK
         2. Data token not present
            a. Request data token success → Return: Busy

8. stopMultiBlockRead(): CMD18 → CMD12
   a. SPI not busy
      i. R1 response = 0
         1. Send CMD12 success → Return: OK
9. readBlock(): READ_BLOCK → READ_BLOCK
   a. SPI busy → Return: Busy
   b. SPI not busy
      i. Data token = 0xFE
         1. Read block success → Return: OK
         2. Read block failure → Return: SPI error
      ii. Data token not present
         1. Request data token success → Return: Busy
         2. Request data token failure → Return: SPI error
10. stopMultiBlockRead(): READ_BLOCK → READ_BLOCK
    a. SPI busy → Return: Busy
    b. SPI not busy
       i. Data token = 0xFE OR not present
          1. Send CMD12 failure → Return: SPI error
11. stopMultiBlockRead(): READ_BLOCK → CMD12
    a. SPI not busy
       i. Data token = 0xFE
          1. Send CMD12 success → Return: OK
       ii. Data token = error
          1. Send CMD12 success → Return: Continue with error
12. startMultiBlockRead() OR startMultiBlockWrite(): CMD12 → CMD12
    a. SPI busy → Return: Busy
    b. SPI not busy
       i. SD card busy → Return: Busy
       ii. SD card not busy
          1. R1 response = 0
             a. Send CMD18/CMD25 failure → Return: SPI error
          2. R1 response ≠ 0 → Return: Send stop multi-block read
13. startMultiBlockRead() OR startMultiBlockWrite(): CMD12 → CMD18/CMD24
    a. SPI not busy
       i. SD card not busy
          1. R1 response = 0
             a. Send CMD18/CMD25 success → Return: OK

MULTI-BLOCK WRITE STATE TRANSITIONS

14. startMultiBlockWrite(): IDLE → IDLE
    a. Send CMD25 failure → Return: SPI error
15. startMultiBlockWrite(): IDLE → CMD25
    a. Send CMD25 success → Return: OK
16. sendBlock(): CMD25 → CMD25
    a. SPI busy → Return: Busy
    b. SPI not busy
        i. R1 response = 0
            1. Write block failure → Return: SPI error
17. stopMultiBlockWrite(): CMD25 → CMD25
    a. SPI busy → Return: Busy
    b. SPI not busy
        i. R1 response = 0
            1. Send stop transmission token failure → Return: SPI error
18. stopMultiBlockWrite(): CMD25 → IDLE
    a. SPI not busy
        i. R1 response ≠ 0 → Return: Restart multi-block write
19. sendBlock(): CMD25 → IDLE
    a. SPI not busy
        i. R1 response ≠ 0 → Return: Restart multi-block write
20. sendBlock(): CMD25 → WRITE_BLOCK
    a. SPI not busy
        i. R1 response = 0
            1. Write block success → Return: OK
21. stopMultiBlockWrite(): CMD25 → STOP_TRAN
    a. SPI not busy
        i. R1 response = 0
            1. Send stop transmission token success → Return: OK
22. sendBlock(): WRITE_BLOCK → WRITE_BLOCK
    a. SPI busy → Return: Busy
    b. SPI not busy
        i. SD card busy → Return: Busy
        ii. SD card not busy
            1. Data response = 0x05
                a. Write block failure → Return: SPI error
                b. Write block success → Return: OK
            2. Data response ≠ 0x05 → Return: Send stop multi-block write
23. stopMultiBlockWrite(): WRITE_BLOCK → WRITE_BLOCK
    a. SPI busy → Return: Busy
    b. SPI not busy
        i. SD card busy → Return: Busy
        ii. SD card not busy
            1. Data response = 0x05 OR ≠ 0x05

            a. Send stop transmission token failure → Return: SPI error

24. stopMultiBlockWrite(): WRITE_BLOCK → STOP_TRAN
    a. SPI not busy
        i. SD card not busy
            1. Data response = 0x05
                a. Send stop transmission token success → Return: OK
            2. Data response ≠ 0x05
25. startMultiBlockRead() OR startMultiBlockWrite(): STOP_TRAN → STOP_TRAN
    a. SPI busy → Return: Busy
    b. SPI not busy
        i. SD card busy → Return: Busy
        ii. SD card not busy
            a. Send CMD18/CMD25 failure → Return: SPI error
26. startMultiBlockRead() OR startMultiBlockWrite(): STOP_TRAN → CMD18/CMD25
    a. SPI not busy
        i. SD card not busy
            a. Send CMD18/CMD25 success → Return: OK

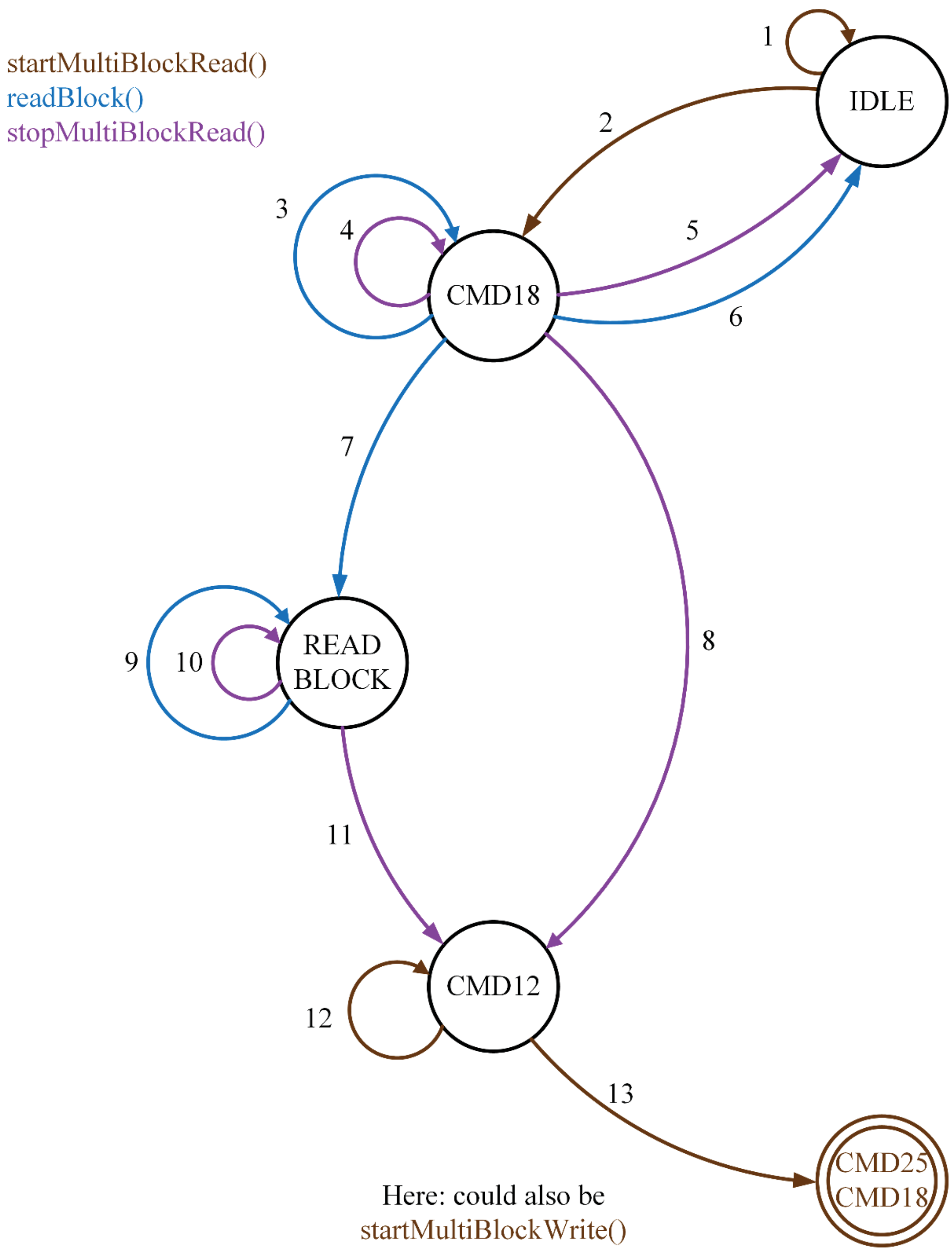

Figure A.1. Multi-block read state machine annotated with numbered sections.

