CIRCUIT BOARD AND MICROCONTROLLER SOFTWARE DESIGN FOR A SATELLITE POWER SYSTEM

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

Rajesh Atluri

Project Advisor: Bruce Land

Degree Date: August 2011

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Circuit Board and Microcontroller Software Design for a Satellite Power System

Author: Rajesh Atluri

Abstract:

Out of Violet's electrical subsystems, the Power subsystem is the most critical because it is housed on a single board that supplies power to all other subsystems and is designed in-house. The Power System functions include using solar panels to collect energy, storing energy in batteries with appropriate protection systems, sampling components' and subsystems' sensors to monitor their power consumption, distributing power from solar cells and batteries, acting as a controller for the Flight Computer, and communicating with the Flight Computer through data packets. Hence, the Power microcontroller (MCU) serves as power controller and power monitor. Revision 1 of the Power Board layout was completed, and the populated board was tested in a Flat-sat setup. MCU code was written to execute the major control, monitoring, and communication functions. Correct functionality of the software functions was verified on a STK500-based testbench, which closely resembles the interfaces of the power system in the satellite. In the near future, further testing of the software will be done on a Revision 2 Power Board with a programmed MCU in order to model flight-like conditions.

EXECUTIVE SUMMARY

The goal of this project was to design key components of the Power subsystem of the Violet Satellite Project, specifically the layout of the Power Board and the software for the Power Microcontroller (MCU). Violet is an interdisciplinary team (see violet.cusat.cornell.edu) of Cornell engineering students who are building a nano-satellite for launch in 2012 after ranking second in the Air Force Research Laboratory's University Nanosat-6 Flight Competition.

Most components of Violet's Power subsystem, including the switches that connect other subsystems to power sources, DC-DC converters, and current and voltage sensor circuits, are housed on a single Power Board that supplies power from solar panels and NiCad batteries to all the other subsystems of the satellite. Although many other subsystems had reached mature stages of design when this project was started, the Power Board part of the Power subsystem had only reached the schematic stage. Hence, the design of the Power Board was greatly constrained by the existing designs of other subsystems and the interfaces of the Board. The major constraint on the Power Board layout design was this rectangular metal box that the Board and all its components had to fit inside. The metal box was already part of the structural model of Violet, so it was a rigid constraint. The first iteration, i.e. Revision 1, of the layout of the dense, 10-layer Power Board was completed. A Revision 1 Board was fabricated and populated with most of the components, and it was tested to verify that the connections agreed with the schematic, and that there were no unintended shorts. A few mistakes were found with the Revision 1 Board due to mistakes in the schematic, which the layout was derived from, and so, a Revision 2 layout was completed by other members of the Violet team, which is outside the scope of this project.

The Power Board has a microcontroller because several functions of the Power subsystem need to be carried out by a local, dedicated computer instead of using the limited resource of Violet's Flight Computer. The Power MCU functions include distributing and monitoring power from solar cells and batteries, acting as a controller of power switches and the magnetic torquer for the Flight Computer, and communicating with the Flight Computer through data packets. Hence, the Power MCU has three major functional roles: sampling the components' and subsystems' sensor values, controlling the switches and the power drawn from the power sources, and communicating with the Flight Computer via the Command and Data Handling Board using a reliable, packet-based protocol called the Violet Communication Protocol (VCP). These major functions of the Power MCU were implemented in the current version of the MCU code; however, the battery state-of-charge monitoring function and associated functions have not been implemented yet since Violet's Power sub-team is still determining the details of the state-of-charge computing algorithm. Unfortunately the testing for this project was limited by the availability of flight-grade hardware. The functions in the current code have been verified by preliminary tests conducted on the STK500, but further testing under more flight-like conditions should be done in the future.

Table of Contents

INTRODUCTION

Since it is one of the more challenging aspects of Violet's electrical design, a functional Power Board is critically important to Violet's mission success as the nano-satellite is expected to launch in late 2012. My original goal was to have the MCU code done and working on a Revision 1 Power Board by early January for the UNP-6 Final Competition Review, which is where the Violet team leads will present, and the Violet satellite design will be judged. Due to limited availability of hardware, I actually ended up completing most of my code testing on a STK500 system as a prototype for the actual Power subsystem.

Out of Violet's electrical subsystems, the Power subsystem is the most critical because it is housed on a single, central board, which supplies power to all other subsystems and will be designed entirely by Violet team members as opposed to acquiring a commercial power system. The Power subsystem must be robust and very reliable as it is one of the major single points of failure that can disrupt the successful operation of the satellite in space. Hence, it is an excellent project to apply the best practices of board-level hardware and MCU software design.

