Hardware and High Data Speeds on the CINEMA CubeSat

David McGrogan

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-83
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-83.html

May 19, 2010

Acknowledgement

Thanks to my coworkers at SSL, to my advisor Kris Pister, and to Mom and Dad.

Hardware and High Data Speeds on the CINEMA CubeSat

by David Paul McGrogan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Kris Pister
Research Advisor

(Date)

* * * * * * *

Professor Robert Lin
Second Reader

(Date)

Table of Contents

Table of Figures

Table of Listings

Table of Tables

1. Introduction to the CINEMA CubeSat

CubeSats are small satellites that conform to a standard developed by Stanford and California Polytechnic University. Their mass can be up to 1 kg for the standard 10x10x10 cm (1U) model, placing them in the large picosatellite/small nanosatellite range. Other standard CubeSat sizes are 10x10x20 cm (2U) and 10x10x30 cm (3U), with 2 kg and 3 kg mass limits respectively.

CINEMA is a 3U CubeSat currently under development at the UC Berkeley Space Sciences Laboratory in partnership with the Imperial College London, Kyung Hee University, and the Inter-American University of Puerto Rico. Funded by an NSF grant, its mission is to monitor space weather using newly developed miniaturized sensors -- a particle detector and magnetometers. This CubeSat is distinguished from many CubeSats developed by others in two major ways. First, CINEMA will need to handle large quantities of data due to the potential glut of information generated by the particle detector, which may approach 2 megabits per second during peaks of particle flux. Second, CINEMA will gather useful scientific data despite being built largely by students; many student-built CubeSats have been technology test platforms or simple radio beacons, built as a learning experience [Glaser 2009, Thomsen 2009]. CINEMA's projected launch date is in late 2011.

I joined the CINEMA team near the end of the high-level design phase of the project. At this point the sensors, form factor, avionics bus, part layout, power supply, communications link, mission parameters, and software compartmentalization had been established. However, the processor and the operating system had not yet

been chosen. My task was to assist in these decisions, define the data connections between components, and begin work on the flight software.

2. Hardware Design

2.1. Peripherals

The most significant pre-established elements of CINEMA's design were the peripherals with which the processor would need to communicate. Defined by the mission specification, they are:

- STEIN particle detector, capable of detecting the energy and charge polarity of incoming particles below 40 keV and energy alone below 100 keV. Using semiconductor detector pixels, it is an order of magnitude smaller than the electrostatic analyzer instrument used in the Wind spacecraft. Incorporates an FPGA, which is communicated with using the SPI bus. Maximum data rate is 16 bits/particle * 30,000 particles/sec/pixel * 4 pixels = 1.92 Mbps.

- Magnetometers (MAGs) for establishing satellite orientation and taking scientific data. Three axes per sensor unit, with one unit inside the satellite's body and one unit on the end of a 1m boom to reduce magnetic interference from onboard electronics. Only one of the two 3-axis magnetometer units is read at a time by the controlling FPGA, which connects via the SPI bus. Data rate is 19 bits/sample * 10 samples/sec/axis * 3 axes = 570 bits/sec, plus one byte/sec for a temperature reading for a total of 578 bits/sec.

- SD card, providing bulk data storage of scientific and engineering data for later downlink. Sized 2 GB, the maximum size for a standard SD (as opposed

to SDHC) card. Can communicate either in SD or SPI mode; SD mode is faster but has no freely released specifications, so SPI mode was chosen. Data is manipulated in blocks, with up to 512 bytes in a block. Maximum data rate varies with the applied clock rate; in SPI mode, one bit is exchanged per clock cycle. Maximum clock rate for SD cards is 25 MHz in either mode.

- Radio uplink, which receives commands from the ground station. Uses a serial connection with a data rate of 9600 bps.

- Radio downlink, which transmits responses to commands including engineering data and scientific observations. Uses a simple bitstream connection with a data rate of 1 Mbps, but is interfaced via an internally designed FPGA which performs Reed-Solomon encoding for data correction, so any sufficiently capable data connection could be used.

- Electrical power system (EPS), a system of batteries and electronics to provide electricity for all onboard systems. Has a diagnostics/command port that uses the I^2C bus. As an I^2C slave device, its data rate is determined by the master device, which is the processor. Its maximum data rate is 400 kilobits/second.

2.2. Processors

The processor selection was limited to those provided on daughterboards compatible with the CubeSat Kit Motherboard, a commercially provided foundation for CubeSats that provides a serial interface, a communications bus, an SD card connector, a real-time clock, remove-before-launch and deployment switches, and

other useful components. Only six daughterboards are available from the kit vendor, which conveniently limited our options from the entire universe of microcontrollers to a reasonably sized set. The main differences among the processors were their processing power and their available I/O ports.

Table 1: Properties of Available Processors

Processor	Archi-tecture	Memory	ADC	Instruction Clock	Integrated Peripherals
MSP430F1612	16-bit	55KB program memory, 5KB on-chip SRAM	8-channel 12-bit	7.3728 MHz	2 SCIs (UART0/SPI0/I2C & UART1/SPI1)
MSP430F1611	16-bit	50KB program memory, 10KB on-chip SRAM	8-channel 12-bit	7.3728 MHz	2 SCIs (UART0/SPI0/I2C & UART1/SPI1)
MSP430F2618	16-bit	116KB program memory, 8KB on-chip SRAM	8-channel 12-bit	7.3728 MHz	2 USCI_A (UARTA0/SPIA0 & UARTA1/SPIA1) 2 USCI_B (I2CB0/SPIB0 & I2CB1/SPIB1)
C8051F120	8-bit	128KB program memory, 8.5KB on-chip SRAM	8-channel 8-bit 500ksps	External xtal	2 UARTs, 1 SPI, 1 I2C
dsPIC33FJ256GP710	16-bit	256KB program memory, 30KB on-chip SRAM	32-channel 10/12-bit 1.1/0.5 Msps	10 MHz	2 UARTs, 2 SPIs, 2 I2Cs, 2ECANs, DMA
PIC24FJ256GA110	16-bit	256KB program memory, 16KB on-chip SRAM	16-channel 10-bit 500ksps	16 MHz	4 UARTs, 3 SPIs, 3 I2Cs, Parallel Master Port

My greatest body of microcontroller experience was with PICs, with some brief contact with the MSP430 series. The latter's claim to fame is their extremely low power needs; all three MSP430 options would run on 10 mW, an attractive property when all power must originate from solar panels. However, as long as the 100 mW power budget for the processor was not exceeded, any option was acceptable. (This put limits on the speed of the 8051 and dsPIC33 parts, as higher frequencies demand more power.)

The ultimate processor selection depended on the communication demands of the peripherals. Due to the high data rates anticipated on both the STEIN-to-processor and SD card-to-processor links, I assumed that at least two SPI ports would be needed, one for each connection during data transfer. The uplink radio had a serial

output, so clearly a serial UART would be needed. An I^2C port to interface with the EPS would be useful, but not necessary, as the data rate would be sufficiently low to allow software implementation of the protocol. This eliminated half of the options immediately; the MSP430F2618 and the PICs were still viable options.

The PICs had two distinct advantages over their competitor. First, there were modules available in MATLAB to allow the creation of PIC binaries, which was attractive to the people developing the Attitude Control System (ACS) software that would detect and reorient the satellite's spin. Second, and more importantly, the manufacturer of PICs, Microchip Inc, provides a free development environment that includes a cycle-accurate simulator. With this simulator, the execution speed of the instructions which comprise the ACS code could be tested, demonstrating that the PICs were up to the task. I was also able to use the simulator to demonstrate the capabilities of the PICs in general, such as serial communication and DMA. These simulations and demos, in addition to its clearly reduced code and data memory, removed the MSP430F2618 from consideration.