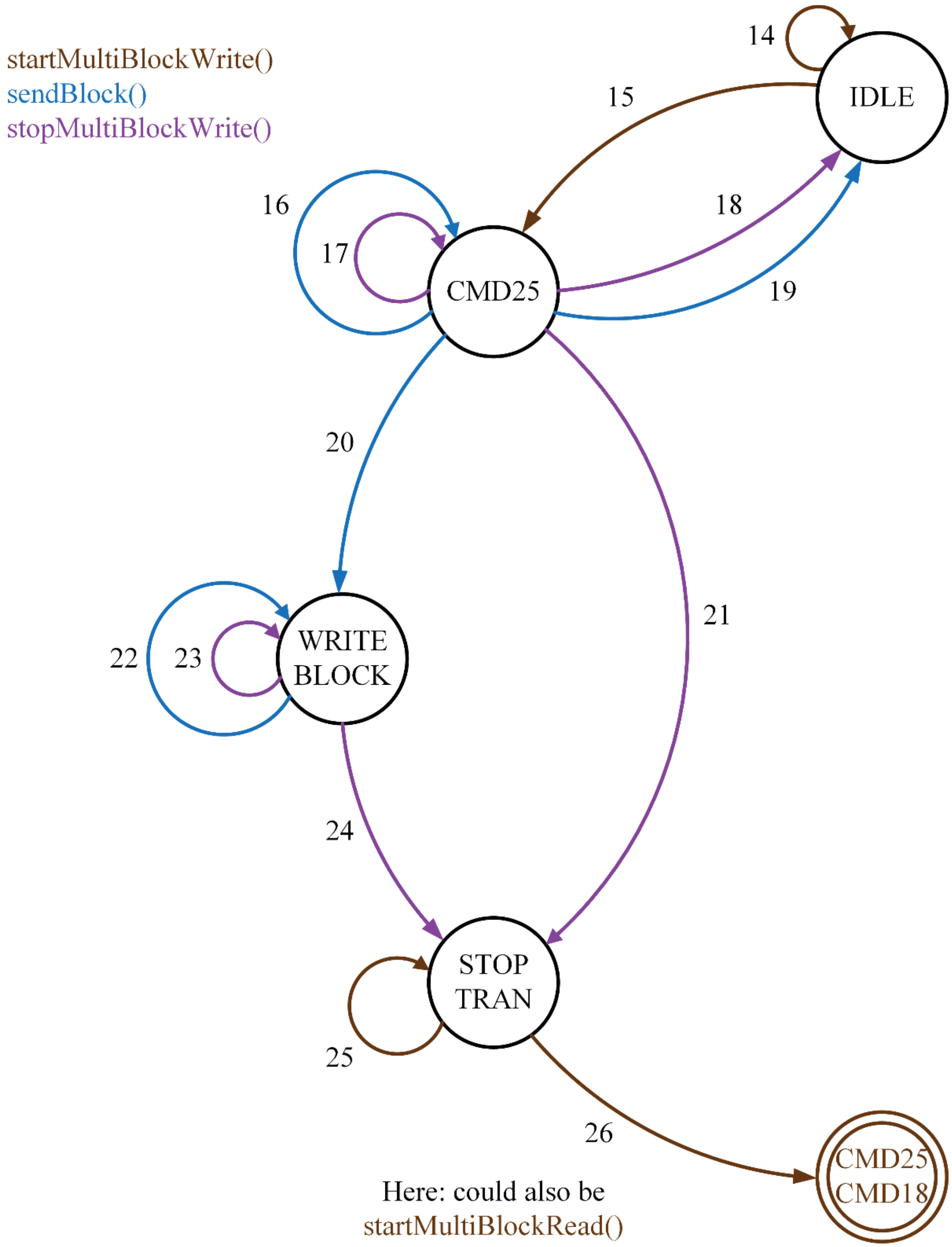

Figure A.2. Multi-block write state machine annotated with numbered sections.

B. CODE

```
// note(s):
// 1. Page and block are used interchangeably to refer to a 512-byte SD
data block
// 2. Some structures exist for single block reads/writes but are
unfinished

//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
// HELPER FUNCTION AND MACRO DEFINITIONS
//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

typedef enum {
    SD_CONTINUE = 0,                    ///< Continue to next operation
    SD_CONTINUE_WITH_ERR,               ///< Continue to next operation w/
error from previous state
    SD_BUSY_RETRY,                      ///< Card or process is busy, retry
(long timeout)
    SD_ERR_RETRY,                       ///< Error, retry (with timeout)
    SD_ERR_RESTART,                     ///< Error, restart from command
beginning (i.e. - CMD17/18/24/25)
    SD_ERR_SEND_STOP,                   ///< Error, send stop (suggested to
restart from cmd beginning)
    SD_INVALID_CMD,                     ///< Invalid command
    SD_ERR                              ///< Error, unknown
} SD_TYPE;

/*
 * @brief Initialize SPI for SD operation at a given speed.
 * @param c [in]    Desired SPI clock speed.
 *
 * @note \see dma_spi_usci_initialize() for more info.
 */
#define spi_initialize( c )

/*
 * @brief Perform SPI transaction for SD operation.
 * @param t [in]        TX buffer.
 * @param r [out]       RX buffer.
 * @param b [in]        Byte count.
 * @param d [in/out]    Transaction complete flag pointer.
 *                      Is incremented upon completion of SPI
transaction.
 */
#define spi_transaction(t,r,b,d)

/*
```

```
 * @brief Initialize pins for SPI communication to SD card.
 *
 * @note CLK is pulled low with internal resistor.
 * @note SIMO is pulled high with internal resistor.
 * @note SOMI is pulled high with internal resistor.
 */
inline void init_pins() {
    // CLK
    SEL_FXN( SD_CLK_SEL, SD_CLK_BIT );
    DIR_OUT( SD_CLK_DIR, SD_CLK_BIT );
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    REN_ENABLE( SD_CLK_REN, SD_CLK_BIT );
    // SIMO
    SEL_FXN( SD_SIMO_SEL, SD_SIMO_BIT );
    DIR_OUT( SD_SIMO_DIR, SD_SIMO_BIT );
    OUT_HIGH( SD_SIMO_OUT, SD_SIMO_BIT );
    REN_ENABLE( SD_SIMO_REN, SD_SIMO_BIT );
    // SOMI
    SEL_FXN( SD_SOMI_SEL, SD_SOMI_BIT );
    DIR_IN( SD_SOMI_DIR, SD_SOMI_BIT );
    OUT_HIGH( SD_SOMI_DIR, SD_SOMI_BIT );
    REN_ENABLE( SD_SOMI_REN, SD_SOMI_BIT );
}

// chip select controls
#define CS_SELECT           OUT_LOW( SD_CS_OUT, SD_CS_BIT )
#define CS_DESELECT         OUT_HIGH( SD_CS_OUT, SD_CS_BIT )
#define CS_INIT()                                \
    SEL_GPIO( SD_CS_SEL, SD_CS_BIT );        \
    DIR_OUT( SD_CS_DIR, SD_CS_BIT );         \
    REN_DISABLE( SD_CS_REN, SD_CS_BIT )

// card on/off controls
#if defined( SD_ONOFF_ACTIVE_HIGH )
#  define SD_ON             OUT_HIGH( SD_ONOFF_OUT, SD_ONOFF_BIT );
#  define SD_OFF            OUT_LOW( SD_ONOFF_OUT, SD_ONOFF_BIT );
#elif defined( SD_ONOFF_ACTIVE_LOW )
#  define SD_ON             OUT_LOW( SD_ONOFF_OUT, SD_ONOFF_BIT );
#  define SD_OFF            OUT_HIGH( SD_ONOFF_OUT, SD_ONOFF_BIT );
#else
#  error "SD card on/off control is defined neither as active high nor
active low."
