

CUSAT TEAM: TELEMETRY & COMMAND SUBSYSTEM
MEMORY MANAGEMENT AND FLASH INTERFACE

A Design Project Report
Presented to the Engineering Division of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)

by
Tyler Dolen
Project Advisor: Professor Bruce Land
Degree Date: January 2007

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: CUSat Team: Telemetry & Command Subsystem Memory Management and Flash Interface

Author: Tyler Dolen

Abstract:

This project involves my work as a member of the CUSat team on campus, Cornell's entry into the Nanosat-4 Program run by the Department of Defense. The goal of Cornell's team for this program is design, build, and launch a two satellite system with a mission goal of in-orbit satellite inspection. My task on the team was to design a memory management system for the Telemetry and Command (T&C) subsystem. This subsystem acts as the gateway between the satellite and the ground station, taking data from the flight computer and storing in memory until the satellite is within transmission range of a ground station.

Although the memory management system design was completed, due to complications throughout the entire CUSat team during the semester the T&C system as a whole is not yet fully operational. However despite not having a completely final version of the microcontroller code finished, it is close. The functions have been written to take in data from the flight computer, store it in designated locations in the flash, and then read this data and send it to the radios when over the ground stations. The groundwork has been established for successful completion of the overall system design goals.

Report Approved by

Project Advisor: _____ Date: _____

Executive Summary

The CUSat team is a multi-year student organized project to compete in the University Nanosat-4 Program. This competition involves universities around the country designing a satellite system in order to encourage the future aerospace workforce in developing new space technologies. The goal of Cornell's team for this program is design, build, and launch a two satellite system with a mission goal of in-orbit satellite inspection. The Telemetry and Command (T&C) subsystem acts as the gateway between the satellite and the ground station. Data such as GPS position, voltage and current levels, and images is collected by the flight computer and transmitted to the T&C board, and must then be stored in memory until the satellite is within range of a ground station. My task with the T&C team was to design a memory management system to handle all of this data. Some of the considerations and limiting factors involved in this process were the amount of data to be stored, bandwidth limitation in transmitting data to the ground, and the size of the memory used to store the data.

Much work was accomplished toward achieving the original goals, but due to complications throughout the entire CUSat team during the semester, the T&C system as a whole is not yet fully operational. However despite not having a completely final version of the microcontroller code finished, it is close. The functions have been written to take in data from the flight computer, store it in designated locations in the flash, and then read this data and send it to the radios when over the ground stations. The groundwork has been established for successful completion of the overall system design goals. The only task remaining is applying the flash interface to the commands from the flight computer, which are still in the process of being finalized.

Table of Contents

<u>1. Project Overview</u>	5
1.1 Background	5
1.2 My Role	5
<u>2. Design Problem</u>	6
2.1 Background	6
2.2 Initial Plan	7
<u>3. Design and Implementation Process</u>	8
3.1 Introduction.....	8
3.2 Overview	8
3.3 Static Memory System	9
3.4 Antenna Communication	11
3.5 Data Packet Size	13
3.6 Flash Lifetime.....	15
<u>4. Design Description</u>	16
4.1 Flash Storage Specifics	16
4.2 Description of Flash code	18
4.3 Testing.....	20
4.4 Future Work.....	21
<u>5. Conclusion</u>	23
<u>Appendix A – flash.h</u>	25
<u>Appendix B – Selection from cucp_ex.c</u>	32

1. Project Overview

1.1 Background

The CUSat team is a student organized project to compete in the University Nanosat-4 Program run by the Department of Defense. The program involves universities around the country in designing a satellite system in order to encourage the future aerospace workforce in developing new space technologies. The goal of Cornell's team for this program is design, build, and launch a two satellite system with a mission goal of in-orbit satellite inspection. The two satellites will separate in orbit and use GPS to maintain a specified distance apart. Cameras on each satellite will record images of the other satellite, and transmit them to the ground so a 3-D model of the target can be reconstructed. Successful demonstration of this technology is a precursor for future satellite missions that can navigate to a target satellite in orbit and provide images to help diagnose any problems. My project focuses on the Telemetry and Command (T&C) subsystem, which handles communication between the satellites and the ground stations. The goal of my project was to develop a method for the T&C subsystem to store data from the satellite until it could be transmitted to the ground.

1.2 My Role

My task with the T&C subsystem was to design a memory management system for all data to be sent from the Space Segment (the two satellite system) to the Ground Segment (consisting of three ground stations in California, Colorado, and New York). Some of the considerations involved were the amount of data to be stored, the amount of data that could actually be transmitted to the ground, and the size of the memory to store the data.

At the beginning of the semester, documentation existed outlining the current estimates and associated design considerations. Development work for the CUSat team has been ongoing for the past four years. To a certain extent, my aim was to work with what was given to me by previous members of the CUSat team. My role was to implement the designs within these constraints. However as I discovered, many changes were needed to the initial design to achieve the desired goals of the system.

2. Design Problem

2.1 Background

The basic design of the T&C hardware board is a microcontroller, flash memory chip, radio, and custom designed antenna. Two identical sets of these components exist on the board (for reasons discussed below). An Atmel Mega 128 microcontroller was chosen for use on the T&C board. The radios that had been selected for the satellites were Kenwood TH-D7E dual band amateur packet-radios. These radios have a rated data transmission of 9600 bps in packet mode. A number of rough initial calculations had been performed in the past to gain a picture of the general function of the T&C subsystem. With an estimated average transmission period of 9.4 minutes per orbit, approximately 700 kB of data could be transmitted. Of this, about 200 kB would be engineering and telemetry data, such as voltage and current values, temperature sensors, and GPS positional data. This would leave 500 kB of image data, or approximately two images at then current estimates of image size after compression. The MCU has 2 kB of built-in SRAM, obviously not enough to store all of the required data until transmission could begin.

Therefore it was decided to use an external flash memory chip. The Atmel AT45DB041B 4Mbit Dataflash® existed on the current revision of the board.