Finally, a decision had to be made between the two PICs. While the higher clock rate and three SPI ports on the PIC24 were appealing, I ultimately endorsed the dsPIC33 due to its unique capacity for DMA. I anticipated that the flexibility gained from DMA would be well worth the loss in speed and I/O. The CINEMA team found no fault with my assessment, so we incorporated the dsPIC33 into the project design.

The dsPIC33 incorporates some additional features that are useful in a spaceborne design. For example, it can reprogram itself while running, which permits

reprogramming after launch. It has automatic brownout-induced reset, which might save the mission after a power failure or allow it to operate for a little longer as the batteries wear out. It also has a number of power-saving features, from a selection of low-activity run modes to the ability to turn off power to unused subsystems on the chip.

2.3. High-speed data transfer mechanisms

2.3.1. SPI

Not all communications buses are capable of sustaining megabit speeds. I^2C, for example, has a 400 kbps limit in "high-speed" mode. Serial ports on the dsPIC33 are capable of reaching above 1 Mbps if pushed. The most capable interface available, however, is SPI [Microchip, 2009].

SPI behaves like a two-word shift register shared between two devices -- with every pulse on the clock line sent from the master device to the slave device, the MSb of each device's shift register is moved to the LSb of the other device's shift register. After as many clocks as there are bits in a word, the two devices have completely exchanged the contents of their shift registers.

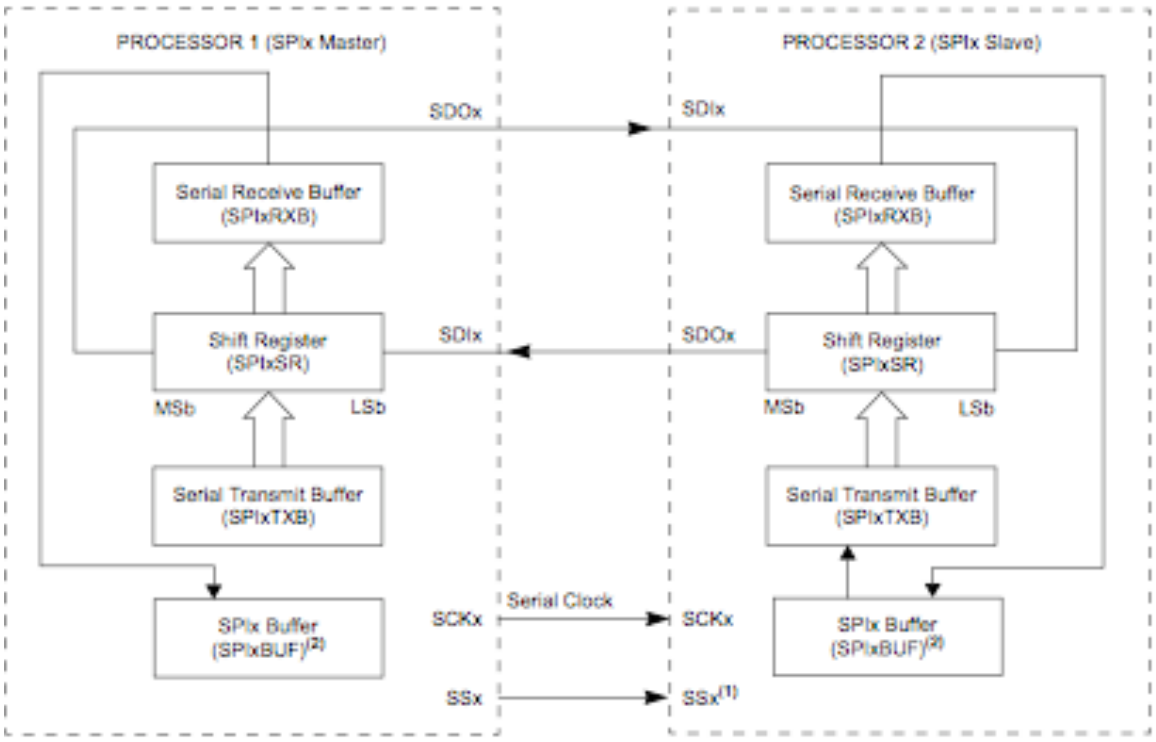

Figure 1: SPI communication between two devices [Microchip, 2009]

On the dsPIC33, the SPI bit rate is limited to half of the instruction clock rate, and for CINEMA this limit is 5 Mbps. This is more than enough to support the full 1.92 Mbps peak input from STEIN, even given significant overhead. It is also well within the abilities of the SD card, which could reach 25 Mbps if the processor allowed.

2.3.2. DMA

Direct Memory Access is unique to the dsPIC33 among the available processors. It comprises a set of "DMA channels" which can be configured to automatically move data from memory to a peripheral's buffer, or vice-versa, as needed without any ill effect on the CPU's activity. For example, the serial port could be configured to automatically output the contents of a section of memory. After each byte was sent,

the DMA channel assigned to the task would move the next byte from memory to the transmit buffer of the serial port [Microchip, 2009].

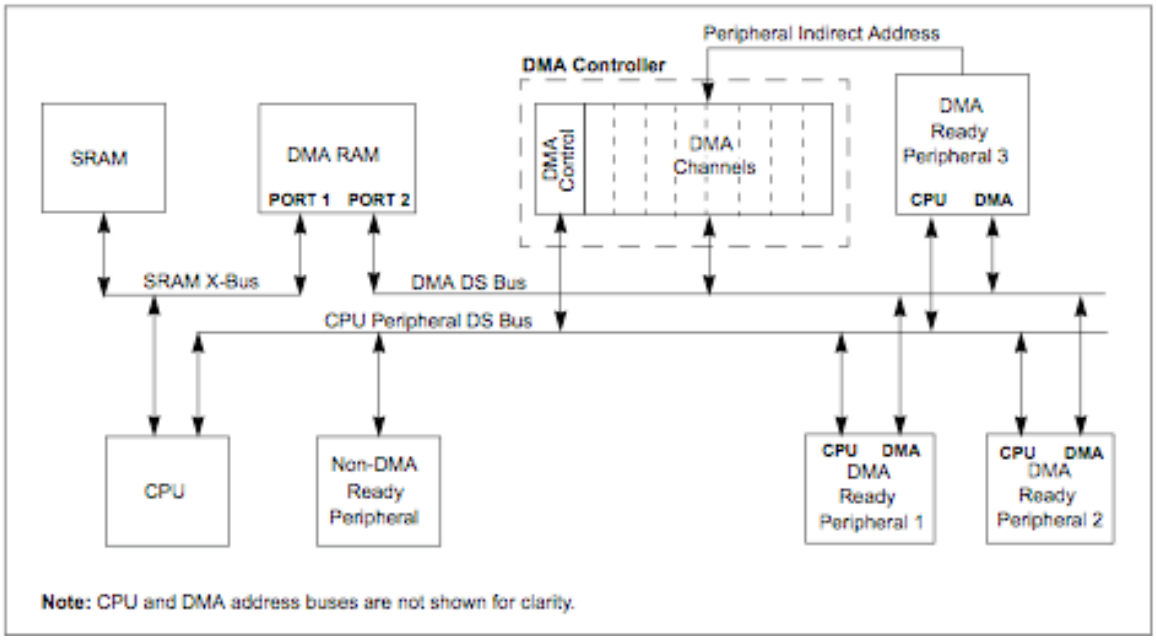

Figure 2: DMA system in the dsPIC [Microchip, 2009]

DMA is flexible and useful in many ways, particularly when handling multiple simultaneous data streams, but it is limited by the amount of DMA-capable RAM in the microcontroller; of the dsPIC33's 30 KB of RAM, only 2 KB is DMA RAM, so it must be used judiciously by the programmer. Another limitation is that only one device may access a given word of memory at a time. The CPU has first priority, followed by the first DMA channel, then the second, and so on down to the eighth. Data can conceivably be lost if a DMA channel, configured to write a peripheral's incoming data to RAM, is constantly blocked by other accesses to that location until another datum arrives at the peripheral and overwrites its buffer. However, this is very unlikely to happen without intentionally being caused, and should not occur at

all on CINEMA due to the relative infrequency of activity on any given DMA channel - at maximum, only one cycle in 16 will trigger the channel, which is the case for an SPI port. (The SPI ports transfer 8 bits at a time, but at half the rate of the instruction clock, hence there will be 16 cycles between each DMA read or write on an SPI buffer.)