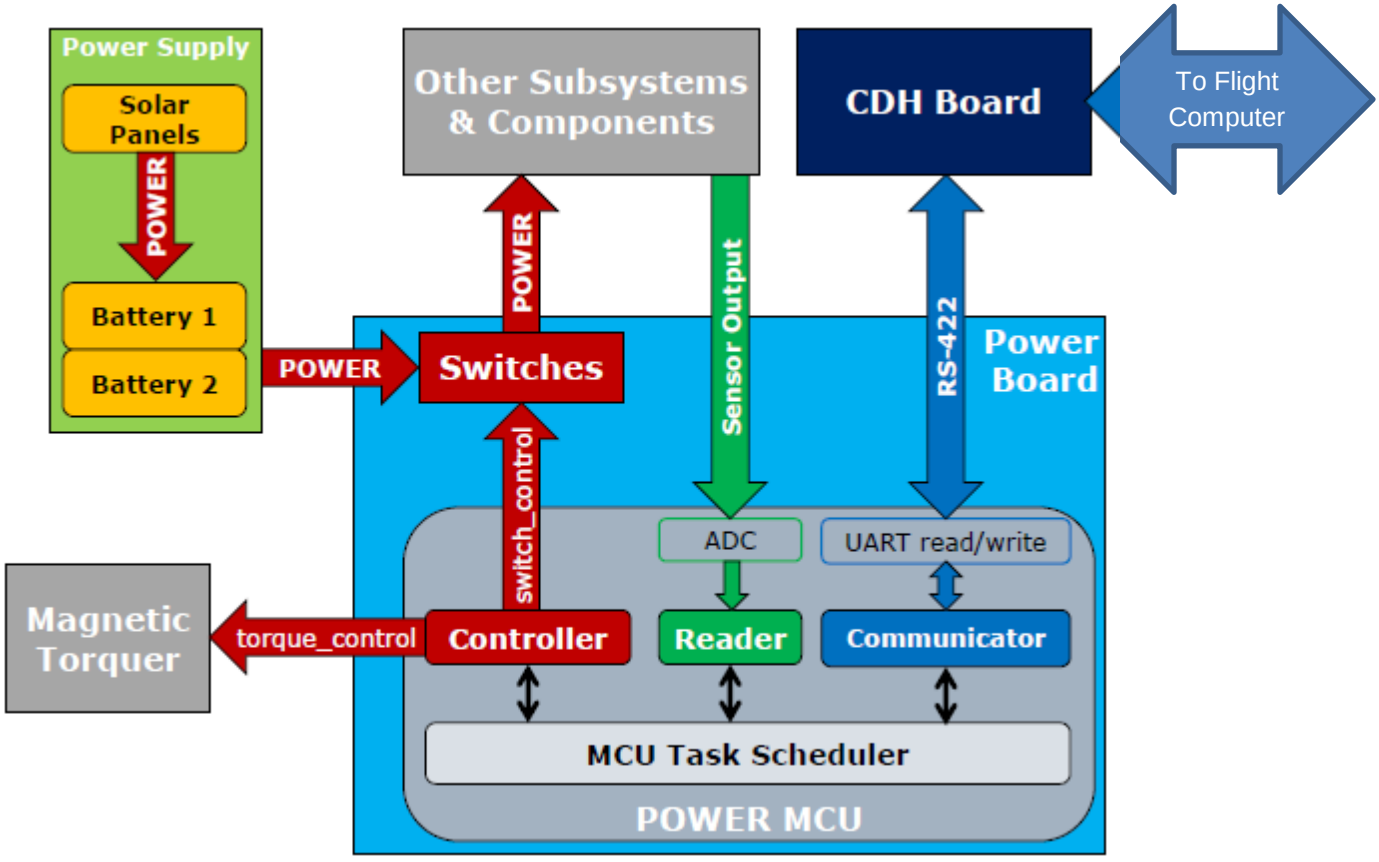


Figure 1: System Block Diagram of the Power Board and its interfaces

As shown in Figure 1, the Power Board will have to interface with the other components of the satellite and will have the following functions:

- Using solar panels to collect energy
- Storing energy in batteries with appropriate protection systems
- Distributing and monitoring power from the solar cells and batteries
- Employing inhibits, or launch lockout devices, that isolate the satellite from any power source prior to and during launch
- Acting as a controller for the Flight Computer and the ground station
- Controlling the magnetic torque

All of the control aspects of these functions will be carried out by the Atmel ATmega128 microcontroller (MCU) on the Power Board. Thus, the MCU will serve as both power regulator and power controller. It will maintain safe battery charge level at all times and supply power to various subsystems and components. Safe battery charging will also require a charging algorithm to be implemented on the MCU.

To make monitoring easier to implement, the Power Board MCU will handle the processing of power telemetry signals. The MCU will sample current, voltage, and temperature to detect any faulty behavior of satellite components, which is indicated by large power consumption and could endanger the satellite as whole.