One interesting aspect of the T&C board design was the concept of single-fault tolerance, something that was a consideration through the CUSat design process. Because the T&C system is so critical to the success of the mission, any hardware errors that occur in space would leave the satellites unable to communicate with the ground. Therefore the T&C board has two sets of MCUs, flash chips, radios, and antennas. In the event of an error with the primary system, the identical backup set could be used to allow the mission to continue. This remained an important consideration throughout the semester as the design evolved.

2.2 Initial Plan

After catching up on the current state of the T&C system and discussing matters with team lead Kevin Graf, it was decided to develop a dynamic memory allocation system to store the necessary data in the flash until transmission to occur. The limited size of the flash meant that much more data would be incoming to the T&C subsystem than could possibly be stored or transmitted to the ground. Packets of telemetry data, estimated to be about 2 kB in size, were getting sent to the MCU once every second. Over a 90 minute orbit, this totals $2 \times 60 \times 90 = 10,800$ kB of data, more than 15 times what can actually be transmitted. The memory management system would need to prioritize what data to store and delete the rest. Another consideration was that on some orbits images would be taken and needed to be transmitted to the Ground Segment. For these situations even less

telemetry and engineering data could be stored. Finally, this memory management system needed to be dynamic due to the uncertain order in which data is received from the flight computer, and the lack of storage space in the flash. Some orbits there may be multiple images taken, other orbits there may be none. It was also unknown at what point in the orbit an image may be taken, so there would be no way to be sure to allocate room in the flash. At the beginning of the semester, these were some aspects of the initial plan for the T&C memory design.

3. Design and Implementation Process

3.1 Introduction

My first task after joining the CUSat team and learning about my planned task was to begin analyzing the potential challenges I would face. As stated previously, the main factors affecting my project were the size of data, type of data, and amount of storage I would have available. In order to begin, I needed to know that the documented information on these factors was indeed accurate. Especially since these numbers were based off work from other students, I had no way of knowing the reliability of the information. Any significant changes to these factors would have a significant affect on my project.

3.2 Overview

Throughout the semester, the T&C subsystem significantly evolved in its requirements. These changes also impacted my role on the team, and thus the goals of my project. As changes were made to the memory management aspects of the design, my tasks expanded

to include working on the radio transmission segment of the system. This was a natural extension of my work on the team. My initial goals had me focusing on the interaction between the T&C MCUs and the flight computer that sends data from the other subsystems. Taking this data and storing it in flash is only half the job of the board. The other half was taking the data from flash and actually transmitting it from the radios to the ground.

The largest change to my project came toward the end of October. Based off of recommendations from Professor Mason Peck, it was decided that the T&C subsystem should be relatively “dumb.” In other words, it should not have a complicated system for dealing with all of the data being sent to it from the main flight computer. Instead, the flight computer should only send data to T&C that it definitely wants to have sent to the Ground Segment. It should not be T&C’s job to determine what data to save, what to delete, and what to send. This task should fall on the flight computer. In essence, the T&C MCU only takes action based off commands specifically sent to it. This simplifies T&C’s role, is consistent with the design of the flight computer being the “brain” of the satellite, and eliminates unnecessary data transfer between subsystems.

3.3 Static Memory System

The change to the T&C subsystem resulted in the decision to send a much more limited amount of data to T&C and in a somewhat more expected way. Telemetry packets, estimated at a maximum size of 2 kB were only going to be sent to T&C once every ten minutes to gain snapshots of the satellite system throughout the orbit. Then for a period

of three minutes shortly before the satellite was visible over the ground, telemetry packets would be collected once every second to give an idea of the short term variation. This totals approximately 200 packets, or 400 kB of data. Separate from this is log data from the flight computer, in 1 kB packets. These packets would be received once every ten minutes throughout the orbit, for a total of about 10 kB. Because images were estimated at about a 300 kB maximum size, I decided that there was never a need to have more than two images stored in the T&C flash at any given time. I was also informed that room needed to be allocated in flash for possible uplink data, such as new flight code in the event of an error. This was estimated at 100 kB max. These values are summarized below in Figure 1.

Data Type	Size (kB)
Telemetry	400
Log	10
Images	600
Uplink	100
Total	1110

Figure 1. Data Types and Size Estimates.

The most important conclusion from these numbers is that the 4 mbit flash that was currently on the board is about half the size required to store the estimated amount of data. I checked with the Atmel website and discovered that the DataFlash® product line also came in 8, 16, 32, and 64 mbit versions. We needed at least the 16 mbit chip in order to store the necessary data. Looking over the datasheets, I discovered that the 32 and 64 mbit versions did not come in the same SOIC package type as the 4 mbit version, while the 16 mbit was the exact same package size and type as the 4mbit. After reviewing the potential hardware changes required to the board with subsystem lead Kevin Graf, we decided that switching to the 32 or 64 mbit chips this late in the design process would

require completely redesigning the board and was not feasible. For these reasons, I chose to upgrade to the 16 mbit AT45DB161D DataFlash® for the T&C board. The design calls for a maximum of only 55% of the storage to be used at any given time, allowing for a factor of 2 change in all data quantities without changes needed to the design. One result of using such a large amount of memory was that the memory management system no longer needed to be dynamic. Static locations could be determined for each data type to prevent any complications.

3.4 Antenna Communication

As the semester continued, a fundamental flaw in the T&C design was discovered. The original design called for both radios to be used for crosslink when the satellites are not above a ground station, since you cannot transmit and receive on a single radio. This would allow the most amount of data to be transferred. In the event of a hardware failure only one radio could be used on each satellite. Although this would cut the bandwidth between the two satellites in half the mission could still be function, thus preserving the single fault tolerance concept. Once a ground station came into view, the satellite would switch to have one radio communicating with the ground, and the other communicating with the other satellite. These led to the two MCU/flash/radio sets to be called crosslink and up/downlink. One set always handled the crosslink function, the other set always handled the uplink and downlink functions, and the up/downlink could function for crosslink to allow two way communication when not over the ground.