2.4. Data flow

As with the processor itself, the connections between the peripherals and the processor were determined by the peak data rates of the peripherals. The primary demands on data flow were the STEIN detector, which could emit 1.92 Mbps at peaks, and the downlink radio, which demanded a constant 1 Mbps during transmission. These two peripherals would never be active simultaneously, but both would require the use of the SD card when active. This suggested that one of the two SPI ports be shared between STEIN and the downlink, and the other SPI port be dedicated to the SD card.

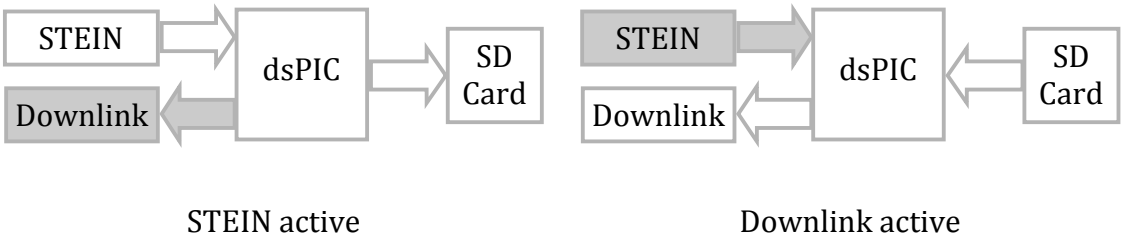

Figure 3: Communication through a shared SPI port

The other major peripherals, namely the magnetometers, EPS, and uplink radio, were simple to add. The MAG FPGA generated only 578 bits per second, which could be easily multiplexed with STEIN and the downlink on the shared SPI connection, as its data too would need to be written to the SD card. The EPS was the

only device using an I²C connection, so it received its own port on the processor. Similarly, the uplink radio was the only device requiring a serial connection.

At this point the team decided to use a single FPGA to both communicate with STEIN and operate the MAGs. This was a trivial change, as both STEIN and MAGs would still use the same SPI port on the processor. The final data flow layout plan was as follows; the arrows show the primary direction of information transfer, neglecting most control flow.

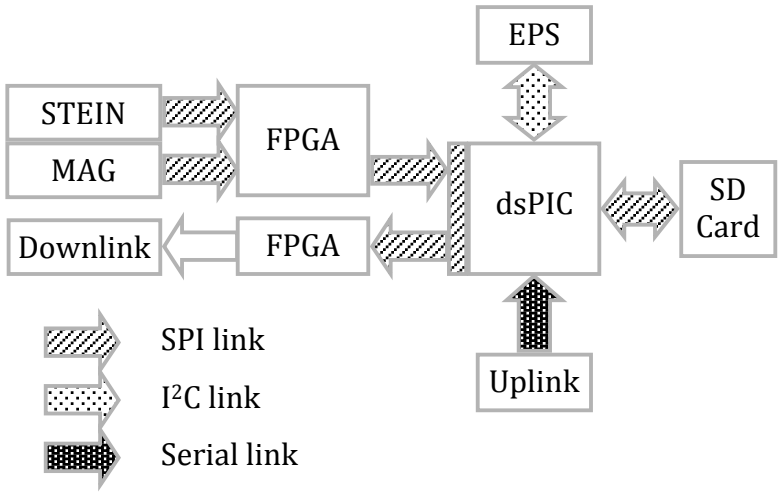

Figure 4: Data flow layout

3. Software

The overarching design principles of the CINEMA flight software are that it be flexible, reusable, well written, and debuggable. Besides being good general practice, flexibility and reusability are required because at least four more CubeSats are planned to use this software as a base, and the fewer changes and surprises involved in the modifications, the better. The code must be well written for similar reasons; because the programmers are students, most or all of the original

programmers will be gone when the next satellite is being programmed. Ease of comprehensibility of the code will therefore be significant for future "generations" of programmers. Finally, the code must be debuggable to speed development and make it more accessible to non-advanced programmers.

3.1. Operating system

One of the biggest software design questions was whether to use a real-time operating system (RTOS). A simple RTOS called Salvo was available from the CubeSat Kit vendor, with a free demo that I was assigned to scrutinize. By using Microchip's free simulator I was able to superficially verify that it worked, but I was concerned about the effects an RTOS might have on debuggability. Having written multithreaded applications in the past, I presumed that a program using an RTOS, which includes similar processor-yielding and semaphore mechanisms, would be similarly more difficult to debug. An alternative scheme was proposed by one of the more experienced team members. This scheme, used in a previous satellite, allowed all tasks to execute on a predictable, fixed schedule without the use of an RTOS.

This approach was to divide all of the processor's tasks into background tasks and foreground tasks. Background tasks would run on a tight, predefined schedule and would generally be maintenance tasks, such as taking science data and checking the status of the power supply. Foreground tasks would run in the remaining time with no particular time constraints and would include lengthy calculations for attitude adjustment and commands transmitted from the ground. The task types would have independent schedulers. The foreground scheduler would prioritize uploaded commands, but run everything else round-robin. The background scheduler would

be interrupt-driven - every 1/1024th of a second, a timer interrupt would call the scheduler, which would run the next task in a table of background tasks.

3.1.1. Background task scheduling

Each background task must be executed frequently enough to satisfy the demands of its peripheral's data flow. For a task like HSK, which samples analog voltages from the dsPIC33's pins, this is trivial because the task itself controls the creation of data. However, a task like STE or TX has significant constraints. At peak flux, the STEIN detector is capable of generating 1.92 megabits per second, which the processor must absorb and relay to the SD card. The transmitter must be fed 1 megabit per second to avoid wasting precious downlink time. Are these tasks schedulable?

The tasks with the largest data rates must interact with the SD card, and every transaction with the SD card has some overhead. To minimize this, we use the maximum SD data block size, which is 512 bytes. At 1.92 Mbps, STEIN generates 469 blocks per second. Only one block can be sent in a time-slice of 1/1024th of a second, so STEIN's handler task STE must occupy at least 469 of 1024 time-slices. This is rounded up slightly, such that half of all time-slices belong to the STE task. Similarly, in order to send 1 Mbps to the downlink transmitter, the TX task must handle 245 blocks per second. TX must therefore occupy 245 of 1024 time-slices, which is rounded up to one-fourth of all time-slices.

3.1.2. Scheduling the command handler

Compared to STEIN's 1.92 Mbps, the uplink radio's 9600 bps seems trivial. However, the microcontroller's link to STEIN incorporates an FPGA, which buffers

data until an entire data block is ready. The uplink's serial connection has no such luxury; there is a buffer built into the dsPIC33's serial port, but it holds only 4 bytes. A 9600 bps connection will fill this 300 times per second, so it must be emptied at 300 Hz. However, STE already has half of the 1024 time-slices, and TX has one-quarter. The task table cannot accommodate an additional task requiring 300 of 1024 slices.