#endif
// initialize to OFF
#define SD_ONOFF_INIT()                              \
    SEL_GPIO( SD_ONOFF_SEL, SD_ONOFF_BIT );      \
    DIR_OUT( SD_ONOFF_DIR, SD_ONOFF_BIT );       \
    REN_DISABLE( SD_ONOFF_REN, SD_ONOFF_BIT);    \
    SD_OFF

// use in if/while stmt. to test whether SD card is busy
#define SD_NOT_BUSY                 sd_isReady()

// Returns 0 if card not detected, 1 if card is detected
/// @todo replace with read of Chip Detect
```

```
#define CARD_DETECT                   (1)

// card response processing can be 0-8 bytes,
// R1 resp is 1 byte,
// plus one extra byte for processing
// total :  10 bytes
#define R1_TEST_LENGTH                10

// R3 = R1 + 4 byte OCR
#define R3_TEST_LENGTH                R1_TEST_LENGTH+4

// number of bytes to receive to test for data token presence
#define DATA_TOKEN_TEST_LENGTH        3

// Definitions for MMC/SDC command
#define CMD0    (0)           // GO_IDLE_STATE
#define ACMD41  (0x80|41)     // SEND_OP_COND (SDC)  (precede w/ CMD55)
#define CMD8    (8)           // SEND_IF_COND
#define CMD12   (12)          // STOP_TRANSMISSION
#define CMD17   (17)          // READ_SINGLE_BLOCK
#define CMD18   (18)          // READ_MULTIPLE_BLOCK
#define CMD24   (24)          // WRITE_BLOCK
#define CMD25   (25)          // WRITE_MULTIPLE_BLOCK
#define CMD55   (55)          // APP_CMD
#define CMD58   (58)          // READ_OCR

// All states for single-/multi-block read/write
/// @todo Add READ_SINGLE_BLOCK_DONE state and logic
typedef enum {
    IDLE = 0,             //< IDLE
    SEND_CMD25,           //< Start multi-block write
    WRITE_MULTI_BLOCK,    //< Write block for multi-block write
    STOP_TRAN,            //< Stop multi-block write
    SEND_CMD24,           //< Start single-block write
    WRITE_SINGLE_BLOCK,   //< Write block for single-block write
    SEND_CMD18,           //< Start multi-block read
    READ_MULTI_BLOCK,     //< Read block for multi-block read
    SEND_CMD12,           //< Stop multi-block read
    SEND_CMD17,           //< Start single-block read
    READ_SINGLE_BLOCK     //< Read block for single-block read
} sd_state;

// current state
static sd_state process_state = IDLE;

// spi transaction flag
static volatile unsigned char cmd_flag = 0;

// used to determine state of a command (i.e. - sendCommand())
static unsigned char command_state = 0x00;
// 0: free
// 1: sent/sending command
// 2: requested/ing response

// SD card response is placed into this buffer
```

```
static volatile unsigned char response[R3_TEST_LENGTH+6] =
        { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };

// this buffer holds an SD card command
static unsigned char command_buffer[R3_TEST_LENGTH+6] =
        { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };

// this buffer holds the stop tran token and processing byte
static const unsigned char stop_tran_buffer[2] = { 0xfd, 0xff };

// this buffer is used for receiving the data token (potentially w/
data)
// when testing for data token presence
static volatile unsigned char data_token_buffer[DATA_TOKEN_TEST_LENGTH]
= { 0xff, 0xff, 0xff };

// this buffer is sent to the SD card during a page/block read
static const unsigned char page_request_buffer[515] =
    {
        //0xff,
// data token
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 0-15
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 16-31
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 32-47
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 48-63
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 64-79
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 80-95
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 96-111
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 112-127
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 128-143
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 144-159
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 160-175
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 176-191
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 192-207
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 208-223
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 224-239
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 240-255
```

```
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 256-271
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 272-287
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 288-303
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 304-319
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 320-335
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 336-351
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 352-367
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 368-383
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 384-399
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 400-415
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 416-431
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 432-447
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 448-463
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 464-479
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 480-495
        0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff, 0xff, 0xff, 0xff, 0xff, 0xff, // data 496-511
        0xff, 0xff,
// crc
        0xff
// processing byte
    };

// buffer into which received data from SD card is placed for write
block
static volatile unsigned char page_response_buffer[517];

// number of bytes read and transferred to the read buffer
static signed char bytes_read = 0;

// is the card addressed by byte or block?
bool sd_block_addressing = false;

// The code to send a single byte via DMA-SPI uses about 300-350 cycles
just
//   to set up the transfer.
// The below code uses about 220 cycles.
inline char sd_isReady( void ) {
    SEL_GPIO( SD_CLK_SEL, SD_CLK_BIT );
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
```

```
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_HIGH( SD_CLK_OUT, SD_CLK_BIT );
    _nop();
    OUT_LOW( SD_CLK_OUT, SD_CLK_BIT );
    SEL_FXN( SD_CLK_SEL, SD_CLK_BIT );
    return IN_READ( SD_SOMI_IN, SD_SOMI_BIT );
}

inline RETURN_TYPE sendEmptyByte( volatile unsigned char *
command_done_flag ) {
    return spi_transaction( (unsigned char *)( &command_buffer[6]),
page_response_buffer, 1, command_done_flag );
}

inline RETURN_TYPE requestDataToken( volatile unsigned char *
command_done_flag ) {
    return spi_transaction( (unsigned char *)( &command_buffer[6]),
            data_token_buffer,
            DATA_TOKEN_TEST_LENGTH, command_done_flag );
}

inline RETURN_TYPE sendPageBufferMulti( volatile unsigned char *
page_buffer, volatile unsigned char * command_done_flag ) {
    page_buffer[0] = 0xfc; // data token
    page_buffer[513] = 0xff; // crc dummy 1
    page_buffer[514] = 0xff; // crc dummy 2
    page_buffer[515] = 0xff; // place for response
    page_buffer[516] = 0xff; // extra processing byte
```

```
    return spi_transaction( page_buffer, page_response_buffer, 517,
command_done_flag );
}

inline RETURN_TYPE sendPageBufferSingle( volatile unsigned char *
page_buffer, volatile unsigned char * command_done_flag ) {
    page_buffer[0] = 0xfe; // data token
    page_buffer[513] = 0xff; // crc dummy 1
    page_buffer[514] = 0xff; // crc dummy 2
    page_buffer[515] = 0xff; // place for response
    page_buffer[516] = 0xff; // extra processing byte
    return spi_transaction( page_buffer, page_response_buffer, 517,
command_done_flag );
}

inline RETURN_TYPE sendStopTran( volatile unsigned char *
command_done_flag ) {
    return spi_transaction( (volatile unsigned char *)
stop_tran_buffer, page_response_buffer, 2, command_done_flag );
}

inline unsigned char readDataResponse() {
    return page_response_buffer[515];
}

inline RETURN_TYPE receivePage( volatile unsigned char * page_buffer,
        volatile unsigned char * command_done_flag, unsigned short
chars_to_read ) {
    return spi_transaction( (volatile unsigned char *)
page_request_buffer, page_buffer, chars_to_read, command_done_flag );
}


// send command (do not read response)
// Possible ways to improve performance:
// 1) Develop special sendCommand routines for CMD0, CMD8, CMD58
// 2) Remove support for non-SDHC cards (init routine changes required)
// 3) Develop special sendCommand routine for commands w/ zero arg
value
RETURN_TYPE sendCommand( unsigned char cmd, unsigned long arg, volatile
unsigned char * command_done_flag ) {
    unsigned char crc;
    if( command_state != 0 ) {
        return RETURN_BUSY;
    } else {
        command_state = 0x02;
    }
    /// @warning If you add more commands to this code, the following
test may break!
    /// You must lsh 9 if command takes a data address as an argument.
    /// e.g. - CMD17, 18, 24, 25
    if( sd_block_addressing == false && ( ( cmd & 0xf0 ) == 0x10 ) ) {
        arg = arg << 9; // arg *= 512;
    }
    CS_SELECT;
```

```
    command_buffer[0] = (unsigned char) (0x40 | cmd);   // Start +
Command index
    command_buffer[1] = (unsigned char) (arg >> 24);    //
Argument[31..24]
    command_buffer[2] = (unsigned char) (arg >> 16);    //
Argument[23..16]
    command_buffer[3] = (unsigned char) (arg >> 8);     //
Argument[15..8]
    command_buffer[4] = (unsigned char) arg;            //
Argument[7..0]
    crc = 0x01;                                         // Empty CRC +
Stop
    if (cmd == CMD0) crc = 0x95;                        // (valid CRC
for CMD0(0))
    if (cmd == CMD8) crc = 0x87;                        // (valid CRC
for CMD8(0x1AA))
    command_buffer[5] = crc;
    if( cmd==CMD8 || cmd==CMD58 ) {
        return spi_transaction( command_buffer, response,
R3_TEST_LENGTH+6, command_done_flag );
    } else {
        return spi_transaction( command_buffer, response,
R1_TEST_LENGTH+6, command_done_flag );
    }
}

// read R1 card response
// returns R1 card response
unsigned char readR1() {
    if( command_state != 0x02 && command_state != 0x00 ) {
        return 0xff; // return fail
    }
    unsigned char i=6;
    for( i=6 ; i<R1_TEST_LENGTH+6 ; i++ ) {
        if( ! (response[i] & 0x80) ) {
            command_state = 0x00;
            return response[i];
        }
    }
    command_state = 0x00;
    return response[R1_TEST_LENGTH+6-1];
}

// read R1 response after a CMD12
// returns R1 card response
unsigned char readR1_CMD12() {
    if( command_state != 0x02 && command_state != 0x00 ) {
        return 0xff; // return fail
    }
    unsigned char i=7;
    // "The received byte immediately following CMD12 is a stuff byte.
    // It should be discarded before receiving the response."