The MCU will also allow the Flight Computer and the ground station to control the power switches that connect the power sources, batteries and solar cells, to components. The Flight Computer will have greater processing power than the Power MCU and will be able to determine when to power on/off a component. When controlling a power switch is desired, the Flight Computer will send the appropriate command to the Command & Data Handling (CDH) Board and then to the Power MCU. The MCU will use switch circuits to control all the individual power lines to each component. Additionally, the Power MCU will control the precise motion of the magnetic torquers. A magnetic torquer is a device that uses electromagnetic coils to control the attitude of the satellite by coupling the magnetic field produced by the coils to the Earth's ambient magnetic field.

PCB LAYOUT OF POWER BOARD

As described in the previous section, the Power Board interfaces with the solar cells, the batteries, the inhibits, the CDH Board, the magnetic torquer, and other major components of the satellite. Because the power system requirements (see VIOLET-PWR-001) involve measurements and control, the power board contains a MCU which controls the operation of the

entire system. The main voltage bus lines deliver either regulated or unregulated power to each subsystem. The MCU controls switches that determine whether the subsystem receives power through the voltage bus lines. Sensor circuits provide the MCU with real time data about subsystems' power consumption. Analog multiplexors (MUXs) allow the MCU to monitor and process this large amount of data. These are some of the components that are on the Power Board. For a more detailed description of the circuits on the Power Board, refer to Violet document VIOLET-PWR-003, "Power Board Design." Also, VIOLET-PWR-001 gives the system requirements of the power system.

Design Requirements

More than 400 chips and discrete parts were placed and routed within a 98.00 mm by 187.91 mm area as part of the PCB layout. The Board schematic appears in Appendix I. The board dimensions were a "hard" design constraint determined by the Structures sub-team of Violet for the shielded metal box that houses the power board. Another "hard" design constraint was the location of the 5 off-board connectors, which could not be changed since Structures had made the metal enclosures, and the Harness sub-team had determined the final arrangement of all the connections/wires throughout the satellite. The Violet document VIOLET-PD-PWR-200, which I got from the Structures sub-team, shows these constraints and appears in Appendix I.

The layout EDA tool used was Orcad Layout, which the Violet team owns a license for. The layout process considered the following design specifications:

- The mechanical/physical specifications of VIOLET-PD-PWR-200, including the board dimensions and the location and size of off-board connectors, must be met.
- The number of layers used must be minimized to reduce board fabrication costs.
- The number of vias and the length of traces must be minimal to reduce fabrication costs.
- The trace widths must account for the high current that some of the devices will carry.
- The layout must include features that consider electromagnetic effects and reduce noise on signal lines, "cross-talk" between analog and digital lines, and interaction between high voltage and low voltage signals.

An earlier version of the power board layout had been fully routed with all of the components placed, but none of the trace widths were adjusted for high currents. Minimal trace width of 0.005 inches, or 5 mils, can carry ~0.5 Amps of current and was used for all of the traces of this board. That board design used 10 layers and was never fabricated because of doubts of its functionality, and because there was no immediate need for a Power Board. The fact that the first revision of the layout that I completed occupies 10 layers is a significant accomplishment since my layout actually accounts for high currents through certain traces.

Layout Design and Implementation

This section gives a ten step process that was followed to complete the Revision 1 layout. Since the Power Board is densely populated with components and occupied 10 layers—why so many layers were needed will be explained in this section—it was important to follow such steps to successfully complete the time-intensive layout design.

Step 1: To start the layout, I needed to know the maximum amount of current that each trace on the power board could be carrying. Luke Ackerman and Evan Respaut calculated the expected maximum current through all the traces of the Power Board and provided me with the Excel file. Since the minimum trace width of 4pcb.com's fabrication process was 0.005 inches, or 5 mils, which could carry ~0.5 Amps current, I only had to calculate the trace widths for lines carrying 0.5 Amps or more current. 4pcb.com is the PCB manufacturer that the Violet team uses.
The widths of all high-current traces on the power board were calculated using 4pcb.com's trace width calculator.

Step 2: Routing of the high power switching blocks was done first because the large width traces limited the minimum area that these switching blocks could occupy. The high power switches include CMG_PWR, FC_PWR_5, MAE_PWR, RF1_PWR, RF2_PWR, FC_PWR_3.3, and VCC, supplying power to the most power hungry components on the satellite. These switching blocks were laid out on the top and bottom sides of the PCB, and the blocks that had similar trace width sizing, e.g. MAE_PWR and FOG_PWR_5V, were laid out on opposite sides of the board. I thought that this would be a good strategy for reducing the board area taken up by the switches. Since each block has almost the same kinds of components, the surface mount components could actually fit on top of each other perfectly. Then the other power switching blocks were routed, again having switching blocks on both the top and bottom sides of the board. Table 1 shows the voltage used by each subsystem of the satellite and connected across the power switches.