There are a few ways to fix this problem. Commands might be handled in the scheduler, which runs at 1024 Hz. However, this would take time away from every task, including tasks such as STE, which has little time to lose. Alternatively, because STEIN and the downlink are never used simultaneously, their tasks STE and TX might be joined into one. This would leave half of the time-slices available, but also present a new problem:

	STE/TX		STE/TX		STE/TX

The STE/TX composite task runs in every other slice.

		CMD			CMD

The command handler task runs in every third slice.

	STE/TX	CMD	STE/TX		?????

A conflict exists!

The conflict might be resolved by running CMD in every other slice instead of every third, but then there would be no time left for any other tasks. The other tasks would need to be incorporated into CMD, at the cost of the predictable timing of the task table.

My solution was to use DMA to replace the existing 4-byte buffer with a buffer in memory. Specifically, a DMA channel is configured to move each incoming serial byte to a section of DMA RAM. The destination address automatically increments with each byte moved; when the end of the buffer region is reached, the destination address returns to the beginning automatically. In this way, DMA emulates the data-storing half of a ring buffer, leaving the data-retrieving duty for the command handler. Because the DMA channel operates independently of the CPU, it's as independent of the current task as the original 4-byte buffer. All that is necessary is to make the DMA buffer large enough to store as much information as might arrive between executions of the command handler; if the handler runs at 16 Hz (one slot in the above table), the buffer must be at least 75 bytes long, which is only a small portion of the 2 KB of DMA RAM available on the dsPIC33.

Table 2: Table of Background Tasks

BKG Module 1024 Hz Table

	0	1	2	3	4	5	6	7
0		STE	TX	STE		STE	TX	STE
8	HSK	STE	TX	STE		STE	TX	STE
16	TM	STE	TX	STE		STE	TX	STE
24	CMD	STE	TX	STE		STE	TX	STE
32		STE	TX	STE		STE	TX	STE
40	PWR	STE	TX	STE		STE	TX	STE
48	SSR	STE	TX	STE		STE	TX	STE
56	MAG	STE	TX	STE		STE	TX	STE

This table describes .063 seconds and is repeated 16 times per second.

BKG Module Function Calls Distributed in Time Phase

FN	FREQ	Description
TX	256	Transfer data from SDCARD to Transmitter
STE	512	Transfer data from STEIN to SDCARD
MAG	16	Transfer data from MAG to SDCARD
HSK	16	Housekeeping A/D Sampling

CMD	16	Uploaded Command Handling
PWR	16	Power Monitoring 16-bit timing
SSR	16	Solid State Recorder Manager
TM	8	Packet Multiplexing
BKG0	8	Available for future use
BKG1	32	Available for future use
BKG2	128	Available for future use

3.2. Development platform

For ease of development and insurance against accident, two sets of the CubeSat Kit development board and dsPIC33 daughterboard were purchased. The development board is electrically identical to the flight motherboard, but with additional components built in to simplify development, such as indicator LEDs, a standard 9-pin RS-232 serial port, and additional lateral space for peripheral boards. One set of hardware was used for development of the ACS software, while I used the other set to begin system software development.

Figure 4: CubeSat Kit Development Board [Pumpkin Inc, 2010]

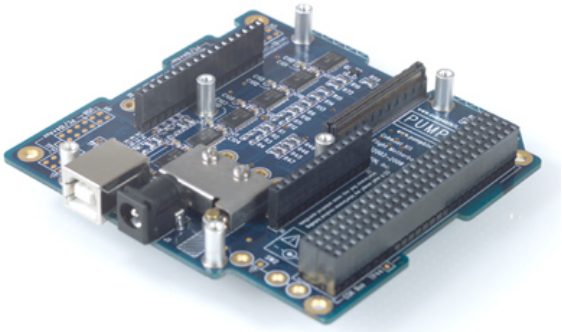

Figure 5: CubeSat Kit Flight Motherboard [Pumpkin Inc, 2010]

One foible of the dsPIC33 is that it possesses three different pin pairs that may be used for in-circuit debugging (ICD). Although it may be programmed using any of the three pairs, it will only accept ICD commands from one pair during execution. This increases flexibility (what if the functionality you wanted to test used a port

that shared the only provided ICD pins?), but it came as a surprise to me. Naturally, the default pin pair was not the pair connected to the programming header on the dsPIC33 daughterboard, and therefore the "run" command I tried to issue fell on deaf ears. I had to investigate several datasheets in order to discover, first, the nature of the problem, and second, the line of C code needed to make the compiler activate the correct pin pair.

3.3. The SD Card

After the microcontroller and operating system were decided upon, I became leader of the students developing the flight software. I took it upon myself to write the interface for the SD card, as it would most likely be among the most timing-critical pieces of code due to the high data rates involved. I was experienced in writing assembly for PICs, so I could write code at the lowest level if the SD card's timing made it necessary.

Secure Digital (SD) flash memory cards are a favorite of electronics hobbyists because they present a simple physical interface with few electrical connections while providing gigabytes of available storage. They also have a freely available interface protocol specification for the SPI mode [SD Group, 2006]. For the same reasons, the CubeSat Kit motherboard incorporates a slot for an SD card and a buffer amplifier to isolate it from the processor [Pumpkin, 2009].

3.3.1 SD cards in space

For an electronic item to travel safely into space, it must be resistant to vibration, shock, extreme temperatures, and radiation. I sought a high-durability SD card from

a mainstream manufacturer, so that we could purchase and test multiple units without costing more than a few hundred dollars. Eventually I discovered the SanDisk Extreme III line of cards, which SanDisk claims will keep operating under vibration, shock, and temperatures from -25°C to 85°C [Amazon.com, 2010]. Oddly, nobody seems to know how modern SD cards behave when bombarded with radiation. Nguyen et al. measured radiation's effects on flash RAM, but used cards from 1999 with a capacity of 128 MB. More modern cards with smaller features and higher bit densities may be more vulnerable than the tested cards, which exhibited a significant (0.7 milliamp) increase in standby current and functional failure after receiving a cumulative dose of 8000 rad(Si) while powered on [Nguyen et al., 1999]. In a low Earth orbit with high inclination (20 to 85 degrees), which is the type of orbit CINEMA is intended to inhabit [Curtis, 2008], typical radiation doses are 1000-10,000 rad(Si) per year [NASA, 1996]. Assuming that modern flash RAM is twice as vulnerable as the flash tested in 1999, I conjecture that CINEMA's solid-state memory should be viable for at least 2/5 of a year (~21 weeks) in the worst case. This is on the same order of magnitude as CINEMA's expected mission time of one year, and should be more than enough time to demonstrate the viability of the concept and take scientific data, barring some unforeseen calamity that must be handled.

3.3.2. SD card basics

The freely available specification provided the details of initializing, commanding, reading, and writing the SD card. The card uses the SPI interface in a half-duplex manner; despite the full-duplex nature of SPI, the processor and SD card never

transmit valid information simultaneously. After a command, the processor must repeatedly transmit null bytes with all bits set until the SD card returns its response.

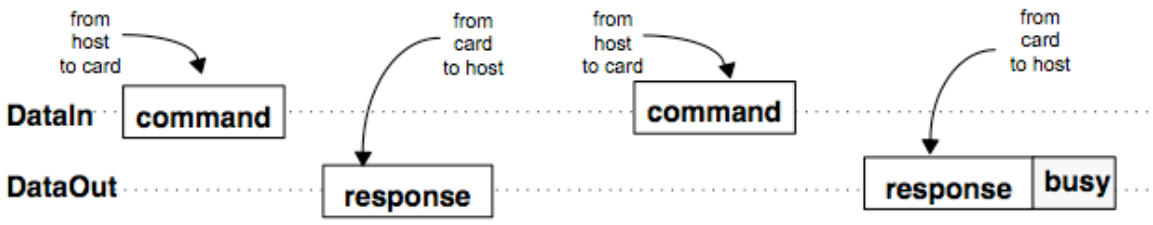

Figure 6: SD Card Command Interaction [SD Group, 2006]

The command format is fairly simple. It consists of six bytes, the first of which contains a two-bit header and a six-bit command number. The middle four bytes compose the argument of the command, and the last byte contains the seven-bit CRC (cyclic redundancy check) of the previous 5 bytes with a one-bit trailer.