    // -- elm-chan.org/docs/mmc/mmc_e.html
    for( i=7 ; i<R1_TEST_LENGTH+6 ; i++ ) {
        if( ! (response[i] & 0x80) ) {
```

```
            command_state = 0x00;
            return response[i];
        }
    }
    command_state = 0x00;
    return response[R1_TEST_LENGTH+6-1];
}


// read R1 card response and check for data token
// returns R1 card response and checks for data token
// return_data_read_count will return 0 if data token is not found
// 1 if data token only is found
// 1 + <number of data tokens transferred to the data buffer>
// -1 if data token is error token
inline unsigned char readR1andDataToken( volatile unsigned char *
page_buffer ) {
    if( command_state != 0x02 && command_state != 0x00 ) {
        return 0xff; // return fail
    }
    unsigned char resp = 0xFF;
    bytes_read = 0;
    unsigned char i=6, j=0;
    // look for response
    for( i=6 ; i<R1_TEST_LENGTH+6 ; i++ ) {
        if( ! (response[i] & 0x80) ) {
            resp = response[i];
            break;
        }
    }
    // look for data token
    for( i++ ; i<R1_TEST_LENGTH+6 ; i++ ) {
        if( response[i] == 0xfe ) { // data token
            bytes_read++;
            break;
        } else if( (response[i] & 0xe0 ) == 0 ) { // error token
            bytes_read = -1;
            command_state = 0x00;
            return resp;
        } else {
            // busy
        }
    }
    // copy data to buffer
    for( i++ ; i<R1_TEST_LENGTH+6 ; i++ ) {
        page_buffer[j] = response[i];
        j++;
        bytes_read++;
    }
    command_state = 0x00;
    return resp;
}

void readDataToken( volatile unsigned char * page_buffer ) {
    bytes_read = 0;
```

```
    unsigned char i=0, j=0;
    // look for data token
    for( i=0 ; i<DATA_TOKEN_TEST_LENGTH ; i++ ) {
        if( data_token_buffer[i] == 0xfe ) { // data token
            bytes_read++;
            data_token_buffer[i] = 0xff;
            break;
        } else if( (data_token_buffer[i] & 0xe0 ) == 0 ) { // error
token
            bytes_read = -1;
            return;
        } else {
            // busy
        }
    }
    // copy data to buffer
    for( i++ ; i<DATA_TOKEN_TEST_LENGTH ; i++ ) {
        page_buffer[j] = data_token_buffer[i];
        data_token_buffer[i] = 0xff;
        j++;
        bytes_read++;
    }
    return;
}

// returns R1 card response
// response_flags returns R3/R7 flags
unsigned char readR3(
        volatile unsigned char * response_flags // 4 byte character
array
        ) {
    if( command_state != 0x02 ) {
        return 0xff; // return fail
    }
    unsigned char i=6;
    for( i=6 ; i<R1_TEST_LENGTH+6 ; i++ ) {
        if( ! (response[i] & 0x80) ) {
            response_flags[0] = response[i+1u];
            response_flags[1] = response[i+2u];
            response_flags[2] = response[i+3u];
            response_flags[3] = response[i+4u];
            command_state = 0x00;
            return response[i];
        }
    }
    command_state = 0x00;
    return response[R1_TEST_LENGTH+6-1];
}

// send command and return response (busy waits for response)
// initialize timeout timer before calling
// will not modify value of timeout_counter
unsigned char commandAndResponse(
        unsigned char cmd, // [in] command index
        unsigned long arg, // [in] argument
```

```
        volatile const unsigned long * timeout, // [in/out] timeout
variable (5 kHz or lower)
        volatile unsigned char * R3_code // [out] R3 response (4 byte
char array)
        ) {
    unsigned char trans_done = 0;
    unsigned long timeout_copy = 0;
    unsigned char tresponse = 0xff;
    RETURN_TYPE spi_stat = RETURN_ERR;

    if( cmd & 0x80 ) {  // ACMD
        cmd &= 0x7F;    // bit 7 should not be set
        tresponse = commandAndResponse( CMD55, 0, timeout, R3_code );
// ACMDn is CMD55-CMDn
        if ( tresponse > 1 ) return tresponse;
    }

    if( cmd==CMD8 || cmd==CMD58 ) {
        // send command
        trans_done = 0;
        timeout_copy = (*timeout);
        spi_stat = sendCommand( cmd, arg, &trans_done );
        if( spi_stat != RETURN_OK ) {
            return 0xff;
        }
        // 1/10 interrupts per byte * 20 bytes * 3 for overhead = 6
        while( (trans_done==0) && (((*timeout)-timeout_copy)<=6) );
        if( ( (*timeout) - timeout_copy ) > 6 ) {
            timeout_copy = (*timeout);
            return 0xff;
        }
        // read response
        return readR3( R3_code );
    } else {
        // send command
        trans_done = 0;
        timeout_copy = (*timeout);
        spi_stat = sendCommand( cmd, arg, &trans_done );
        if( spi_stat != RETURN_OK ) {
            return 0xff;
        }
        // 1/10 interrupts per byte * 16 bytes * 3 for overhead = 5
        while( (trans_done==0) && (((*timeout)-timeout_copy)<=5) );
        if( ( (*timeout) - timeout_copy ) > 5 ) {
            timeout_copy = (*timeout);
            return 0xff;
        }
        // read response
        return readR1();
    }
}

inline SD_TYPE process_STOP_TRAN() {
    if( cmd_flag ) {
        if( SD_NOT_BUSY ) {
```

```
            return SD_CONTINUE;
        } else {
            return SD_BUSY_RETRY;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_WRITE_SINGLE_BLOCK() {
    if( cmd_flag ) {
        if( SD_NOT_BUSY ) {
            if( ( readDataResponse() & 0x1F ) == 0x05 ) {
                return SD_CONTINUE;
            } else {
                process_state = IDLE;
                return SD_ERR_RESTART;
            }
        } else {
            return SD_BUSY_RETRY;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD12() {
    if( cmd_flag ) {
        if( SD_NOT_BUSY ) {
            if( readR1_CMD12() == 0 ) {
                return SD_CONTINUE;
            } else {
                return SD_ERR_SEND_STOP;
            }
        } else {
            return SD_BUSY_RETRY;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_READ_SINGLE_BLOCK( volatile unsigned char *
page_buffer ) {
    if( cmd_flag ) {
        readDataToken( page_buffer );
        if( bytes_read < 0 ) {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
        if( bytes_read == 0 ) {
            cmd_flag = 0;
            if( ! requestDataToken( & cmd_flag ) ) {
                return SD_BUSY_RETRY;
            } else {
```

```
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        }
        return SD_CONTINUE;
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_READ_MULTI_BLOCK( volatile unsigned char *
page_buffer ) {
    if( cmd_flag ) {
        readDataToken( page_buffer );
        if( bytes_read < 0 ) {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
        if( bytes_read == 0 ) {
            cmd_flag = 0;
            if( ! requestDataToken( & cmd_flag ) ) {
                return SD_BUSY_RETRY;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        }
        return SD_CONTINUE;
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_READ_BLOCK_dumb() {
    if( cmd_flag ) {
        return SD_CONTINUE;
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD25() {
    if( cmd_flag ) {
        if( readR1() == 0 ) {
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_WRITE_MULTI_BLOCK() {
```

```
    if( cmd_flag ) {
        if( SD_NOT_BUSY ) {
            if( ( readDataResponse() & 0x1F ) == 0x05 ) {
                return SD_CONTINUE;
            } else {
                return SD_ERR_SEND_STOP;
            }
        } else {
            return SD_BUSY_RETRY;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD24() {
    if( cmd_flag ) {
        if( readR1() == 0 ) {
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD18_dumb() {
    if( cmd_flag ) {
        if( readR1() == 0 ) {
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD18( volatile unsigned char * page_buffer
) {
    if( cmd_flag ) {
        if( readR1andDataToken( page_buffer ) == 0 ) {
            if( bytes_read < 0 ) {
                process_state = IDLE;
                return SD_ERR_RESTART;
            }
            if( bytes_read == 0 ) {
                cmd_flag = 0;
                if( ! requestDataToken( & cmd_flag ) ) {
                    process_state = READ_MULTI_BLOCK;
                    return SD_BUSY_RETRY; // even though the state
transitions, return busy b/c the read is not started yet
```

```
                } else {
                    cmd_flag = 1;
                    return SD_ERR_RETRY;
                }
            }
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD17( volatile unsigned char * page_buffer
) {
    if( cmd_flag ) {
        if( readR1andDataToken( page_buffer ) == 0 ) {
            if( bytes_read < 0 ) {
                process_state = IDLE;
                return SD_ERR_RESTART;
            }
            if( bytes_read == 0 ) {
                cmd_flag = 0;
                if( ! requestDataToken( & cmd_flag ) ) {
                    process_state = READ_SINGLE_BLOCK;
                    return SD_BUSY_RETRY; // even though the state
transitions, return busy b/c the read is not started yet
                } else {
                    cmd_flag = 1;
                    return SD_ERR_RETRY;
                }
            }
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
}

inline SD_TYPE process_SEND_CMD17_dumb() {
    if( cmd_flag ) {
        if( readR1() == 0 ) {
            return SD_CONTINUE;
        } else {
            process_state = IDLE;
            return SD_ERR_RESTART;
        }
    } else {
        return SD_BUSY_RETRY;
    }
```

```
}

//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
// EXTERNAL FUNCTION DEFINITIONS
//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

/*!