However, a member of another subsystem realized that with the way the orbits were designed, the satellites switched relative positioning halfway through every orbit. Since the crosslink antenna was on one side of the satellite and the up/downlink one on the other, this meant that it was possible for the antenna for the desired function to be facing in the opposite direction. This could obviously lead to significant transmission problems.

One method discussed to solve this problem was changes to the hardware. If both MCU's could read/write to both flash chips and send this data to both radios, it would solve the problem. Therefore there could still be a dedicated MCU for crosslink and a dedicated MCU for up/downlink, but during the times when the antennas are switched it could still access the correct data and send it to the correct antenna. A system was discussed using 2 to 1 multiplexers on the lines the MCUs use to communicate between the flash chips and radios. When the switch command was received from the flight computer, an output port on the MCU could be toggled and used as a select line to the muxes. However there were two problems with this solution. Hardware changes to the boards take time to perform and it could be many weeks before the new boards could be received and tested, holding up the subsystem. This change would also introduce more complexity into a system we were seeking to make less complex.

It was decided that a software solution would be easier to implement, and be simpler to fix and debug due to the time it takes to issue another hardware revision of the T&C board. The first aspect of this software solution was that CDH would send all downlink data to both MCUs and have it stored in both flash chips. This way, the MCU associated

with whichever antenna was in position for downlink was guaranteed to have access to the data. The flight computer would know the relative positioning of the satellites based off the GPS data, and would issue a command setting a flag in each MCU if it was the current crosslink or up/downlink system. The crosslink system takes the data from the other satellite and sends it right to the flight computer, meaning that it does not need to be stored for very long in the crosslink flash. This solution adds some complexity to the flight computer, as it must manage the two MCUs at once to ensure that each has the required data and is in the correct mode. The one problem with this solution was that it partly affected the single fault tolerance concept that we attempted to adhere to. In the event of a hardware failure, it would be possible that for certain orbits, communication with the ground would not be possible. However this is due to the nature of the orbits, something that it is too late to change. Given the problem we faced, adding some complexity to the flight computer and adding an extra communication protocol command was the most reliable solution.

3.5 Data Packet Size

One important issue for the success of the project was the nature of the interaction between the MCUs and the flight computer. In short, how would the “data” actually be sent to the MCU? This was a detail that had not been discussed too specifically until I attended a meeting with the Command and Data Handling (CDH) team leads. The original plan called for CDH to send 1024 byte data packets to T&C. However I felt that this created a number of problems. First of all, I was worried about the large size of this single packet given that the MCU only had 2 kB of SRAM. Also, the page size of the

flash chip I was planning on using (16 mbit) had a 528 byte page size. This would require a single “packet” of data to be written to two different pages in flash. The radios have a packet size of 256 bytes. These differences in “packet” sizes could create inconsistencies with packet ID numbers of the larger packet coming from CDH and what was being stored in the flash and then transmitted. This could cause problems when trying to track what was actually transmitted, and in the case of data errors, requesting the packet again from CDH. For these reasons, we agreed on a packet size of data coming from CDH of 512 bytes. Packets of 256 bytes were not viable because an entire page of flash must be written to at a single time, so half of the flash memory would be wasted by empty space.

Once the packet size was agreed upon, we discussed methods of storing the data in flash (the heart of my original project focus). One of my concerns was how ID numbers were being assigned to packets of data so that they could be tracked. I mentioned to the CDH leads an important fact about the flash chip that we could use to our advantage. Page sizes in the flash are 528 bytes instead of a “standard” size of 512. This allows us up to 16 bytes of header information to attach to each packet. We decided that two separate ID numbers should be assigned to each packet of data. The first would be a universal ID number unique to every packet on the satellite. We assigned 4 bytes to this, giving 2^{32} (over 4 billion) possible ID numbers before overflow, far more than enough for the satellite in its lifetime. This ID number would have no effect on how the data is stored in the T&C flash, but would be important when T&C reports to CDH what packets were successfully transmitted to the ground. The second ID number would indicate the position of that 512 byte packet within the larger packet of data. For example telemetry

packets of about 2 kB in size would require 4 packets, while pictures that can be up to a few hundred kB could require as many as 800 packets. For this reason, 2 bytes would be required for this ID number. Another 1 byte was assigned to indicating if the packet contains telemetry, log, or image data.

3.6 Flash Lifetime

One concern I had when I began my project was the use of flash memory and the fact that it has a limited number of read/write cycles. The line of DataFlash® products that we use claim to be valid for a minimum of 100,000 program/erase cycles in their lifetime.

Anytime a decision was made regarding how much data would be sent to T&C and how often, I always factored in how it would affect the lifetime of the flash. For example, the original design called for data to be crosslinked between the two satellites at a rate of once per second. If every second this 2 kB packet of data was received and stored in the same pages of flash until the command came to send it to the flight computer, that page of flash would wear out within one day.

$$1 \frac{\text{reprogram}}{\text{second}} * 3600 \frac{\text{seconds}}{\text{hour}} * 24 \frac{\text{hours}}{\text{day}} = 86,400 \frac{\text{reprograms}}{\text{day}}$$

The lifetime of the flash was another one of the reasons why I wanted to use a flash memory large enough to store an entire orbit's worth of data to be transmitted without having to overwrite any pages. For the average of 90 minute orbits, this would be 16 reprograms per day instead of 86,400. As it turns out having a flash large enough to store all this data was beneficial for other reasons, and in the final design flash lifetime should not be an issue.

4. Design Description

4.1 Flash Storage Specifics

The AT45DB161D 16 mbit DataFlash® is a serial interface sequential access flash memory. It contains 4,096 pages of 528 bytes, organized into 512 blocks of 8 pages, and 16 sectors of 32 blocks each (256 pages each). A picture from the datasheet of the organization is shown below.