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

Figure 7: SD Card Command Format [SD Group, 2006]

My first concern was to find out how much overhead existed in reading and writing, as the overhead might limit how much could be done in a scheduled time-slice. As SPI could transfer a given number of bytes in a time-slice, I measured time in bytes.

Table 3: Time Requirements of SD Card Data Retrieval and Storage

Read phase	Time/data		Write phase	Time/data
Command	6 bytes		Command	6 bytes

Response delay	0-8 bytes		Response delay	0-8 bytes
Response	1 byte		Response	1 byte
Access time	1 byte + t_{ACCESS}		Pause	1 byte min.
Start block	1 byte		Data	512 bytes
Data	512 bytes		CRC of data	2 bytes
CRC of data	2 bytes		Data response	1 byte + t_{BUSY}
Total (max)	531 bytes + t_{ACCESS}		Total (max)	531 bytes + t_{BUSY}

A single time-slice is $5{,}000{,}000\frac{\text{bits}}{\text{sec}} \times \frac{\text{byte}}{8\text{ bits}} \times \frac{\text{sec}}{1024\text{ time-slices}} = 610.35\frac{\text{bytes}}{\text{time-slice}}$

long, which means that there is a maximum of 610 - 531 = 79 bytes left for other activities in a time-slice. For example, after a write is completed, it is good practice to make sure there were no errors by using another command to find out how many bytes were just written. However, as long as the access and busy times weren't too large, I wasn't very worried about running out of time in a slice.

3.3.3. CRC in limited time

What did worry me were the CRC fields in the command and data operations. A command was given for disabling the data's CRC16, but it didn't mention the commands' CRC7. Arbitrary commands had to be possible to allow arbitrary read/write addresses, which meant a CRC7 routine had to be made. Standard CRC algorithms operate on a single bit at a time, XORing or not XORing a given value to the evolving CRC value depending on the state of the input bit. This was not acceptable for CINEMA, as each bit was transferred in two instructions' worth of

time, and the standard algorithm could not be reduced to two instructions on the PIC architecture. I looked for faster CRC7 algorithms online, but found nothing useful excepting a demonstration of CRC5 for an FPGA that used nibbles of input together with the entire current CRC to form a parallel-computable expression for bits of the next CRC [Evgeni, 2009]. I figured that I could use similar techniques to produce a lookup table-based version of CRC7.

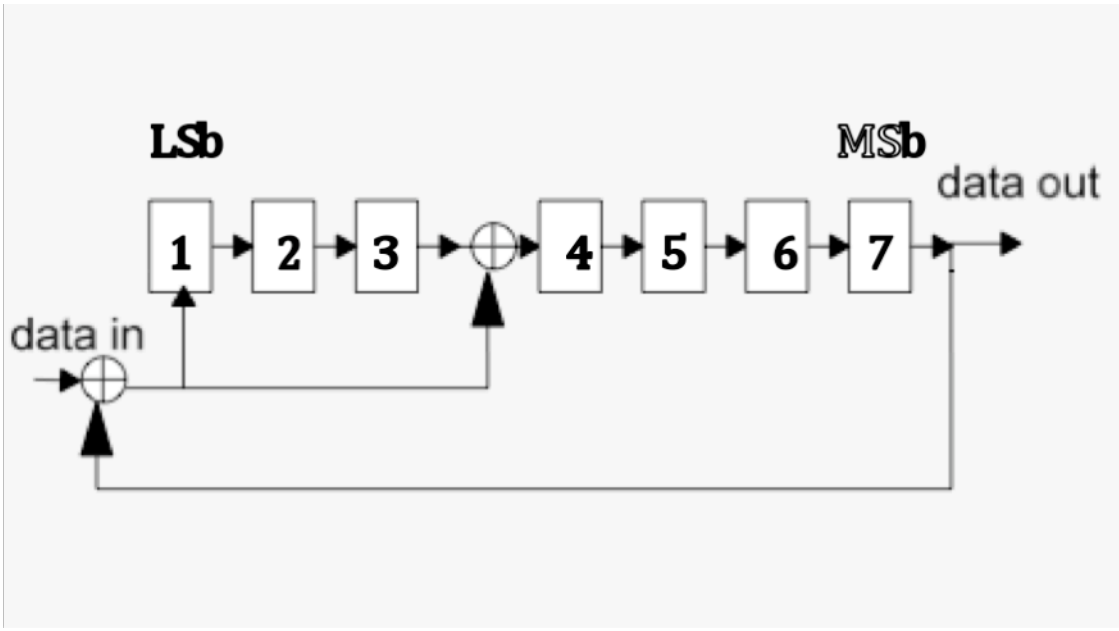

Figure 8: CRC7 Equivalent Generator [SD Group, 2006]

Observing the diagram of a CRC7 generator provided with in the SD card specifications, I noticed that the CRC bytes move in a loop -- all the bits move up by one, and bit 7 is XORed with the input bit to create the new bit 1 (and affect bit 4). This suggested that bit-shifting by 4 might play a significant role in my algorithm.

Calculating 4 generations of the CRC7 generator resulted in the following (Dx represents the value on the data input immediately before generation x):

generation	cells						
0	1	2	3	4	5	6	7
1	D1^7	1	2	3^D1^7	4	5	6
2	D2^6	D1^7	1	2^D2^6	3^D1^7	4	5
3	D3^5	D2^6	D1^7	1^D3^5	2^D2^6	3^D1^7	4
4	D4^4	D3^5	D2^6	D1^7^D4^4	1^D3^5	2^D2^6	3^D1^7

Table 4: The State of a CRC7 Generator Over 4 Generations

I noticed that the values of bits 1, 2, and 3 could be shifted to their position in the 4th generation, but as for the rest I initially followed the example of the CRC5 generator, making a table for all potential values of bits D1-D4 and bits 7-4. The tables were identical, pointing out the inherent pairings: D4 was always XORed with bit 4, D3 with bit 5, D2 with bit 6, and D1 with bit 7. This is especially convenient because D1 and bit 7 are both the most significant bits of their categories, so the two highest nibbles can be XORed in place. The final algorithm for CRC7ing one byte at a time is as follows; all variables are actually registers, and the initial value of crc is 0.

1. temp = XOR(input,crc) // begin high nibble
2. temp = AND(temp,0xf0) // four bit-pair XOR values ready
3. temp2 = RIGHT SHIFT(temp,3)
4. temp = XOR(temp,temp2) // eight bit-pair XOR values ready
5. crc = LEFT SHIFT(crc,4) // unpaired values positioned
6. crc = XOR(crc, temp) // new CRC7 complete

```
7.  temp = LEFT SHIFT(input,4)        // begin low nibble

8.  temp = XOR(temp,crc)

9.  temp = AND(temp,0xf0)             // four bit-pair XOR values ready

10. temp2 = RIGHT SHIFT(temp,3)

11. temp = XOR(temp,temp2)            // eight bit-pair XOR values ready

12. crc = LEFT SHIFT(crc,4)           // unpaired values positioned

13. crc = XOR(crc, temp)              // new CRC7 complete
```

Listing 1: CRC7 Algorithm (dsPIC33 Assembly Language)

The least significant bit in the output byte is always 0, which is perfect for SD card purposes. Set that bit, and the result is the final byte of the SD command format.