Table 1: Power Line Requirements

Subsystem	Voltage(V)
Interface MCU	3.3
GPS1	5
GPS2	5
Flight Computer	5
Spectrometer	5
Sun Sensor	12 (unregulated)
Magnetic Torquer	12 (unregulated)
Maestro	12 (unregulated)
Radio1	12 (unregulated)
Radio2	12 (unregulated)
CMG	12 (unregulated)

Spectrometer	12
Camera	12
Magnetometer	15
LN-200	+/- 5, +/- 15

Step 3: All of the other circuits on the board that had to be placed close together and could form blocks were placed in a way that minimized area consumed and still made routing possible. The solar cell current sense blocks fall under this category.

Step 4: Now the placement of components on the board could be finalized. Before blocks were moved around, they were first assigned to "groups" using a feature of OrCad Layout that allows you to group components that you would like to move/place together. This made it possible to easily move a routed block to other parts of the board without messing up the traces and vias that had been made inside the block.

As I said earlier, the off-board connectors were a hard design constraint that could not be moved, so their location strongly influenced the placement of components. A switching block that made connections to one of the off-board connectors was placed near that off-board connector. Thus, the left side of the board was almost entirely occupied by switching blocks, which were connecting to the micro-D connectors adjacent to the left end of the board.

The analog MUXs were spaced apart from each other and placed to the right of the switching blocks, which send voltage and current telemetry signals to the MUXs' inputs. The MUXs were spaced apart to allow for routes that had to go across them. I expected the layers around the MUXs to be occupied, and the layout to become very dense there since many signals had to be routed to the MUXs' pins.

The MCU was placed to the right of the MUXs, leaving enough space between the MUXs and MCU for routing that would pass through this area. Again, I expected this to be a high density part of the board because these surface mount chips with many pins were all connected together. Then the analog MUXs and MCU were rotated in such a way that the output lines from the MUXs could be routed in parallel (like a bus) to their corresponding MCU pins. The MCU ended up occupying the center of the board, which was ideal since it connected to many different components/points on the board.

The DC-DC converters all had to be placed on the top side of the board because there was ~0.2 inches of clearance for bottom-side components in the metal enclosure housing the board. I placed them along the bottom and right parts of the board because they had few connections, and some of their high-current connections had to go to the connectors. I expected to route these high-current traces along the perimeter of the board. Also, these locations kept the high power lines of the DC-DC converters away from the MCU and low power, digital circuitry.

The solar current sensors did not occupy a lot of board area, but they all had to connect to an analog MUX that interfaced with the MCU, so they were placed on the bottom side of the board

opposite from the DC-DC converter on the top side. This allowed for a way to route the current sense lines to the MUX that was closest to the bottom edge of the board.
The relays were placed on the right side of the board in order to keep them separate from the low voltage and digital circuits on the other parts of the board. The major traces through the relays were very high current – in fact they had the highest trace width lines on the board – so they would be impossible to route by traces. Since there trace widths were huge, I expected to use copper pours to serve as routes for these connections. Keeping them close together and away from everything else reduced the amount of area that they occupied and made it possible to route them.

Step 5: All of the blocks that had not been internally (to the block) routed were routed at this step. The power switching blocks were already internally routed.

Step 6: Luke, Evan, and I discussed what to do with the relays at this point. After looking at the placement of the relays, we realized that there would be several overlapping copper pours, which would determine the minimum number of internal layers for routing this board. The overlapping copper pours that made the high-current relay connections occupied 5 different inner layers, so I used 5 routing layers along with the top and bottom layer for routing this board. I did not know how to make a copper pour connect two pins on the same net, so Evan made the copper pours that connected the un-routable-with-traces nets of the relays.

Step 7: Any high current traces on the board were then routed. I tried to reduce the number of vias and transitions to other layers, but in certain parts of the board, especially near the off-board connectors where the layout became very dense, the high current traces had to traverse into another inner layer sometimes.

Step 8: The routing between the MUXs' output and the MCU was done manually in order to ensure the routing was done in the simplest way possible.

Step 9: At this point, approximately 65% of the routes on the entire board were complete according to the "Statistics" tab of OrCad Layout. Layout's auto-router was used to route the remainder of the board. By adjusting the settings, I made the auto-router limit the amount of vias it used. When the auto-router was complete only ~15 routes were still missing. These routes were completed manually. The board was completely routed at this point, but the layout had to go through 4pcb.com's DFM (Design for Manufacture) checker. Table 2 shows each layer of the board and briefly describes its purpose.