 * @brief Initializes SD card.
 *
 * @note completely resets SD card by power cycling before
initializing.
 *
 * Commands valid after this command:   \n
 * \b sd_startMultiBlockWrite()         \n
 * \b sd_startSingleBlockWrite()        \n
 * \b sd_startMultiBlockRead()          \n
 * \b sd_startSingleBlockRead()         \n
 *
 * Commands valid before this command:  \n
 * \b Any/None                              \n
 *
 * @return \ref RETURN_TYPE
 * @retval RETURN_NEEDINTERRUPT General interrupts must be enabled.
 * @retval RETURN_NOTFOUND  SD card not present.
 * @retval RETURN_PERIPHERR SPI interface error.
 * @retval RETURN_TIMEOUT   Device timed out.
 * @retval RETURN_NOINIT    Device could not be initialized.
 * @retval RETURN_OK        Device initialized successfully.
 */
RETURN_TYPE sd_initialize( void ) {
    unsigned char dummy[10] = // used for dummy clocks
            { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
0xff };
    RETURN_TYPE spi_stat = RETURN_ERR;
    unsigned long calculated_timeout_freq = 0;
    volatile unsigned long timeout_counter = 0; // timeout
    unsigned char trans_done = 0;
    //unsigned char r1_resp = 0xff;
    volatile unsigned char r3_flags[4] = { 0xff, 0xff, 0xff, 0xff };

    // require interrupt
    assert( (__get_SR_register() & GIE) );

    /// __Configuration details:__

    ///-# Initialize SPI pins to SD card.
    init_pins();

    ///-# Configure chip select to SD card.
    CS_INIT();
    CS_DESELECT;

    ///-# Detect card.
    if( ! CARD_DETECT ) {
        return RETURN_NOTFOUND;
```

```
    }

    ///-# Initialize SPI to 400 kHz for SD initialization.
    spi_initialize( SD_INIT_FREQ );

    ///-# Initialize timeout clock for 5 kHz (200 us)
    calculated_timeout_freq = timeout_initialize( CLK_SRC_SMCLK, 5000
);
    if( calculated_timeout_freq != 5000 ) {
        return RETURN_PERIPHERR;
    }
    ///-# Start timeout clock
    timeout_counter = 0;
    if( timeout_start( (unsigned long *) &timeout_counter ) ) {
        return RETURN_PERIPHERR;
    }

    ///-# Power cycle SD to achieve known reset state. @note This is
important.
    /// @todo This goes in memory.c later???
    timeout_counter = 0;
    SD_ONOFF_INIT();
    while( timeout_counter < 1250 ); // 1250 = wait 250 ms for SD to
fully power off
    timeout_counter = 0;
    SD_ON;
    while( timeout_counter < 1250 ); // 1250 = wait 250 ms for SD to
reach stable voltage

    ///-# Send 74 or more dummy clocks (send 80) with CS high
(deselected)
    CS_DESELECT;
    trans_done = 0;
    timeout_counter = 0;
    spi_stat = spi_transaction( (volatile unsigned char *)
page_request_buffer, dummy, 10, &trans_done );
    if( spi_stat != RETURN_OK ) {
        timeout_stop();
        return RETURN_PERIPHERR;
    }
    // 1/10 interrupts per byte * 10 bytes * 3 for overhead = 3
    while( (trans_done==0) && (timeout_counter<=4) );
    if( timeout_counter > 4 ) {
        timeout_stop();
        return RETURN_TIMEOUT;
    }

    ///-# Issue CMD0 with CS low (selected)
    CS_SELECT;
    if( commandAndResponse( CMD0, 0, (unsigned long *)
&timeout_counter, r3_flags ) == 1 ) {
        timeout_counter = 0; // reset timeout to 0 (12500 is 1 sec)
        ///-# Issue CMD8 to check voltage range
        if( commandAndResponse( CMD8, 0x000001AA,
                (unsigned long *) &timeout_counter,
```

```
                r3_flags ) == 1 ) { // SDv2?
            if( (r3_flags[2] == 0x01) && (r3_flags[3] == 0xaa) ) { //
SDv2 & card works at Vdd 2.7-3.6V
                ///-# Issue ACMD41 waiting for idle
                while( (timeout_counter<15000) &&
                        commandAndResponse( ACMD41, 1UL << 30,
                                (unsigned long *) &timeout_counter,
                                r3_flags ) ); // wait for idle w/
ACMD41
                ///-# Read OCR, check CCS bit (indicates SDHC/SDXC
card)
                if( (timeout_counter<15000) &&
                        (commandAndResponse( CMD58, 0,
                                (unsigned long *) &timeout_counter,
                                r3_flags )==0) ) { // Req card OCR
register
                    sd_block_addressing = (r3_flags[0] & 0x40) ? (true)
: (false); // CCS bit in OCR -> SDHC
                    ///-# Re-init SPI at max SD frequency
                    spi_initialize( SD_MAX_FREQ );
                    if( spi_stat ) {
                        timeout_stop();
                        CS_DESELECT;
                        return RETURN_PERIPHERR;
                    } else {
                        ///-# Stop timeout timer
                        timeout_stop();
                        CS_DESELECT;
                        process_state = IDLE;
                        return RETURN_OK;
                    }
                }
            }
        }
    }
    // if it's not version 2+, we can give up, version 1 doesn't have
enough storage space
    timeout_stop();
    CS_DESELECT;
    return RETURN_NOINIT;
}

/*!
 * @brief Start a multi-block write process.
 *
 * Commands valid after this command:   \n
 * \b sd_writeBlockBuffer(...)          \n
 * \b sd_stopMultiBlockWrite()          \n
 *
 * Commands valid before this command:  \n
 * \b sd_initialize()                   \n
 * \b sd_stopMultiBlockWrite()          \n
 * \b sd_writeBlockBuffer() when preceded by sd_startSingleBlockWrite()
\n
 * \b sd_stopMultiBlockRead()           \n
```

```
 * \b sd_readBlock() when preceded by sd_startSingleBlockRead() \n
 *
 * If the next command responds with SD_ERR_RESTART,
sd_startMultiBlockWrite(...) should
 * be re-issued.
 *
 * @param [in]  block_start Block number defining start location in
memory.
 * Writes to contiguous blocks.
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE          Command was issued successfully. System
is ready for
 *                              blocks to be sent or for a stop multi-
block write command.
 * @retval SD_ERR_RETRY         There was an error issuing the command.
Retry. If error
 *                              persists, there may be a critical
error.
 * @retval SD_INVALID_CMD       The command is not valid at this time.
 * @retval SD_BUSY_RETRY        The card is busy. Please retry the
command. If card continues
 *                              to be busy for a very long period of
time (>1s), a greater
 *                              problem may exist.
 * @retval SD_CONTINUE_WITH_ERR The command was issued successfully;
however, there was a
 *                              problem with the previous process or
command.
 *                              Specifically, if a single block was
written or read most
 *                              recently, this indicates that the block
was not read or
 *                              written successfully.
 * @retval SD_ERR_SEND_STOP     There was an error with stopping the
multi-block read process.
 *                              Re-issue the sd_stopMultiBlockRead()
command.
 * @retval SD_ERR               Unknown error. Indicates a critical
error.