The algorithm is 13 instructions long, leaving 3 instructions for getting the "input" byte from memory, copying it to the SPI port for sending, and reading the the SPI port's most recently received byte to prevent overflow. It just barely fits. This algorithm is made possible by the dsPIC33's multiple working registers; I'm not sure that it would be possible with only one working register, which was the case on PICs like the PIC16F84 I cut my teeth on.

Flush with success, I decided to try optimizing CRC16 as well, so that data transactions might also be verified. CRC16's generator is a bit more complex:

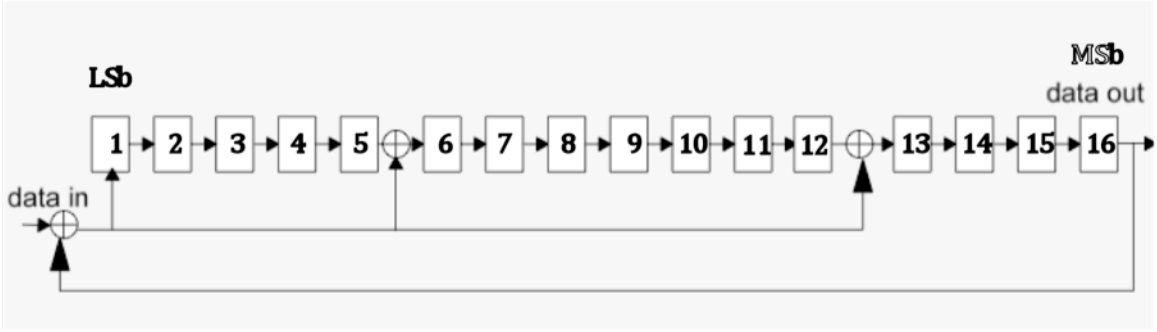

Figure 9: CRC16 Equivalent Generator [SD Group, 2006]

generation	cells			
0	1	2	3	4
4	D4^13	D3^14	D2^15	D1^16
8	D8^9^D4^13	D7^10^D3^14	D6^11^D2^15	D5^12^D1^16

generation	cells		
0	5	6	7
4	1	2^D4^13	3^D3^14
8	D4^13	D3^14^D8^9^D4^13	D2^15^D7^10^D3^14

generation	cells			
0	8	9	10	11
4	4^D2^15	5^D1^16	6	7
8	D1^16^D6^11^D2^15	1^D5^12^D1^16	2^D4^13	3^D3^14

generation	cells		
0	12	13	14
4	8	9^D4^13	10^D3^14
8	4^D2^15	5^D1^16^D8^9^D4^13	6^D7^10^D3^14

generation	cells	
0	15	16
4	11^D2^15	12^D1^16
8	7^D6^11^D2^15	8^D5^12^D1^16

Table 5: The State of a CRC16 Generator at 0, 4, and 8 Generations

As with CRC7, CRC16's bits move in a loop, and the bit pairs appear as expected; D1 is always XORed with bit 16, D2 with 15, and so on. Because there are more than 8

bits in CRC16, the computation can move on to the byte level and make even more bit pairs than at the nibble level. Using similar principles to CRC7, after an 8-bit shift all of the unpaired bits are in the right place. The remaining elements are in two continuous, unbroken groups: {D1^16 ... D8^9} and its subset {D1^16 ... D4^13}. I brought this line of thought to fruition, resulting in a 13-instruction algorithm as before. I then perused some CRC websites I found before making CRC7, and noticed an algorithm that took my results one step further [ckielstra, 2005]. There was really only one continuous group: {D1^16 ... D4^13, D5^12^D1^16 ... D8^9^D4^13}. This group appeared three times (once hanging off the most-significant edge), and made possible a 10-instruction algorithm. Again, all variables are actually registers.

1. SWAP(crc)	// exchange bytes; works like rotate left by 8, which is what's needed
2. crc = XOR.BYTE(crc,temp)	// high byte of crc must stay the same; I build my XOR group in crc's low byte
3. temp = RIGHT SHIFT(crc,4)	// no byte-only form exists, hence the next instruction
4. temp = AND(temp,0x000f)	
5. crc = XOR.BYTE(temp,crc)	// the low byte is now the magic XOR group, and is in the right place once
6. temp = ZERO EXTEND(crc)	// that is, temp = AND(crc,0x00ff), which isn't an available instruction

```
7.  temp = LEFT SHIFT(temp,5)

8.  crc = XOR(temp,crc)              // XOR group is in the right place twice

9.  temp = LEFT SHIFT(temp,7)

10. crc = XOR(temp,crc)              // XOR group is in the right place three times
```

Listing 2: CRC16 Algorithm (dsPIC33 Assembly Language)

(I later discovered that the SD command that makes data CRCs optional also applies to commands, so the entire exercise could have been skipped. I'm glad that I didn't know that, though, because I probably would have dropped the idea if I didn't think it had to work. Now that the algorithm exists, CINEMA can use it to resist errors.)

3.3.4. SD first contact

Using the specification, I designed routines to write the appropriate values to the SPI port to output the first command for SD card initialization and read back the response. Nothing happened, so I hooked up the oscilloscope to verify that the waveforms being emitted were as I expected. Nothing seemed wrong, but I knew my information was limited, so I looked for reference waveforms online. Two people had documented the SD card initialization process with included waveforms [ChaN, 2008; ESawdust, 2008], so I learned to change the idle state of the bus to a high state and to emit 10 bytes of high bits to properly initialize the SD card.

Proceeding in the card initialization sequence, I discovered that the SPI port on the dsPIC33 was very serious about error detection -- if a byte arrived before the previously received byte had been read by the program, the port set an overflow

flag and refused to accept any more bytes until the flag was manually cleared. This was generally unnecessary, as the SD card never returns valid data while the processor is sending a command, but it did make me a better programmer by pointing out when I insufficiently understood how my code was interacting with the SPI port.

The SD card has two main states: initializing and ready. Initialization must be done with a lower speed SPI clock, as the card operates in an open-drain mode until initialization is complete [SanDisk, 2002]. Up to this point, I had been driving the SD card at 312 kHz. Once full speed became available, commands and data were transferred at the dsPIC33's full speed of 5 Mbps. This immediately caused a rash of SPI port overflows, as the input buffer filled up more than 10 times faster, leading to a corresponding rash of improved code. Also, it had become possible to read from and write to the SD card, determining the values of the access time during a read and the busy time following a write.

3.3.5. SD write busy time

Starting with single-block writes to various locations, I discovered a surprising trend. When I set my program to write to a new address, the SD card was busy for 20 to 30 thousand bytes after the write. If I reran the program, the busy time dropped to between 650 and 900 bytes. This was distressing, as not only did it imply some sort of states existing within the SD card between operations, it also endangered the 1.92 Mbps minimum data rate we needed. Naively speaking, ignoring the constraints imposed by the background task table, the SD card could be busy for 851 bytes, maximum, and still sustain 1.92 Mbps.

I decided that a more accurate simulation of CINEMA operating conditions was needed. When I tried writing to a sequence of adjacent addresses, the first address caused a busy period of 20,000-30,000 bytes as before, but subsequent writes were busy for less than 600 bytes. Even better, rewriting the same set of addresses reduced the startup busy period to around 3000 bytes.

The data showed an interesting pattern, visible only when making multiple writes. After the initial twenty-thousand-byte busy delay, all writes had short busy periods except for one write in every 64, which had a busy delay 2800-4000 bytes long. Presumably this was caused by some internal element of the SD card, such as a 32 KB write buffer, but it had to be dealt with by the processor. To understand this pattern's effect on writing I took 490 data points from writes to two disparate address sequences and modeled their effects if all incoming STEIN data were stored in a single buffer. The simulated system checked the SD card's status at 512 Hz, and if the card were not busy and there were at least 512 bytes in the buffer the system would send a block from the buffer to the SD card.