Table 2: Layers of the Power Board

Layer	Purpose
TOP	General routing; Has all the DC-DC converters, through-hole components, and off-board

	connectors
BOT	General routing; Has many surface mount chips, e.g. the MUXs, MCU, and H-bridges
GND	Plane layer for Ground
PWR	Plane layer for Vcc = 5 V
VUR	Plane layer for unregulated voltage, which powers high power components and is used by DC-DC converters
IN1	Routing layer used mostly for horizontal traces
IN2	Routing layer used mostly for vertical traces
IN3	Routing layer used mostly for horizontal traces
IN4	Routing layer used mostly for vertical traces
IN5	Routing layer used mostly for horizontal traces

Step 10: 4pcb.com has a useful tool that checks the layout Gerber files for DFM errors. I uploaded the Gerber files generated after Step 9, and I waited for an email from 4pcb.com with the DFM report. The DFM checker found several errors. Many of these occurred because of traces and pins being too close together, so some traces were tweaked to pass the DFM checker. These DFM errors were reduced in a few iterations by Evan Respaut and me. When there were no significant DFM errors, the layout files were uploaded to 4pcb.com's website, and Luke had Violet's purchasing personnel put the order through for three boards. The boards cost ~$1000 at this quantity, and Luke, the Electrical sub-team lead, determined three boards is all we would need for the Final Competition Review of the Nanosat-6 competition in January 2011.

Discussion

In this section, I discuss some of the issues I had completing revision 1 of the layout. As I said in the opening page of this report, many of the footprints were incorrect when I started to layout. Evan Respaut updated most of the footprints because he was familiar with them, and he was more familiar with adjusting the footprints' padstacks. I learned from him how to modify footprints and made changes to a few of the footprints myself. The DC-DC converters and the off-board connector footprints all had their footprints modified. I originally did not expect to have to change any footprints, so this was one of the unanticipated developments of PCB layout.

Several component footprints had to be modified after I had started routing because the footprint made it impossible to route in its current placement, or because the footprint size was too small for the current that it was carrying; for example, a surface mount resistor with a ~20 mils pad size was supposed to carry 3 A peak current, which requires a trace width > 100 mils (these numbers are not exact, just for example purposes). I think such problems could have been avoided if the Violet team member who was responsible for selecting parts for the power board had done the current calculations.

Changing component footprints after routing had been started was unexpected. To me, it showed how important it was to consider layout and the actual board design when you are choosing parts because, if you don't consider layout early, parts will have to be changed out as they did for this revision of the Power Board. Many of the components that had their footprint changed were passives, like the resistors and capacitors that are used throughout the power sensing and switching circuitry. Luke and Evan pointed these components out to me. Also, a few of the DC-DC converters did not have capacitors to properly bias them according to their datasheet, so these capacitors were added. I noticed discrepancies like this one by examining the schematic as I was routing traces connected to the DC-DC converters, and I looked up the datasheet because I did not think the schematic was correct. This was a mistake made in previous semesters of power system development that I was able to correct during the layout stages.

After autorouting once, I found that the area around the analog MUXs was very densely populated, and the high density was actually preventing the autorouter from completing all the routes. So I moved the analog MUX that takes the solar current sensors as input to the bottom-right corner of the board, where it is located in the final layout, in order to reduce the density in the middle of the board. I ran the autorouter again, and this time it completed most of the routing.