 */
SD_TYPE sd_startMultiBlockWrite( unsigned long block_start ) {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
    case IDLE:
        cmd_flag = 0;
        if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD25;
            return SD_CONTINUE;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    case SEND_CMD25:
        return SD_INVALID_CMD;
```

```
case WRITE_MULTI_BLOCK:
    return SD_INVALID_CMD;
case STOP_TRAN:
    ret = process_STOP_TRAN();
    if( ret == SD_BUSY_RETRY ) {
        return SD_BUSY_RETRY;
    } else if( ret == SD_CONTINUE ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD25;
            return SD_CONTINUE;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    }
    return SD_ERR;
case SEND_CMD24:
    return SD_INVALID_CMD;
case WRITE_SINGLE_BLOCK:
    ret = process_WRITE_SINGLE_BLOCK();
    if( ret == SD_BUSY_RETRY ) {
        return SD_BUSY_RETRY;
    } else if( ret == SD_CONTINUE ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD25;
            return SD_CONTINUE;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    } else if( ret == SD_ERR_RESTART ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD25;
            return SD_CONTINUE_WITH_ERR;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    }
    return SD_ERR;
case SEND_CMD18:
    return SD_INVALID_CMD;
case READ_MULTI_BLOCK:
    return SD_INVALID_CMD;
case SEND_CMD12:
    ret = process_SEND_CMD12();
    if( ret == SD_BUSY_RETRY ) {
        return SD_BUSY_RETRY;
    } else if( ret == SD_CONTINUE ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD25;
```

```
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_SEND_STOP ) {
            return SD_ERR_SEND_STOP;
        }
        return SD_ERR;
    case SEND_CMD17:
        return SD_INVALID_CMD;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
//      ret = process_READ_SINGLE_BLOCK();
//      if( ret == SD_BUSY_RETRY ) {
//          return SD_BUSY_RETRY;
//      } else if( ret == SD_CONTINUE ) {
//          cmd_flag = 0;
//          if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
//              process_state = SEND_CMD25;
//              return SD_CONTINUE;
//          } else {
//              cmd_flag = 1;
//              return SD_ERR_RETRY;
//          }
//      } else if( ret == SD_ERR_RESTART ) {
//          cmd_flag = 0;
//          if( ! sendCommand( CMD25, block_start, &cmd_flag ) ) {
//              process_state = SEND_CMD25;
//              return SD_CONTINUE_WITH_ERR;
//          } else {
//              cmd_flag = 1;
//              return SD_ERR_RETRY;
//          }
//      }
//      return SD_ERR;
    default:
        _never_executed();
    }
    return SD_ERR;
}

/*!
 * @brief Send a block to the SD card from the buffer during a multi-
block or single-block write process.
 *
 * Commands valid after this command:   \n
 * \b sd_writeBlockBuffer() when preceded by sd_startMultiBlockWrite()
and any number of sd_writeBlockBuffer() commands \n
 * \b sd_stopMultiBlockWrite() when preceded by
sd_startMultiBlockWrite() and any number of sd_writeBlockBuffer()
commands  \n
 * \b sd_startMultiBlockWrite() when preceded by
sd_startSingleBlockWrite() \n
```

```
 * \b sd_startSingleBlockWrite() when preceded by
sd_startSingleBlockWrite()    \n
 * \b sd_startMultiBlockRead() when preceded by
sd_startSingleBlockWrite()  \n
 * \b sd_startSingleBLockRead() when preceded by
sd_startSingleBlockWrite() \n
 *
 * Commands valid before this command:  \n
 * \b sd_writeBlockBuffer() when preceded by sd_startMultiBlockWrite()
and any number of sd_writeBlockBuffer() commands \n
 * \b sd_startMultiBlockWrite()         \n
 * \b sd_startSingleBlockWrite()        \n
 *
 * @param [in] block_buffer Buffer containing data to be sent.
 * @note Should be 517 bytes in length. \n
 * Byte  0 is reserved.                 \n
 * Bytes 1-512 are data.                \n
 * Bytes 513-516 are reserved.          \n
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE           Command was issued successfully. System
is ready for more
 *                               blocks to be sent or for a stop multi-
block write command.
 * @retval SD_ERR_RETRY          There was an error issuing the command.
Retry. If error
 *                               persists, there may be a critical
error.
 * @retval SD_INVALID_CMD        The command is not valid at this time.
 * @retval SD_BUSY_RETRY         The card is busy. Please retry the
command. If card continues
 *                               to be busy for a very long period of
time (>1s), a greater
 *                               problem may exist.
 * @retval SD_ERR_RESTART        There was an error with the previous
command. Re-send
 *                               sd_startMultiBlockWrite() or
sd_startSingleBlockWrite().
 * @retval SD_ERR_SEND_STOP      There was an error with the previous
sd_writeBlockBuffer()
 *                               command. The last data block was not
written successfully.
 *                               Send sd_stopMultiBlockWrite() before
any other actions.
 * @retval SD_ERR                Unknown error. Indicates a critical
error.
 */
SD_TYPE sd_writeBlockBuffer( volatile unsigned char * block_buffer ) {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
    case IDLE:
        return SD_INVALID_CMD;
    case SEND_CMD25:
        ret = process_SEND_CMD25();
        if( ret == SD_BUSY_RETRY ) {
```

```
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendPageBufferMulti( block_buffer, &cmd_flag ) ) {
                process_state = WRITE_MULTI_BLOCK;
                _nop();
                return SD_CONTINUE;
            }
            else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
        return SD_ERR;
    case WRITE_MULTI_BLOCK:
        ret = process_WRITE_MULTI_BLOCK();
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendPageBufferMulti( block_buffer, &cmd_flag ) ) {
                //process_state = WRITE_MULTI_BLOCK;
                return SD_CONTINUE;
            }
            else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_SEND_STOP ) {
            return SD_ERR_SEND_STOP;
        }
        return SD_ERR;
    case STOP_TRAN:
        return SD_INVALID_CMD;
    case SEND_CMD24:
        ret = process_SEND_CMD24();
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendPageBufferMulti( block_buffer, &cmd_flag ) ) {
                process_state = WRITE_SINGLE_BLOCK;
                return SD_CONTINUE;
            }
            else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
```

```
        return SD_ERR;
    case WRITE_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD18:
        return SD_INVALID_CMD;
    case READ_MULTI_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD12:
        return SD_INVALID_CMD;
    case SEND_CMD17:
        return SD_INVALID_CMD;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    default:
        _never_executed();
    }
    return SD_ERR;
}

/*!
 * @brief Stop multi-block write process.
 *
 * Commands valid after this command:   \n
 * \b sd_startMultiBlockWrite()         \n
 * \b sd_startSingleBlockWrite()        \n
 * \b sd_startMultiBlockRead()          \n
 * \b sd_startSingleBlockRead()         \n
 *
 * Commands valid before this command:  \n
 * \b sd_startMultiBlockWrite()         \n
 * \b sd_writeBlockBuffer() when preceded by sd_startMultiBlockWrite()
and any number of sd_writeBlockBuffer() commands \n
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE          Command was issued successfully. System
is ready for a
 *                              new process.
 * @retval SD_ERR_RETRY         There was an error issuing the command.
Retry. If error
 *                              persists, there may be a critical
error.
 * @retval SD_INVALID_CMD       The command is not valid at this time.
 * @retval SD_BUSY_RETRY        The card is busy. Please retry the
command. If card continues
 *                              to be busy for a very long period of
time (>1s), a greater
 *                              problem may exist.
 * @retval SD_CONTINUE_WITH_ERR The command was issued successfully;
however, there was a
 *                              problem with the previous command. The
last data block was
 *                              not written successfully.
 * @retval SD_ERR               Unknown error. Indicates a critical
error.
 */
```

```
SD_TYPE sd_stopMultiBlockWrite( void ) {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
    case IDLE:
        return SD_INVALID_CMD;
    case SEND_CMD25:
        ret = process_SEND_CMD25();
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendStopTran( &cmd_flag ) ) {
                process_state = STOP_TRAN;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
    case WRITE_MULTI_BLOCK:
        ret = process_WRITE_MULTI_BLOCK();
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendStopTran( &cmd_flag ) ) {
                process_state = STOP_TRAN;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_SEND_STOP ) {
            cmd_flag = 0;
            if( ! sendStopTran( &cmd_flag ) ) {
                process_state = STOP_TRAN;
                return SD_CONTINUE_WITH_ERR;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        }
        return SD_ERR;
    case STOP_TRAN:
        return SD_INVALID_CMD;
    case SEND_CMD24:
        return SD_INVALID_CMD;
    case WRITE_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD18:
        return SD_INVALID_CMD;
    case READ_MULTI_BLOCK:
```

```
        return SD_INVALID_CMD;
    case SEND_CMD12:
        return SD_INVALID_CMD;
    case SEND_CMD17:
        return SD_INVALID_CMD;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    default:
        _never_executed();
    }
    return SD_ERR;
}

/*!
 * @brief Checks for SD process errors.
 * @return \ref SD_TYPE
 * @retval SD_CONINTUE      No errors.
 * @retval SD_BUSY_RETRY    Card is busy.
 * @retval SD_ERR_RESTART   There was an error with the previous
command.
 *                          Re-start the read/write process.
 * @retval SD_ERR_SEND_STOP There was an error with the previous
command.
 *                          Send the appropriate stop process command,
then retry.
 * @retval SD_ERR           Unknown error. Indicates a critical error.
 */
SD_TYPE sd_checkErrors( void ) {
        switch( process_state ) {
        case IDLE:
            return SD_CONTINUE;
        case SEND_CMD25:
            return process_SEND_CMD25();
        case WRITE_MULTI_BLOCK:
            return process_WRITE_MULTI_BLOCK();
        case STOP_TRAN:
            return process_STOP_TRAN();
        case SEND_CMD24:
            return process_SEND_CMD24();
        case WRITE_SINGLE_BLOCK:
            return process_WRITE_SINGLE_BLOCK();
        case SEND_CMD18:
            return process_SEND_CMD18_dumb();
        case READ_MULTI_BLOCK:
            return process_READ_BLOCK_dumb();
        case SEND_CMD12:
            return process_SEND_CMD12();
        case SEND_CMD17:
            return process_SEND_CMD17_dumb();
        case READ_SINGLE_BLOCK:
            return process_READ_BLOCK_dumb();
        default:
            _never_executed();
        }
        return SD_ERR;
```

```
}

/*!