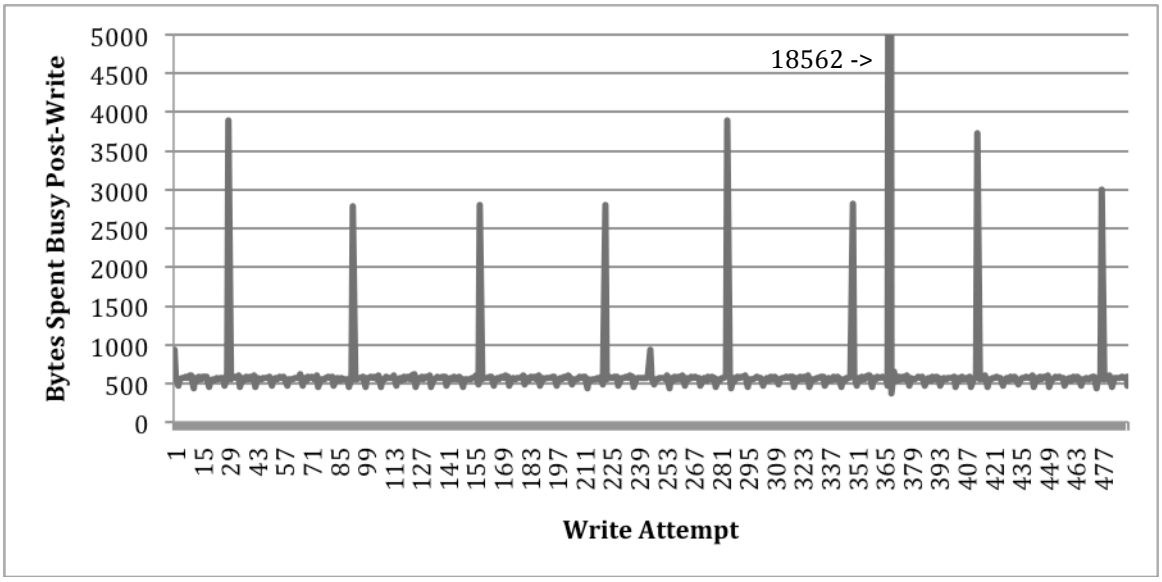

Figure 10: Busy Times Experienced in 490 Write Attempts

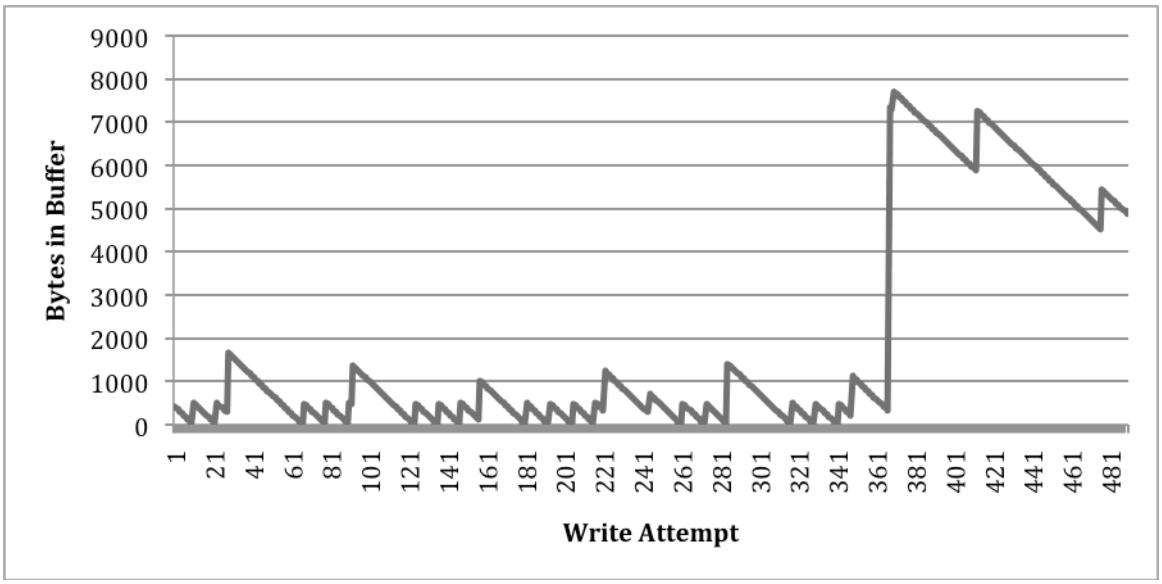

Figure 11: Contents of Simulated Buffer Following 490 Write Attempts

This simulation modeled the behavior of a potential addition to the actual CINEMA system, and produced encouraging results. In the buffer contents graph, the buffer only exceeds 2000 bytes due to an unusually long busy delay caused by switching to a new address region, and recovers steadily from even that. The large sawteeth are due to the one-every-64 long delays, and the small sawteeth occur when the buffer contains less than 512 bytes so there is no 512-byte block to send to the SD card. An

8 KB buffer was sufficient to deal with a 18,562 byte busy period; some busy periods slightly exceed 30,000 bytes, so a 12 KB buffer should be able to handle even those peaks. This buffer might be in the microcontroller, in the FPGA, or shared between the two; the FPGA will contain at least 1 KB of buffer in its present design. The dsPIC33 contains 30 KB of memory; whether or not a third of that can be spared for STEIN buffering is unknown because much of CINEMA's code is unwritten at this time. Also in question is whether it should be spared, as 1.92 Mbps is STEIN's peak rate, which we never expect to see and could never relay to the ground if it were frequent. Ultimately, if the memory is available, it will most likely be used.

In the name of science I tried increasing the address gap between writes. It appeared that there was a roughly logarithmic relation between busy period duration and proximity to the previous write (see figure). However, even moving to every-other-block writing pushes the maximum busy time up to 900 bytes, so CINEMA is effectively limited to writing adjacent memory locations.

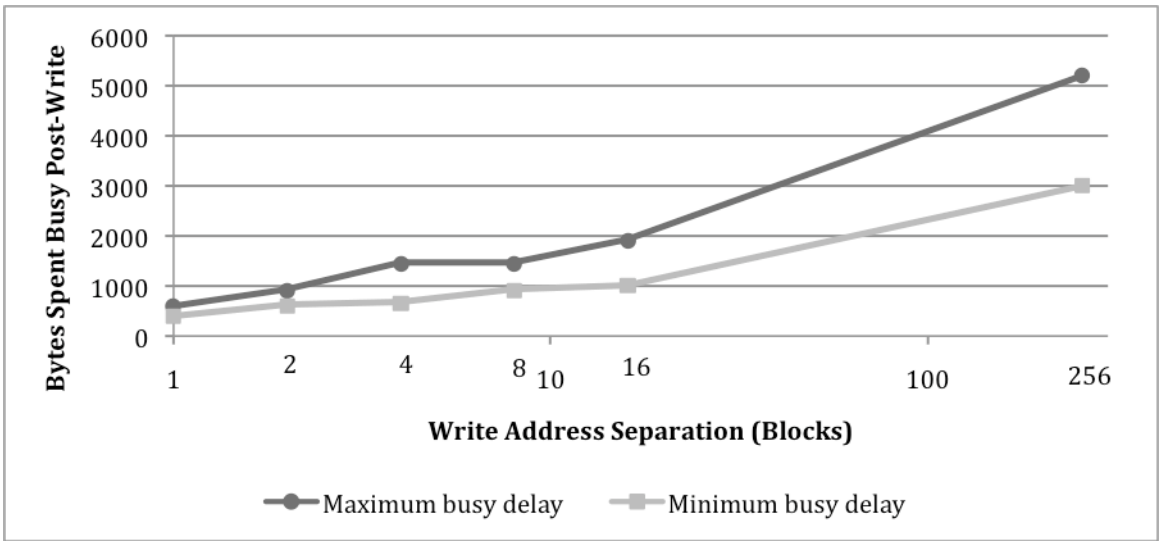

Figure 12: Relation of Busy Period Duration to Write Address Separation

Finally, I tried writing to two address sequences alternately, e.g. 10, 1000, 11, 1001, 12, 1002, ... to simulate a pair of ring buffers. Fortunately, there was no significant increase in busy period after these writes; it was indistinguishable from writing all the addresses in one sequence, then writing all the addresses in the other sequence. As a result, CINEMA can safely maintain a buffer for diagnostic data and records that is independent from the buffer for scientific data. In order to avoid the long busy delay that accompanies the first write to a memory region, the first addresses of both buffers should be written with dummy data well before the buffers are likely to be used.