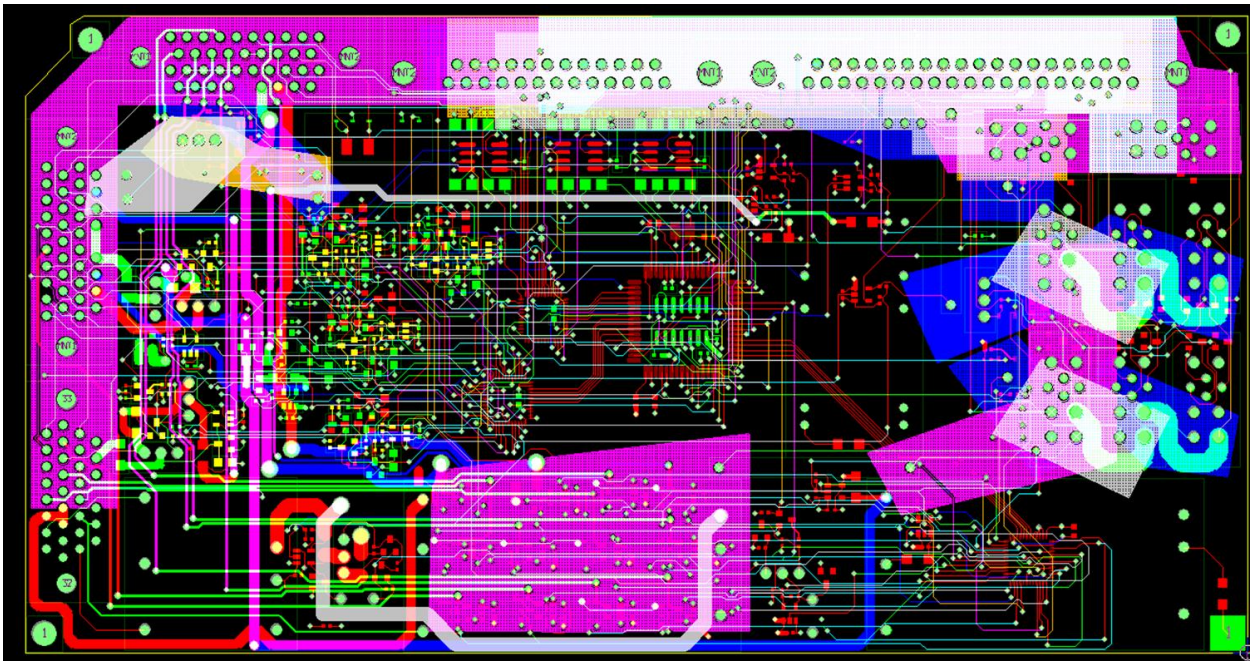


Figure 2: Layout of Power Board, Revision 1 (screenshot in OrCad)

Testing of Revision 1 Board

I tested the Revision 1 board for the correct electrical connections and no unexpected shorts. In addition, the voltages output by each of the DC-DC converters was measured and equaled the designated supply voltages. Other than a few mistakes found in the schematic, the

Revision 1 Board was designed and made correctly. Luke Ackerman also conducted tests required by the Violet team since the Revision 1 board is considered a piece of flight hardware.

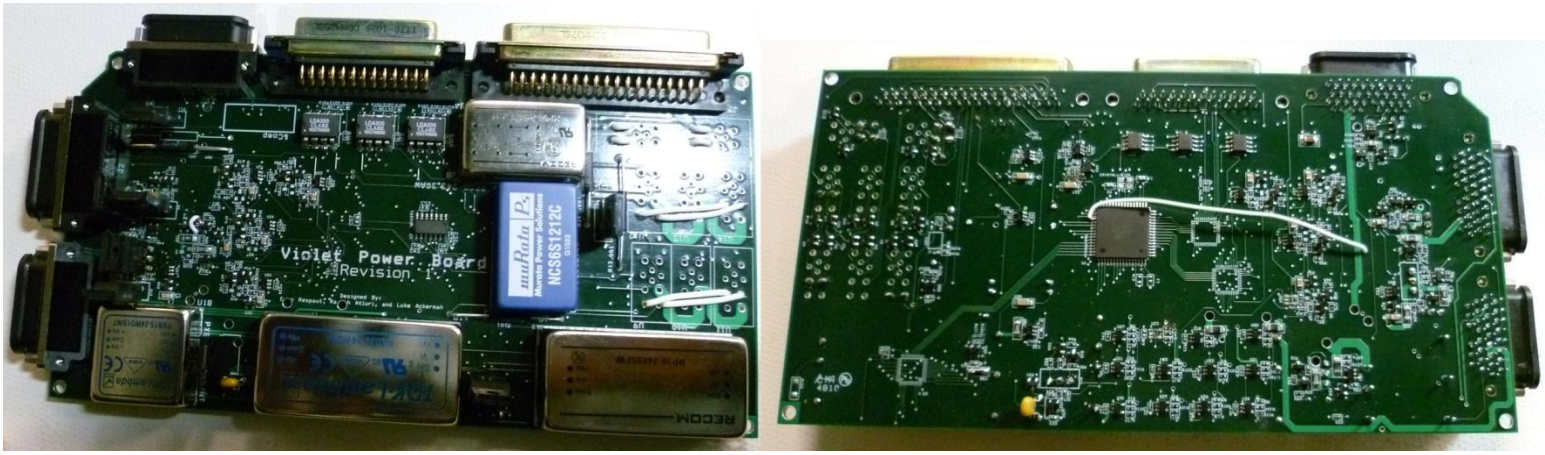


Figure 3: Populated Revision 1 Power Board for Flat-Sat Test

POWER MCU SOFTWARE

Many of the functions for the power subsystem require the ability to monitor and record voltages and currents. Safe battery charging also requires an implementation of a charging algorithm. For these reasons, the power system employs a programmable microcontroller unit (MCU). With the aid of multiplexors, one MCU is enough to monitor all the currents and voltages for the entire satellite. In addition to monitoring voltages and currents, the MCU can control all the individual power lines to each component through switch circuits. For a higher degree of integration, the MCU is mounted on the power board itself.