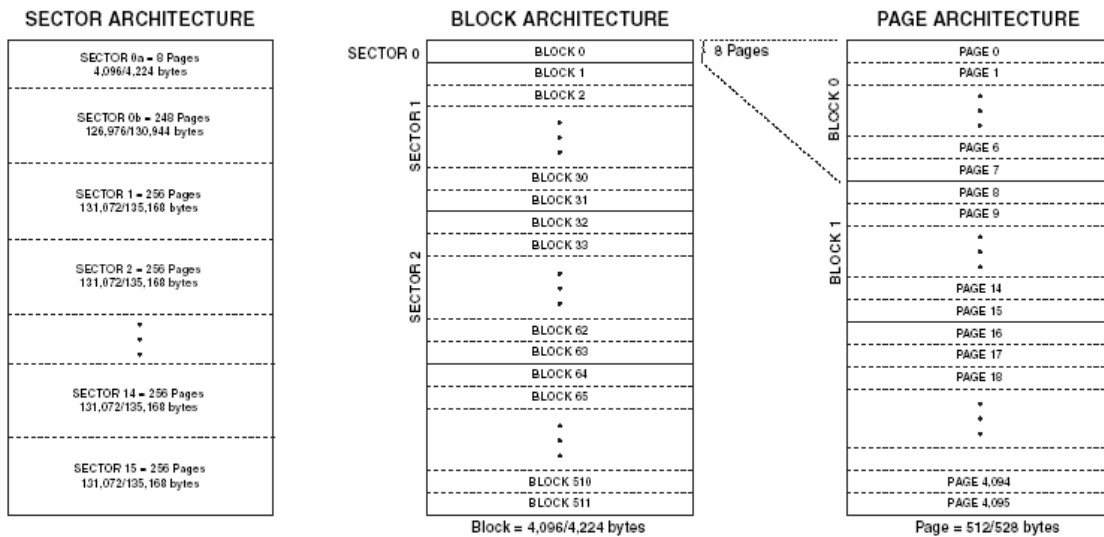


Figure 2. 16 MBit Flash Memory Hierarchy.

The chip also contains two 528 byte SRAM buffers. The buffers allow data to be received while a page in the main memory is being reprogrammed (such as with writing a continuous stream of data). The data types and sizes to be stored in the flash are summarized below in Figure 3. This is an updated version of the table in Figure 1 to represent an increase in the estimate of image size after compression. The values in this table are maximum worst case estimates.

Data Type	Size (kB)
Uplink	100
Telemetry	400
Log	10
2 Images	800
Total	1310

Figure 3. Data Types and Updated Size Estimates.

Based on the above information, the Table in Figure 4 outlines the locations in flash memory where data of each type will be stored. Based on the estimated required storage room, I assigned sectors to each data type. I chose to align the data by sector both in order to simplify the design. While this does lead to some wasted data space in flash, it also allows for some variation in the sizes without any changes being made to the design. For the images, while 400 kB was the estimated worst case size after compression, this value will most likely be on the order of 100-200 kB, well below the designated space of 384 kB. Another benefit of aligning data by sector is to utilize the EraseSector function (discussed later) that erases an entire sector in one command. This will simplify the code and ensure the fastest possible response when an entire sector must be deleted (say at the end of an orbit when the command to clear all telemetry is received).

Data Type	Amount of Storage (kB)	# of Sectors Required	Actual Memory Allocated (kB)	Sector Numbers	Beginning Page Number
Uplink	100	1	128	4	1024
Telemetry	400	4	512	5, 6, 7, 8	1280
Log	10	1	128	9	2304
Image 1	400	3	384	10, 11, 12	2560
Image 2	400	3	384	13, 14, 15	3328

Figure 4. Flash Memory Organization.

As mentioned above data is aligned by sector. However the functions used to write to the flash all take as input a page number, not a sector number. The table below in Figure 5 gives the starting page number of each sector in the flash.

Sector Number	Start Page Number
0	0
1	256
2	512
3	768
4	1024
5	1280
6	1536
7	1792
8	2048
9	2304
10	2560
11	2816
12	3072
13	3328
14	3584
15	3840

Figure 5. Page Numbers by Sector.

4.2 Description of Flash code

When my project began, some of the flash interaction code already existed in rough form. However with the change from the 4 mbit to 16 mbit DataFlash® chip, changes were necessary to the existing function to account for the different organization of pages. In addition, I added new functions that were not implemented in the previous code. The current version of the flash header file flash.h is included in Appendix A. First the starting page location for each data type (based on the table above) is defined, followed by opcodes from the flash datasheet are defined for each of the current functions to keep the code clear. There are a number of read and write functions currently implemented in the code. Below is a brief overview of each one:

BUFRead – This function reads the data from a specified buffer and stores it to a variable in the MCU memory. It takes as input a pointer to a data variable in SRAM, a

byte address to begin reading from in the buffer (0 to 527), the length of the amount of data to be read, and a char indicating which buffer to read from. In the event that the final byte of the buffer is reached, it will wrap around and begin reading at the first byte of the buffer.

BUFWrite - This function reads the data from a variable in the MCU memory and stores it to a specified buffer. It takes as input a pointer to a data variable in SRAM, a byte address to begin writing to in the buffer (0 to 527), the length of the amount of data to be read, and a char indicating which buffer to write to. In the event that the final byte of the buffer is reached, it will wrap around and begin writing to the first byte of the buffer.

MainWrite – This function reads the data from a variable in the MCU memory and stores it to a specified page in flash memory, using a specified buffer. It takes as input a pointer to a data variable in SRAM, a page address to write the data to, a byte address within that page to begin writing the data to, the length of the amount of data to be read, and a char indicating which buffer to pass the data through.

Buf2MainWrite - This function reads the data from a specified buffer and stores it to a specified page in flash memory. It takes as input a page address to write the data to in flash memory and a char indicating which buffer to read from.

PageRead - This function reads the data from a page in flash memory and stores it to a variable in the MCU memory. It takes as input a pointer to a data variable in SRAM, a page address to read the data from, a byte address within that page to begin reading from, and the length of the amount of data to be read. This function bypasses the SRAM buffers.

Main2BufRead - This function reads the data from a specified page in flash memory and stores it to a specified buffer. It takes as input a page address to read the data from in flash memory and a char indicating which buffer to write to.