 * @brief Starts a multi-block read process.
 *
 * Commands valid after this command:   \n
 * \b sd_readBlock()                    \n
 * \b sd_stopMultiBlockRead()           \n
 *
 * Commands valid before this command:  \n
 * \b sd_initialize()                   \n
 * \b sd_stopMultiBlockWrite()          \n
 * \b sd_stopMultiBlockRead()           \n
 * \b sd_writeBlockBuffer() when preceded by sd_startSingleBlockWrite()
\n
 * \b sd_readBlockBuffer() when preceded by sd_startSingleBlockRead()
\n
 *
 * @param [in] block_start  The first block to read. Reads contiguous
blocks.
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE                  Command was issued successfully.
System is ready for
 *                                      blocks to be read or for a stop
multi-block write command.
 * @retval SD_BUSY_RETRY                The card is busy. Please retry the
command. If the card
 *                                      continues to be busy for a very
long time (>1s), a greater
 *                                      problem may exist.
 * @retval SD_ERR_RETRY                 There was an error issuing the
command. Retry. If error
 *                                      persists, there may be a critical
error.
 * @retval SD_INVALID_CMD               The command is not valid at this
time.
 * @retval SD_CONTINUE_WITH_ERR         The command was issued
successfully; however, there was a
 *                                      problem with the previous process
or command. Specifically,
 *                                      if a single block was written or
read most recently, this
 *                                      indicates that the block was not
read or written successfully.
 * @retval SD_ERR_SEND_STOP             There was a problem stopping the
multi-block read process.
 *                                      Re-issue the
sd_stopMultiBlockRead() command.
 * @retval SD_ERR                       Unknown error. Indicates a critical
error.
 */
SD_TYPE sd_startMultiBlockRead( unsigned long block_start ) {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
```

```
case IDLE:
    cmd_flag = 0;
    if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
        process_state = SEND_CMD18;
        return SD_CONTINUE;
    } else {
        cmd_flag = 1;
        return SD_ERR_RETRY;
    }
    // return SD_ERR;
case SEND_CMD25:
    return SD_INVALID_CMD;
case WRITE_MULTI_BLOCK:
    return SD_INVALID_CMD;
case STOP_TRAN:
    ret = process_STOP_TRAN();
    if( ret == SD_BUSY_RETRY ) {
        return SD_BUSY_RETRY;
    } else if( ret == SD_CONTINUE ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD18;
            return SD_CONTINUE;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    }
    return SD_ERR;
case SEND_CMD24:
    return SD_INVALID_CMD;
case WRITE_SINGLE_BLOCK:
    ret = process_WRITE_SINGLE_BLOCK();
    if( ret == SD_BUSY_RETRY ) {
        return SD_BUSY_RETRY;
    } else if( ret == SD_CONTINUE ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD18;
            return SD_CONTINUE;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    } else if( ret == SD_ERR_RESTART ) {
        cmd_flag = 0;
        if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
            process_state = SEND_CMD18;
            return SD_CONTINUE_WITH_ERR;
        } else {
            cmd_flag = 1;
            return SD_ERR_RETRY;
        }
    }
    return SD_ERR;
```

```
    case SEND_CMD18:
        return SD_INVALID_CMD;
    case READ_MULTI_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD12:
        ret = process_SEND_CMD12();
        if( ret == SD_BUSY_RETRY ){
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
                process_state = SEND_CMD18;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_SEND_STOP ) {
            return SD_ERR_SEND_STOP;
        }
        return SD_ERR;
    case SEND_CMD17:
        return SD_INVALID_CMD;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
//      ret = process_READ_SINGLE_BLOCK();
//      if( ret == SD_BUSY_RETRY ) {
//          return SD_BUSY_RETRY;
//      } else if( ret == SD_CONTINUE ) {
//          cmd_flag = 0;
//          if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
//              process_state = SEND_CMD18;
//              return SD_CONTINUE;
//          } else {
//              cmd_flag = 1;
//              return SD_ERR_RETRY;
//          }
//      } else if( ret == SD_ERR_RESTART ) {
//          cmd_flag = 0;
//          if( ! sendCommand( CMD18, block_start, &cmd_flag ) ) {
//              process_state = SEND_CMD18;
//              return SD_CONTINUE_WITH_ERR;
//          } else {
//              cmd_flag = 1;
//              return SD_ERR_RETRY;
//          }
//      }
//      return SD_ERR;
    default:
        _never_executed();
    }
    return SD_ERR;
}

/*!
```

```
 * @brief Read a block from the SD card to the buffer during a multi-
block or single-block read process.
 *
 * Commands valid after this command:   \n
 * \b sd_readBlock() when preceded by sd_startMultiBlockRead() and any
number of sd_readBlock() commands    \n
 * \b sd_stopMultiBlockRead() when preceded by sd_startMultiBlockRead()
and any number of sd_readBlock() commands   \n
 * \b sd_startMultiBlockRead() when preceded by
sd_startSingleBlockRead()   \n
 * \b sd_startSingleBLockRead() when preceded by
sd_startSingleBlockRead()  \n
 * \b sd_startMultiBlockWrite() when preceded by
sd_startSingleBlockRead()  \n
 * \b sd_startSingleBlockWrite() when preceded by
sd_startSingleBlockRead() \n
 *
 * Commands valid before this command:  \n
 * \b sd_readBlock()
 * @note ONLY when preceded by sd_startMultiBlockRead() and any number
of sd_readBlock() commands   \n
 * \b sd_startMultiBlockRead()          \n
 * \b sd_startSingleBlockRead()         \n
 *
 * @param [in] block_buffer Buffer into which SD read data will be
deposited.
 * @note Should be 515 bytes in length. \n
 * Bytes 0-511 are data.                \n
 * Bytes 512-514 are reserved.          \n
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE          Command was issued successfully. System
is ready for more
 *                              blocks to be sent or for a stop multi-
block write command.
 * @retval SD_BUSY_RETRY        The card is busy. Please retry the
command. If card continues
 *                              to be busy for a very long period of
time (>1s), a greater
 *                              problem may exist.
 * @retval SD_ERR_RETRY         There was an error issuing the command.
Retry. If error
 *                              persists, there may be a critical
error.
 * @retval SD_INVALID_CMD       The command is not valid at this time.
 * @retval SD_ERR_RESTART       There was an error with the previous
command. Re-issue
 *                              sd_startMultiBlockRead() or
sd_startSingleBlockRead().
 * @retval SD_ERR_SEND_STOP     There was an error with the previous
sd_readBlock()
 *                              command. The last data block was not
read successfully.
 *                              Send sd_stopMultiBlockRead() before any
other actions.
```

```
 * @retval SD_ERR              Unknown error. Indicates a critical
error.
 */
SD_TYPE sd_readBlock( volatile unsigned char * block_buffer ) {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
    case IDLE:
        return SD_INVALID_CMD;
    case SEND_CMD25:
        return SD_INVALID_CMD;
    case WRITE_MULTI_BLOCK:
        return SD_INVALID_CMD;
    case STOP_TRAN:
        return SD_INVALID_CMD;
    case SEND_CMD24:
        return SD_INVALID_CMD;
    case WRITE_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD18:
        ret = process_SEND_CMD18( block_buffer );
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! receivePage( & block_buffer[bytes_read-1],
                    &cmd_flag, 515-bytes_read+1 ) ) {
                process_state = READ_MULTI_BLOCK;
                return SD_CONTINUE;
            }
            else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
        return SD_ERR;
    case READ_MULTI_BLOCK:
        ret = process_READ_MULTI_BLOCK( block_buffer );
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! receivePage( & block_buffer[bytes_read-1],
                    &cmd_flag, 515-bytes_read+1 ) ) {
                // process_state = READ_MULTI_BLOCK;
                return SD_CONTINUE;
            }
            else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
```

```
            return SD_ERR_RESTART;
        }
        return SD_ERR;
    case SEND_CMD12:
        return SD_INVALID_CMD;
    case SEND_CMD17:
        ret = process_SEND_CMD17( block_buffer );
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! receivePage( & block_buffer[bytes_read-1],
                    &cmd_flag, 515-bytes_read+1 ) ) {
                process_state = READ_SINGLE_BLOCK;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
        return SD_ERR;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    default:
        _never_executed();
    }
    return SD_ERR;
}

/*!
 * @brief Stops a multi-block read process.