Thus, the Power MCU has the important role of power regulator and power controller. It maintains safe battery charge level at all times and routes power to components. The MCU also samples current, voltage, and temperature using sensor circuits and can detect any faulty behavior of satellite components that is indicated by irregular current, voltage, temperature or power consumption, which can harm the satellite as whole. In addition to providing regulated power, the Power MCU also allows the Flight Computer and the ground station to control switches connecting batteries to components. The Flight Computer has stronger processing power and can communicate with the Power MCU to turn on/off any component. When toggling a switch is desired, the Flight Computer can simply route command through CDH Board to the Power MCU.

Design Requirements

Most of the functions that need to be implemented on the Power Board MCU are derived from the Power system requirements listed in VIOLET-PWR-001 and explained in the Power Board design document VIOLET-PWR-003. The following bulleted list summarizes these functional requirements of the MCU software:

- Read current, voltage, and temperature sensors by controlling the analog MUXs and sampling with the analog-to-digital converter of the MCU. The values should be scaled by a conversion factor appropriate for the range of expected analog values.
- Control the switches on the power lines to each subsystem.
- Compute current to the batteries and calculate the state of charge of the batteries.
- Charge and discharge the batteries with the power from the solar panels.
- Control the magnetic torque.
- Form data packets and communicate with the CDH Board according to the Violet Communication Protocol (defined in a Violet CDH document).
- Allow the Flight Computer or the ground station (through the CDH board) to tell the MCU which switches to turn on/off.

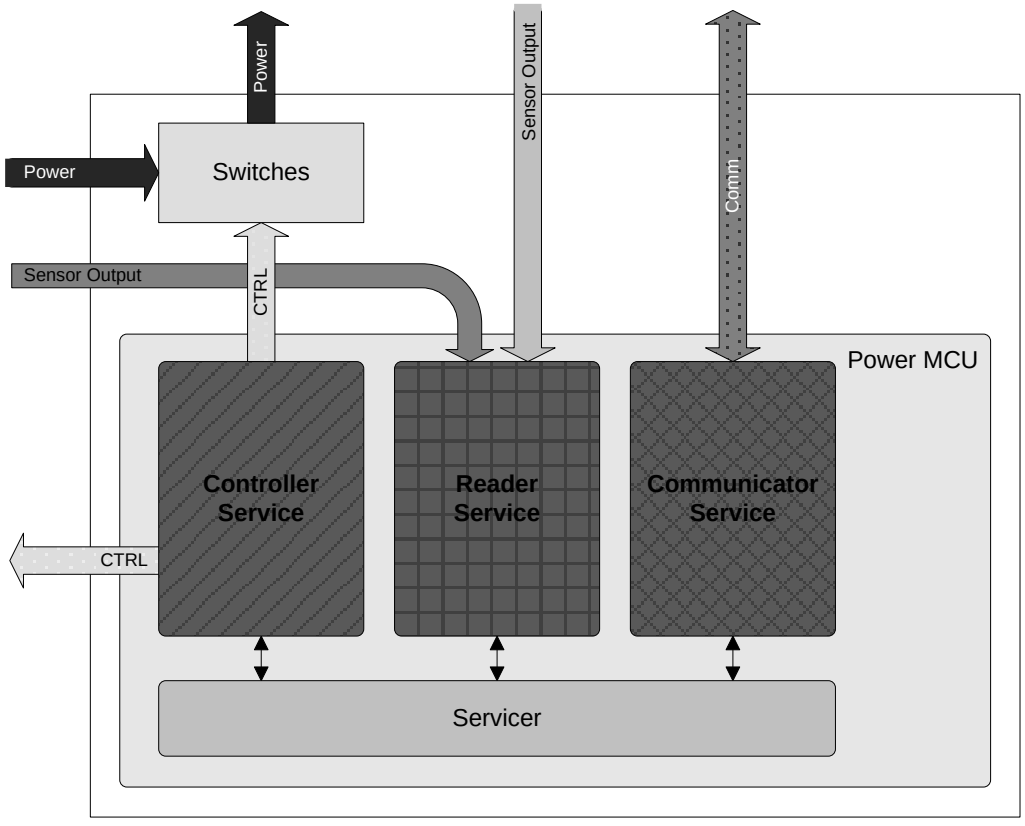


Figure 4: Services on the Power MCU

Software Design & Implementation

The full, commented Power MCU code appears at the very end of the Appendix. The code includes functions to accomplish the three major functional roles of the MCU shown in Figure 4. The following five sub-sections of the report describe the design of the C code that is implemented for the Power microcontroller. It is written in enough detail to serve as a guide to the next engineering student who enhances the Power microcontroller code. Variable and function names from the code are written in blue to make them stand out.

Task Scheduler

The file *power.c* is home to the main function of the Power code. This main function enters a never-ending while loop that calls various functions at the appropriate time intervals after executing the initialize function. The initialize function sets the initial values of the input/output ports of the MCU, sets the UART baud rate and other register settings, and sets starting values for several global variables. Note that the communication service employs a state machine that starts off in the state where the Power MCU is ready to receive VCP commands and the getPacket function call enables the UART receive ISR. This start state is necessary for the MCU since it must receive VCP commands from the Flight Computer when it turns on.