PageErase - This function erases the data from a specified page in flash memory, setting all the bits to logic 1. It takes as input a page address to erase in flash memory.

BlockErase - This function erases the data from a specified block in flash memory, setting all the bits to logic 1. It takes as input a block number to erase in flash memory.

SectorErase - This function erases the data from a specified sector in flash memory, setting all the bits to logic 1. It takes as input a sector number to erase in flash memory.

4.3 Testing

Testing was performed to ensure basic functionality of the flash read and write functions within the T&C system. Because as of the end of this semester the T&C board has not been revised for the change in flash chip, older code was used to test the flash commands on the 4mbit flash with 264 byte page size. An array of 264 bytes was copied to page 0 of the flash using the MainWrite function. This page was then read by the MCU using the PageRead function and the result output to Hyperterm so it could be verified visually.

While this is a relatively simple test, it does ensure that that the functions work and the flash behaves as anticipated. These functions had to be modified to work with the 16mbit flash and its 528 byte page size, but this should have no effect on the functionality.

4.4 Future Work

Attached in Appendix B is the portion of the MCU code that deals with communication commands from the flight computer. The switch statements include all current commands, however some of these are expected to change in the near future. The first switch statement checks the header of a received command, and performs the associated action for that command. For the commands that involve flash interaction (most of them), this is where the flash read/write commands from flash.h will be called, with the page number location in flash depending on the data type for that specific command. This switch statement is mainly a skeleton at the moment, due to the fact that some of these commands will be changing when CDH updates the command list, so there is no sense writing commands that will change. Due to complications with the Diagnostic Board designed to test these commands, very few of them are even capable of being tested at the moment, another holdup in implementing these commands at this time.

The `get_receive_mem` function also contains a switch statement that checks for the command received. This indicates if it is a valid command, and depending on the nature of the command, allocates memory for an incoming payload, or sets a flag indicating that a response to the command is expected. This function should be mainly operational, aside from a few specific problems listed as comments in the code. Again this was unable to be tested, but based on the current command list should be complete. Any future changes to the commands should not require a significant revision to this function. The list of the current commands, with a brief description is shown below in Figure 6. These commands and the descriptions were designed by members of the CDH subsystem.

Command Description	Value	Network Command [NET_CMD ...]	Payload Type	Expected Response Type
GS commands available?	0x61	TC_CMDS_AVAIL	-	RESP_TC_CMDS_AVAIL
Get one GS command without payload	0x62	TC_GET_CMD	-	RESP_TC_GET_CMD
Get GS command payload	0x63	TC_GET_PAYLOAD		CMD_PAYLOAD
Report command result	0x64	TC_CMD_RESULT	PLD_CMD_RESP	-
Report command result response payload	0x65	TC_CMD_RESULT_PLD	PLD_CMD_RESP_PLD	-
Put telemetry in up/down buffer	0x66	TC_TELEM_BUFFER	PLD_TC_TELEM	-
Put IMI data in up/down buffer	0x67	TC_IMI_BUFFER	IMI_DATA	
Give image metadata to T&C	0x68	TC_IMAGE_METADATA	PLD_TC_METADATA	-
Put image chunk in buffer	0x69	TC_IMAGE_CHUNK	PLD_TC_IMAGE	
Put log file in buffer	0x6A	TC_LOG_BUFFER	LOGBUFFER	
Send data to other satellite	0x6B	TC_CROSS_SEND	-	-
Check for new flight code	0x6C	TC_CHECK_NEW_CODE	-	RESP_TC_FC_SIZE
Get new flight code	0x6D	TC_GIVE_NEW_CODE	-	EXE
Get board telemetry data	0x6E	TC_GET_TELEM	-	RESP_TC_TELEM
Put error report in buffer	0x6F	TC_REPORT_ERROR	PLD_REPORT_ERROR	-
Get GPS data from crosslink	0x71	TC_GET_CROSS_GPS		CROSS_GPS_DATA
Put GPS data into crosslink buffer	0x72	TC_GPS_CROSS_BUFFER	CROSS_GPS_DATA	
Put GPS relative distance into crosslink buffer	0x73	TC_CROSS_GPS_RELD	GPS_RELDIST	-
Stop downlink	0x74	TC_STOP_DOWNLINK		

Figure 6. Communication Protocol Command List.

5. Conclusion

When my project began, the end goal was to present working MCU code that would respond to all commands from the flight computer to store and read data from flash memory. Much was accomplished toward this, but due to complications throughout the entire CUSat team during the semester, this goal was not quite achieved. Despite not having a completely final version of the MCU code finished, it is close. The functions exist to take in data from the flight computer, store it in designated locations in the flash, and then read this data and send it to the radios when over the ground stations. All known difficulties with this system have been addressed.

The only aspect missing from the current T&C functionality is the details of the exact commands from the flight computer to initiate this functionality, and the ability to test them. A diagnostic board was developed to mimic the flight computer and issue commands to the various subsystems on the satellite. Tests were planned during the semester to run a “flatsat”, with each board connected together and to this diagnostic board. However these flatsat tests have been plagued by software and hardware issues of other systems for months, and no successful tests of the communication protocol commands were achieved. In addition, as the semester went on and the details of interaction between the T&C MCUs and the flight computer were finalized, changes were planned to the exact commands that also have not yet been implemented. For these reasons, a version of the code that responds to flight computer commands was not a reasonable goal. However as stated before, once these details are ironed out the existing functions just need to be called with the correct inputs for the given commands and the

system should be functional. This project has helped establish a framework for flash memory management of satellite data on the T&C board that can be used next semester for team members to finalize the system and meet the requirements of the team mission.