3.3.6. SD read access time

After the complexities of the SD card's write behavior, I was bracing myself for strange fluctuations in the access times that occur during reads. Because they are present in the middle of reading instead of the end, long access delays might significantly complicate sending data to the transmitter. Desiring to see the worst possible case, I made randomized reads across the entire 2 GB address space. The result was a pleasant surprise. There were two long access delays in the beginning of the process, always exactly 484 and 480 bytes long respectively. (Usually they occurred on the first and second random reads, but once on the first and fifth.) All other access times were in the 72-76 byte range, with the occasional access time in the 52-56 byte range with no obvious distribution.

Because no access delay exceeds 484, any read operation can be handled in a single 610-byte-long time-slice if the initial read request is made in the first hundred bytes of time, but it will require the use of DMA to maintain the transfer after the time-

slice ends. This has an upside for foreground tasks, in that short access times can use the same DMA mechanism to end the background task early, leaving hundreds of bytes of time available for foreground task work. The algorithm is as follows:

1. If the downlink FPGA can hold another block, send SD read command.
2. Set up DMA channel 1: read from single DMA RAM location, write to SD card, move 514 bytes. /*The single location contains 0xFF, SD card's "null" byte.*/
3. Set up DMA channel 2: read from SD card, write to 514 bytes of DMA RAM, move 514 bytes.
4. Set up DMA channel 3: read from 514 bytes of DMA RAM, write to downlink FPGA, move 512 bytes. /*The downlink FPGA doesn't care about the CRC.*/
5. Manually send one 0xFF byte to the SD card.
6. If the response isn't the "block start" byte, go to 5.
7. Enable DMA channels 1, 2, and 3.
8. Activate DMA channel 1. SD card will begin sending the data block, which will be moved into DMA RAM by DMA channel 2.
9. After the first byte arrives via DMA channel 2, activate DMA channel 3.
10. Return from interrupt. All DMA channels will auto-disable when finished.

Listing 3: DMA-Driven Read From SD Card (Pseudocode)

3.3.7. SD read busy time

Something not mentioned in the SD card specification was the busy period that followed the read operations. Every single random read was followed by a delay of exactly 172 bytes. Their origin is uncertain but irrelevant, as they occur after the DMA mechanism finishes moving the data. However, the algorithm above sends no bytes to the SD card after the data block and its CRC16 are received; the SD card may require a clock input to finish whatever it's doing during the 172 bytes. If this is the case, a solution is to increase the number of bytes transferred by DMA channel 1 to 686 (514 + 172). This will cause a receive overflow on the SPI port connected to the SD card, but the only data lost will be "busy" bytes from the SD card. It will, however, be important to clear the overflow flag before the next communication with the SD card, as otherwise the incoming bytes will be dropped.

4. Conclusion

In my time at the Space Sciences Laboratory at UC Berkeley, I have chosen a processor, designed data flows, determined task frequencies, worked around constraints using DMA, and heavily interfaced and characterized an SD card. My work has been integral to the progress of the CINEMA project, and I leave it at a good break point. Future work in these areas might include radiation bombardment of a more modern SD card, and will certainly involve a higher-level interface to my SD card routines.

Appendix: Products Used

Flight motherboard: Pumpkin P/N 710-00484 Rev. D

Development board: Pumpkin P/N 710-00297 Rev. D

Processor daughterboard: Pumpkin P/N 710-00528 Rev. A

EPS: Clydespace 3U EPS, 30 Wh lithium batteries

Uplink radio: Helium 100

Downlink radio: Emhiser EDTC-01DEA

SD card: SanDisk 2 GB Extreme III SD Memory Card (SDSDX3-002G-A21)

Sources

Amazon.com. "SanDisk 2 GB Extreme III SD Memory Card (SDSDX3-002G-A21 Retail Package): Electronics." 2010. http://www.amazon.com/SanDisk-Extreme-Memory-SDSDX3-002G-A21-Package/dp/B000FKKWVM (January 28, 2010)

ChaN. "How to Use MMC/SDC." 2008. http://elm-chan.org/docs/mmc/mmc_e.html (March 8, 2010)

ckielstra. "CRC16, very efficient." 2005. http://www.ccsinfo.com/forum/viewtopic.php?t=24977 (March 9, 2010)

Curtis, David, and John Sample. "CubeSat: CubeSat for Ions, Neutrals, Electron, and Magnetic Fields (CINEMA)." 2008.

ESawdust. "SD/MMC SPI Initialization Waveforms." 2008. http://www.esawdust.com/blog/serial/files/SPI_SD_MMC_ATMega128_AVR.html (March 4, 2010)

Evgeni. "OutputLogic.com >> Parallel CRC Generator." 2009. http://outputlogic.com/?p=158 (March 1, 2010)

GHS Infotronic. "Online CRC Calculation." https://www.ghsi.de/CRC/index.php (March 15, 2010)

Glaser, David L. "Mechanical System Design of the CubeSat for Ions, Electrons, Neutrals, and Magnetic Fields Mission (CINEMA)". 2009.

Jakacki, Peter. "SPI / SD initialization sequence - my way." 2005. http://www.embeddedrelated.com/groups/lpc2000/show/9497.php (March 8, 2010)

Microchip Inc. "dsPIC33FJXXXGPX06/X08/X10 Data Sheet." 2009. http://ww1.microchip.com/downloads/en/DeviceDoc/70286C.pdf (Sept. 11 2009)

NASA. "Space Radiation Effects on Electronic Components in Low Earth Orbit. 1996. http://klabs.org/DEI/References/design_guidelines/design_series/1258jsc.pdf (Oct. 5, 2009)

Nguyen, D.N. et al. " Radiation Effects on Advanced Flash Memories." 1999. http://parts.jpl.nasa.gov/docs/Flash-99.pdf (Oct. 5, 2009)

Pumpkin Inc. "CubeSat Kit Motherboard (MB)." 2009. http://www.cubesatkit.com/docs/datasheet/DS_CSK_MB_710-00484-D.pdf (Sept. 22, 2009)

SanDisk. "Host Design Considerations: NAND MMC and SD-based Products." 2002. http://www.sandisk.com/media/216454/appnotemmc_sdv1.0.pdf (April 26, 2010)

SD Group. "SD Specifications Part 1 Physical Layer Simplified Specification." 2006. http://www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf (Sept 22, 2009)

Thomsen, Michael. "Michael's List of Cubesat Satellite Missions." 2009. http://mtech.dk/thomsen/space/cubesat.php (October 9, 2010)

UltimaSerial. "Factors affect SD and MMC's data throughput rate." 2010.

http://www.ultimaserial.com/sandisk.html (Apr 8, 2010)