 *
 * Commands valid after this command:   \n
 * \b sd_startMultiBlockRead()          \n
 * \b sd_startSingleBLockRead()         \n
 * \b sd_startMultiBlockWrite()         \n
 * \b sd_startSingleBlockWrite()        \n
 *
 * Commands valid before this command:  \n
 * \b sd_readBlock()
 * @note ONLY when preceded by sd_startMultiBlockRead() and any number
of sd_readBlock() commands   \n
 * \b sd_startMultiBlockRead()          \n
 *
 * @return \ref SD_TYPE
 * @retval SD_CONTINUE          Command was issued successfully. System
is ready for more
 *                              blocks to be sent or for a stop multi-
block write command.
 * @retval SD_BUSY_RETRY        The card is busy. Please retry the
command. If card continues
```

```
 *                              to be busy for a very long period of
time (>1s), a greater
 *                              problem may exist.
 * @retval SD_ERR_RETRY         There was an error issuing the command.
Retry. If error
 *                              persists, there may be a critical
error.
 * @retval SD_INVALID_CMD       The command is not valid at this time.
 * @retval SD_CONTINUE_WITH_ERR The command was issued successfully;
however, there was a
 *                              problem with the previous process or
command. The last data
 *                              block was not read successfully.
 * @retval SD_ERR_RESTART       There was an error with the previous
command. Re-issue
 *                              sd_startMultiBlockRead().
 * @retval SD_ERR               Unknown error. Indicates a critical
error.
 */
SD_TYPE sd_stopMultiBlockRead() {
    SD_TYPE ret = SD_ERR;
    switch( process_state ) {
    case IDLE:
        return SD_INVALID_CMD;
    case SEND_CMD25:
        return SD_INVALID_CMD;
    case WRITE_MULTI_BLOCK:
        return SD_INVALID_CMD;
    case STOP_TRAN:
        return SD_INVALID_CMD;
    case SEND_CMD24:
        return SD_INVALID_CMD;
    case WRITE_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    case SEND_CMD18:
        ret = process_SEND_CMD18_dumb();
        if( ret == SD_BUSY_RETRY ) {
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendCommand( CMD12, 0, &cmd_flag ) ) {
                process_state = SEND_CMD12;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_RESTART ) {
            //process_state = IDLE;
            return SD_ERR_RESTART;
        }
        return SD_ERR;
    case READ_MULTI_BLOCK:
        ret = process_READ_BLOCK_dumb();
        if( ret == SD_BUSY_RETRY ) {
```

```
            return SD_BUSY_RETRY;
        } else if( ret == SD_CONTINUE ) {
            cmd_flag = 0;
            if( ! sendCommand( CMD12, 0, &cmd_flag ) ) {
                process_state = SEND_CMD12;
                return SD_CONTINUE;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        } else if( ret == SD_ERR_SEND_STOP ) {
            cmd_flag = 0;
            if( ! sendCommand( CMD12, 0, &cmd_flag ) ) {
                process_state = SEND_CMD12;
                return SD_CONTINUE_WITH_ERR;
            } else {
                cmd_flag = 1;
                return SD_ERR_RETRY;
            }
        }
        return SD_ERR;
    case SEND_CMD12:
        return SD_INVALID_CMD;
    case SEND_CMD17:
        return SD_INVALID_CMD;
    case READ_SINGLE_BLOCK:
        return SD_INVALID_CMD;
    default:
        _never_executed();
    }
    return SD_ERR;
}

//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
// MEMORY READ/WRITE SPEED TEST FUNCTIONS
//||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

/*!
 * @brief Perform a sustained read test on the memory.
 *
 * Use after SD initialization function.
 *
 * @warning This is to be used for testing only and should not be used
in release code.
 *
 * @warning This test may overwrite any data on the memory device.
 *
 * @return \ref RETURN_TYPE
 */
RETURN_TYPE memory_sustainedReadTest( unsigned char file16_offset ) {
    unsigned long block =
            MEMORY_STARTING_PAGE + file16_offset * ( (unsigned long)
32768 );
    unsigned long block_end = block + 65536; // <- 32 MB
```

```
    //unsigned long block_end = block + 5; unsigned char i = 0; // <--
DEBUG
    SD_TYPE sd_stat = SD_ERR;
    // LED1 goes on during reading
    led_debug_1_on();
    do {
        sd_stat = sd_startMultiBlockRead( block );
        if( sd_stat == SD_ERR_RETRY || sd_stat == SD_CONTINUE_WITH_ERR
||
                sd_stat == SD_INVALID_CMD || sd_stat ==
SD_ERR_SEND_STOP ) {
            error_report( ERROR_CODE_MEMORY_START_MULTI_READ );
            return RETURN_DEVICEERR;
        }
    } while( sd_stat != SD_CONTINUE );
    for( ; block < block_end ; block++ ) {
        do {
            sd_stat = sd_readBlock( (unsigned char *) &
page_buffer_buffer[0][SD_PAGE_BUFFER_DATA_START_OFFSET] );
            //sd_stat = sd_readBlock( (unsigned char *) &
page_buffer_buffer[i][SD_PAGE_BUFFER_DATA_START_OFFSET] ); // <-- DEBUG
            if( sd_stat == SD_ERR_RETRY || sd_stat == SD_ERR_RESTART
                    || sd_stat == SD_ERR_SEND_STOP || sd_stat ==
SD_INVALID_CMD ){
                if( sd_stat == SD_ERR_RESTART ) {
                    error_report( ERROR_CODE_MEMORY_START_MULTI_READ );
                } else {
                    error_report( ERROR_CODE_MEMORY_READ_BLOCK_MULTI );
                }
                return RETURN_DEVICEERR;
            }
        } while( sd_stat != SD_CONTINUE );
        //i++; // <-- DEBUG
    }
    do {
        sd_stat = sd_stopMultiBlockRead();
        if( sd_stat == SD_ERR_RETRY || sd_stat == SD_CONTINUE_WITH_ERR
|| sd_stat == SD_INVALID_CMD ) {
            if( sd_stat == SD_CONTINUE_WITH_ERR ) {
                error_report( ERROR_CODE_MEMORY_READ_BLOCK_MULTI );
            } else {
                error_report( ERROR_CODE_MEMORY_STOP_MULTI_READ );
            }
            return RETURN_DEVICEERR;
        }
    } while( sd_stat != SD_CONTINUE );
    led_debug_1_off();
    return RETURN_OK;
}

/*!
 * @brief Perform a sustained write test on the memory.
 *
 * Use after SD initialization function.
 *
```

```
 * @warning This is to be used for testing only and should not be used
in release code.
 *
 * @warning This test may overwrite any data on the memory device.
 *
 * @return \ref RETURN_TYPE
 */
RETURN_TYPE memory_sustainedWriteTest( unsigned char file16_offset ) {
    unsigned long block =
            MEMORY_STARTING_PAGE + file16_offset * ( (unsigned
long) 32768 );
    unsigned long block_end = block + 65536; // <- 32 MB
    SD_TYPE sd_stat = SD_ERR;
    unsigned short i = 1;
    // fill buffer with 0xff so that a non-trivial value is written.
    for( i=1 ; i<513 ; i++ ) {
        page_buffer_buffer[0][i] = 0xff;
    }
    // LED1 goes on during writing
    led_debug_1_on();
    do {
        sd_stat = sd_startMultiBlockWrite( block );
        if( sd_stat == SD_ERR_RETRY || sd_stat == SD_CONTINUE_WITH_ERR
||
                sd_stat == SD_INVALID_CMD || sd_stat ==
SD_ERR_SEND_STOP ) {
            error_report( ERROR_CODE_MEMORY_START_MULTI_WRITE );
            return RETURN_DEVICEERR;
        }
    } while( sd_stat != SD_CONTINUE );
    for( ; block < block_end ; block++ ) {
        do {
            sd_stat = sd_writeBlockBuffer( (unsigned char *) &
page_buffer_buffer[0][0] );
            if( sd_stat == SD_ERR_RETRY || sd_stat == SD_ERR_RESTART
                    || sd_stat == SD_ERR_SEND_STOP || sd_stat ==
SD_INVALID_CMD ){
                error_report( ERROR_CODE_MEMORY_WRITE_BLOCK_MULTI );
                return RETURN_DEVICEERR;
            }
        } while( sd_stat != SD_CONTINUE );
    }
    do {
        sd_stat = sd_stopMultiBlockWrite();
        if( sd_stat == SD_ERR_RETRY || sd_stat == SD_CONTINUE_WITH_ERR
|| sd_stat == SD_INVALID_CMD ) {
            error_report( ERROR_CODE_MEMORY_STOP_MULTI_WRITE );
            return RETURN_DEVICEERR;
        }
    } while( sd_stat != SD_CONTINUE );
    led_debug_1_off();
    return RETURN_OK;
}
```