Appendix A – flash.h

```
/*
Flash header file
Two functions used to interface with flash:

writeFlash(BYTE * buffer, flashPtr writeLoc);
readFlash(BYTE * buffer, flashPtr readLoc);

        USED FOR WRITING AND READING:

BYTE * buffer is an array of size PAGESIZE (528)
buffer is NOT erased after reads and writes

        USED FOR ADDRESSING:

typedef struct {
    unsigned int page_addr : 12;
    unsigned int byte_addr : 10;
} flashPtr;
*/

/*
#define RXB8 1
#define TXB8 0
#define UPE 2
#define OVR 3
#define FE 4
#define UDRE 5
#define RXC 7
*/

/*
#define FRAMING_ERROR (1<<FE)
#define PARITY_ERROR (1<<UPE)
#define DATA_OVERRUN (1<<OVR)
#define DATA_REGISTER_EMPTY (1<<UDRE)
#define RX_COMPLETE (1<<RXC)
*/

// Page size
#define PAGESIZE 528

//BEGINNING PAGE NUMBERS OF EACH DATA TYPE
#define UPLINK_START 1024
#define TELEM_START 1280
#define LOG_START 2304
#define IMAGE1_START 2560
#define IMAGE2_START 3328

// Internal Functions
char FLASHStatus(void);
void BUFRead(unsigned char *data, unsigned int addr, unsigned char
length, unsigned char buf);
```

```

void BUFWrite(unsigned char *data, unsigned int addr, unsigned char
length, unsigned char buf);
void MainWrite(unsigned char *data, unsigned int page_addr, unsigned
int byte_addr, unsigned char length, unsigned char buf);
void Buf2MainWrite (unsigned int addr, unsigned char buf);
void PageRead (unsigned char* data, unsigned int page_addr, unsigned
int byte_addr, unsigned char length);
void Main2BufRead(unsigned int page_addr, unsigned char buf);

// External Functions and Data structures

typedef struct {
    unsigned int pageAddr : 12;
    unsigned int byteAddr : 10;
} flashPtr;

//BYTE buffer[PAGESIZE] = {0};

void writeFlash(BYTE * buffer, flashPtr writeLoc);
void readFlash(BYTE * buffer, flashPtr readLoc);

//Page is 528 bytes, buffer is 528 bytes
#define BUFFER1 1
#define BUFFER2 2

//Status Read
#define STATUS_READ 0xD7 //0x57
#define SPI_BUSY ((SPSR & 0x80)==0)

// DEFINE OPCODES FOR FLASH FUNCTIONS

//Buffer Read
//followed by 14 don't care bits, 10 byte address, 8 don't care - wraps
around
#define BUF1_READ 0xD4 //0x54
#define BUF2_READ 0xD6 //0x56

//Buffer writes
//followed by 14 don't care bits, 10 byte address - write continues
with wrap-around until CS HIGH
#define BUF1_WRITE 0x84
#define BUF2_WRITE 0x87

//Main Memory Page through Buffer with erase
//followed by 2 don't care bits, 12 page address, 10 byte address
#define MAIN_WRITE1 0x82
#define MAIN_WRITE2 0x85

//Buffer to Main mem
//followed by 2 don't care bits, 12 page address, 10 don't care
//on CS High - erases page then writes
#define BUF1_TO_MAIN_WRITE 0x83

```

```

#define BUF2_TO_MAIN_WRITE 0x86
//if desired, same function but it assumes page has already been erased
//currently, these function are not nedded
#define BUF1_TO_MAIN_NO_ERASE 0x88
#define BUF2_TO_MAIN_NO_ERASE 0x89

//Main Mem Read
//followed by 2 don't care, 12 page address, 10 byte address, 32 don't
care
#define PAGE_READ 0xD2 //0x52

// Main read to Buffers
//followed by 2 don't care bits, 12 page address, 10 don't care
#define MAIN_TO_BUF1_READ 0x53
#define MAIN_TO_BUF2_READ 0x55

//Page Erase
//followed by 2 don't care bits, 12 page address, 10 don't care
//erases to 1s on CS HIGH
#define PAGE_ERASE 0x81

//Block Erase
//followed by 2 don't care bits, 9 block address, 13 don't care
//erases block of 8 pages to 1 on CS HIGH
#define BLOCK_ERASE 0x50

//Sector Erase
//followed by 2 don't care bits, 4 sector address, 18 don't care
//erases sector of 256 pages to 1 on CS HIGH
#define SECTOR_ERASE 0x7C

//Currently unused potential function
//Continuous Array Read
//followed by 2 don't care, 12 page address, 10 byte address, 32 don't
care
//additional clock pulses read data while CS low
#define CONT_READ 0xE8 //0x68

//FLASH INTERACTION FUNCTIONS

char FLASHStatus(void) {
    PORTB &= ~0x01; //CS LOW
    SPDR = 0xD7; //opcode for read status
    while (SPI_BUSY); //wait for transmission complete
    SPDR = 0x00; //read SPDR
    while(SPI_BUSY); //wait
    PORTB |= 0x01; //end
    return SPDR;
}

```

```

void BUFRead(unsigned char *data, unsigned int addr, unsigned char
length, unsigned char buf)
{
    unsigned char i;
    PORTB &= ~0x01; //CS LOW
    if (buf==BUFFER1)
    {
        SPDR = BUF1_READ; //opcode for read status
    }
    if (buf==BUFFER2)
    {
        SPDR = BUF2_READ;
    }
    while (SPI_BUSY); //wait for transmission complete
    SPDR = 0x00; //don't care
    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)(addr>>8); //don't care and 1st 2 addr bits
    while (SPI_BUSY);
    SPDR = (char)(addr); //address
    while (SPI_BUSY);
    SPDR = 0x00; //don't care
    while (SPI_BUSY);

    for(i=0;i<length;i++)
    {
        SPDR = 0x00; //read
        while (SPI_BUSY);
        *data = SPDR;
        data++;
    }
    PORTB |= 0x01; //end
}

```

```

void BUFWrite(unsigned char *data, unsigned int addr, unsigned char
length, unsigned char buf)
{
    unsigned char i;
    PORTB &= ~0x01; //CS LOW
    if (buf==BUFFER1)
    {
        SPDR = BUF1_WRITE; //opcode for read status
    }
    if (buf==BUFFER2)
    {
        SPDR = BUF2_WRITE;
    }
    while (SPI_BUSY); //wait for transmission complete
    SPDR = 0x00; //don't care
    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)(addr>>8); //don't care and 1st 2 addr bits
    while (SPI_BUSY);
    SPDR = (char)(addr); //address 0
    while (SPI_BUSY);
    for (i=0;i<length;i++)

```

```

        {
            SPDR = *data;        //data
            while (SPI_BUSY);
            data++;
        }

    PORTB |= 0x01;            //end
}

```

```

void MainWrite(unsigned char *data, unsigned int page_addr, unsigned
int byte_addr, unsigned char length, unsigned char buf)

```

```

{
    unsigned char i;
    PORTB &= ~0x01; //CS LOW
    if (buf==BUFFER1)
    {
        SPDR = MAIN_WRITE1;    //opcode for read status
    }
    if (buf==BUFFER2)
    {
        SPDR = MAIN_WRITE2;
    }
    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)(page_addr>>6);
    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)((page_addr<<2) | ((0x0300 & byte_addr)>>8));
    while (SPI_BUSY);
    SPDR = (char)(byte_addr);
    while (SPI_BUSY);
    for (i=0;i<length;i++)
    {
        SPDR = *data;        //data
        while (SPI_BUSY);
        data++;
    }
    PORTB |= 0x01;            //end
}

```

```

void Buf2MainWrite (unsigned int addr, unsigned char buf)

```

```

{
    PORTB &= ~0x01; //CS LOW
    if (buf==BUFFER1)
    {
        SPDR = BUF1_TO_MAIN_WRITE;
    }
    if (buf==BUFFER2)
    {
        SPDR = BUF2_TO_MAIN_WRITE;
    }
    while (SPI_BUSY);
    SPDR = (char)(addr>>6);
    while (SPI_BUSY);
    SPDR = (char)(addr<<2);
    while (SPI_BUSY);
    SPDR = 0x00; //don't care;
}

```

```

    while (SPI_BUSY);
    PORTB |=0x01;
}

void PageRead (unsigned char* data, unsigned int page_addr, unsigned
int byte_addr, unsigned char length)
{
    unsigned char i;
    PORTB &= ~0x01;

    SPDR = PAGE_READ;
    while (SPI_BUSY);
    SPDR = (char)(page_addr>>6);
    while (SPI_BUSY);
    SPDR = (char)((page_addr<<2) | ((0x0300 & byte_addr)>>8));
    while (SPI_BUSY);
    SPDR = (char)(byte_addr);
    while (SPI_BUSY);

    //32 don't care bits
    for (i=0;i<4;i++)
    {
        SPDR = 0x00;
        while (SPI_BUSY);
    }

    for (i=0;i<length;i++)
    {
        SPDR = 0x00;        //read
        while (SPI_BUSY);
        *data = SPDR;
        data++;
    }
    PORTB |=0x01;
}

void Main2BufRead(unsigned int page_addr, unsigned char buf) {
    PORTB &= ~0x01; //CS LOW
    if (buf==BUFFER1)
    {
        SPDR = MAIN_TO_BUF1_READ; //opcode for read status
    }
    if (buf==BUFFER2)
    {
        SPDR = MAIN_TO_BUF2_READ;
    }

    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)(page_addr>>6); //2 reserve, then 6 page_addr bits,
PA11-PA6
    while (SPI_BUSY); //wait for transmission complete
    SPDR = (char)(page_addr<<2); //6 addr bits and 2 don't care
    while (SPI_BUSY);
    SPDR = 0x00; // 8 don't cares
    while (SPI_BUSY);
}

```

```

        PORTB |= 0x01; //end
    }

void PageErase(unsigned int page_addr) {
    PORTB &= ~0x01; //CS LOW

    while (SPI_BUSY);
    SPDR = (char)(page_addr>>6);
    while (SPI_BUSY);
    SPDR = (char)(page_addr<<2);
    while (SPI_BUSY);
    SPDR = 0x00; //don't care;
    while (SPI_BUSY);
    PORTB |=0x01;
}

void BlockErase(unsigned int block_addr) {
    PORTB &= ~0x01; //CS LOW

    while (SPI_BUSY);
    SPDR = (char)(block_addr>>3);
    while (SPI_BUSY);
    SPDR = (char)(block_addr<<5);
    while (SPI_BUSY);
    SPDR = 0x00; //don't care;
    while (SPI_BUSY);
    PORTB |=0x01;
}

void SectorErase(unsigned int sector_addr) {
    PORTB &= ~0x01; //CS LOW

    while (SPI_BUSY);
    SPDR = (char)(block_addr<<2);
    while (SPI_BUSY);
    SPDR = 0x00; //don't care;
    while (SPI_BUSY);
    SPDR = 0x00; //don't care;
    while (SPI_BUSY);
    PORTB |=0x01;
}

void writeFlash(BYTE * buffer, flashPtr * writeLoc) {
    MainWrite(buffer, writeLoc.page_addr,
writeLoc.byte_addr,PAGESIZE, BUFFER1);
}
void readFlash(BYTE * buffer, flashPtr * readLoc) {
    PageRead(buffer, readLoc.page_addr, readLoc.byte_addr, PAGESIZE);
}

```

Appendix B – Selection from cucp_ex.c

```
if (Net_Msg_Status == MSG_EBSRIE) { // Command and payload have been
received
    // continue CUCP data transfer
    Net_Msg_Status = MSG_READ;
    // 11/27/06 all commands included, but need to actually fill in
actions for each case
switch (Net_Cmd) {
    case NET_CMD_PING:
        break;
    case NET_CMD_GET_STATUS:
        send((BYTE32 *) &myStatus, RESP_STATUS_SIZE);
        break;

    case NET_CMD_SILENCE:
        break;

    case NET_CMD_REPROGRAM:
        break;

    case NET_CMD_SET_TIME:
        break;

    case NET_CMD_TC_CMDS_AVAIL:
        break;

    case NET_CMD_TC_GET_CMD:
        break;

    case NET_CMD_TC_GET_PAYLOAD:
        break;

    case NET_CMD_TC_CMD_RESULT:
        // Temporary TCAP testing cmd.
        tcapActive = 1;
        break;

    case NET_CMD_TC_CMD_RESULT_PLD:
        break;

    case NET_CMD_TC_TELEM_BUFFER:
        break;

    case NET_CMD_TC_IMI_BUFFER:
        break;

    case NET_CMD_TC_IMAGE_METADATA:
        break;

    case NET_CMD_TC_IMAGE_CHUNK:
```



```

        break;

case NET_CMD_TC_LOG_BUFFER:
    break;

case NET_CMD_TC_CROSS_SEND:
    //xmitData(&myCrossData,CROSS_GPS_DATA_SIZE);
    xmitData(&myCrossGPS,128); //hack
    break;

case NET_CMD_TC_CHECK_NEW_CODE:
    break;

case NET_CMD_TC_GIVE_NEW_CODE:
    break;

case NET_CMD_TC_GET_TELEM:
    send((BYTE32 *) &myTelem, RESP_TC_TELEM_SIZE);
    break;

case NET_CMD_TC_REPORT_ERROR:
    break;

case NET_CMD_TC_GET_CROSS_GPS:
    PORTC ^= 0x1;
    /* Send received GPS from Cross to hypertrm */
    xmitData(buffer,10);
    if(GPStoTransfer == 1)
    {
        xmitData(buffer,10);
    }
    break;

case NET_CMD_TC_GPS_CROSS_BUFFER:
    {
        int i = 0;
        for( ; i<10; i++)
        {
            buffer[i] = i+0x30;    // this way it looks like ascii
numbers
        }
    }
    GPStoTransfer = 1;
    /* Receive data from CDH and signal that you have GPS data
       to send to other satelltie */
    break;

case NET_CMD_TC_CROSS_GPS_RELD:
    break;

case NET_CMD_TC_STOP_DOWNLINK:
    break;

default:
    break;
}

```

```

    } //if (Net_Msg_Status == MSG_EBSRIE)

}

}

BYTE* get_receive_mem(BYTE Net_Cmd, UINT* memBufferSize, BYTE*
hasResponse, BYTE* isValidCmd) {
*isValidCmd = 1;

switch (Net_Cmd) {
    case NET_CMD_PING:
        *memBufferSize = 0;
        *hasResponse = 0;
        return NULL;

    case NET_CMD_GET_STATUS:
        *memBufferSize = 0;
        *hasResponse = 1;
        return NULL;

    case NET_CMD_SILENCE:
        *memBufferSize = 0;
        *hasResponse = 0;
        return NULL;

    case NET_CMD_REPROGRAM:
        *memBufferSize = REPROGRAM_PAGE_SIZE;
        *hasResponse = 0;
        return (BYTE32*) &myReprogram;

    case NET_CMD_SET_TIME:
        *memBufferSize = TIMESTAMP_SIZE; //sizeof(MyTime);
        *hasResponse = 0;
        return (BYTE32*) &myTime;

    // edited 11/28/06 to include all current CUCP commands
    case NET_CMD_TC_CMDS_AVAIL:
        /*memBufferSize = RESP_TC_CMDS_AVAIL_SIZE;
//sizeof(RESP_TC_CMDS_AVAIL);
        *memBufferSize = 0;
        *hasResponse = 1;
        return NULL;

    case NET_CMD_TC_GET_CMD:
        /*memBufferSize = RESP_TC_GET_CMD_SIZE;
        *memBufferSize = 0;
        *hasResponse = 1;
        return NULL;

    case NET_CMD_TC_GET_PAYLOAD:
        *memBufferSize = 0;
        *hasResponse = 1;

```

```

        return NULL;

case NET_CMD_TC_CMD_RESULT:
    *memBufferSize = PLD_CMD_RESP_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myCommandResponse;

//not sure how to isolate payload from PLD_CMD_RESP
case NET_CMD_TC_CMD_RESULT_PLD:
    /*memBufferSize = ;
    *hasResponse = 0;
    return;

case NET_CMD_TC_TELEM_BUFFER:
    *memBufferSize = PLD_TC_TELEM_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myTelemBuffer;

case NET_CMD_TC_IMI_BUFFER:
    *memBufferSize = IMI_DATA_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myIMI;

case NET_CMD_TC_IMAGE_METADATA:
    *memBufferSize = PLD_TC_METADATA_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myMetadata;

case NET_CMD_TC_IMAGE_CHUNK:
    *memBufferSize = PLD_TC_IMAGE_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myImage;

//Don't see LOG_BUFFER in cucp_def_data.h
case NET_CMD_TC_LOG_BUFFER:
    /*memBufferSize = ;
    *hasResponse = 0;
    //return (BYTE32*) &;

case NET_CMD_TC_CROSS_SEND:
    *memBufferSize = 0;
    *hasResponse = 0;
    return NULL;

case NET_CMD_TC_CHECK_NEW_CODE:
    *memBufferSize = 0;
    *hasResponse = 1;
    return NULL;

case NET_CMD_TC_GIVE_NEW_CODE:
    *memBufferSize = 0;
    *hasResponse = 1;
    return NULL;

case NET_CMD_TC_GET_TELEM:
    *memBufferSize = 0;

```

```

        *hasResponse = 1;
        return NULL;

case NET_CMD_TC_REPORT_ERROR:
    *memBufferSize = PLD_REPORT_ERROR_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myError;

case NET_CMD_TC_GET_CROSS_GPS:
    *memBufferSize = 0;
    *hasResponse = 1;
    return NULL;

case NET_CMD_TC_GPS_CROSS_BUFFER:
    *memBufferSize = CROSS_GPS_DATA_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myCrossGPS;

case NET_CMD_TC_CROSS_GPS_RELD:
    *memBufferSize = GPS_RELDIST_SIZE;
    *hasResponse = 0;
    return (BYTE32*) &myGPSDist;

case NET_CMD_TC_STOP_DOWNLINK:
    *memBufferSize = 0;
    *hasResponse = 0;
    return NULL;

default:
    *isValidCmd = 0;
    return NULL;

```