The timing of the top-level control, sampling, and communication functions relies on compare-match interrupt service routines (ISR) of hardware Timer 0 and Timer 1, both of which use a 16 MHz crystal as a clock source. Based on the Timer register settings, the Timer 0 compare-match ISR decrements time1, a software counter variable, every 11.75 µs, and the Timer 1 ISR decrements the other software counter variables every 1 ms. These software counters are reset each time they reach 0 in the never-ending while loop of the main function. Thus, this multi-tasking embedded program calls on the appropriate top-level functions at specific time intervals shown in Table 3.

Table 3: Timing of the MCU's Top-Level Functions

Power MCU Service	Function Name(s)	Period of Function Call
Sampling	read_VIT, storeValue	2.94 ms
Communication	vcp_comm	10 ms
Control	switchControl, torqueControl	50 ms

As part of the sampling service, the read_VIT and storeValue functions sample values from the current, voltage, and/or temperature sensors of one subsystem/component of the Violet satellite and update the corresponding value stored in the MCU memory, respectively. Due to time1's reset/initial value, this pair of functions is called every 2.94 ms approximately. The top-level communication function vcp_comm is called every 10 ms based on the timing of the time2

counter. The state machine implemented in vcp_comm needs to be adjusted frequently—hence 10 ms—in order to implement reliable UART communication. The top-level control functions switchControl and torqueControl, which affect the state of the subsystem/component power switches and the magnetic torquers respectively, are called every 50 ms, giving sufficient function calls per second to react to VCP commands from the Flight Computer.

SVIT: The Array of SVIT_t

Declared in *power.h* and defined in *power.c*, the SVIT array is an array of the state of the 36 Violet components of concern to the Power subsystem. Such an organized data structure is necessary because the different services of the MCU must both know the current state of a component's switch or sensors and update the state of a component's switch or sensors after one of the services has completed an action. The SVIT array element is a struct called SVIT_t:

Table 4: Data contained in a SVIT_t struct

Variable Name	Size in bytes (type)	Description
name	8 (char [])	the shorthand subsystem or component name
switchNum	3 (char [])	the port pin connected to this component's power switch
S	1 (char)	the current state of the switch {1 = on, 0 = off}
error	1 (char)	the byte used to encode the detected errors
Vmux	1 (char)	the ADC/PORTF pin of the MCU that this voltage sensor's analog MUX is tied to
VmuxBit	1 (char)	the input of the MUX that this voltage sensor is tied to
V	2 (int)	the most recently stored value for this voltage
ScaleFactorV	2 (int)	the conversion factor for this voltage signal from the digital value output by the ADC to the value stored in memory
Imux	1 (char)	the ADC/PORTF pin of the MCU that this current sensor's analog MUX is tied to
ImuxBit	1 (char)	the input of the MUX that this current sensor is tied to
I	2 (int)	the most recently stored value for this current
ScaleFactorI	2 (int)	the conversion factor for this current signal from the digital value output by the ADC to the value stored in memory
Tmux	1 (char)	the ADC/PORTF pin of the MCU that this temperature sensor's analog MUX is tied to
TmuxBit	1 (char)	the input of the MUX that this temperature sensor is tied to
T	2 (int)	the most recently stored value for this temperature

Note that there is no "ScaleFactorT" stored in SVIT_t because all the temperature sensors share the same conversion factor, and this common conversion factor is defined as the constant TempScaleFactor in *power_sample.c*. Thus, each component requires 29 bytes to keep track of its state, and the entire 36-element SVIT array occupies 1044 bytes of the MCU data memory (SRAM), a significant overhead. The following three sub-sections on the different services of the

Power MCU will describe how each variable in the SVIT_t struct is used to keep track of the state of a component's power switch or sensors. The initial values of the two-dimensional SVIT array, which is defined in *power.c*, appears in Appendix II.

Design of Sampling Service

The Sampling service reads voltage, current, and temperature sensor outputs from Violet components and stores them into the Power MCU memory. In *power_sample.c*, the top-level functions read_VIT and storeValue sample values from the sensors of one component and update the component's corresponding values stored in MCU memory with the just-sampled data, respectively. Since this pair of functions is called every 2.94 ms approximately and operate on the sensors of a single component, it takes about 100 ms to update the stored sensor values of the 34 components of interest to the Power subsystem. Since the flight computer will request to read sensor values at most once per second (1 Hz), sampling each sensor ~10 times per second is an adequate rate. Note that the two batteries are not included in this count because their stored sensor values are updated every 2.94 ms, each time the pair of functions is called, in order to have sufficient samples per second to compute the NiCad batteries' state of charge accurately.

In read_VIT, index_svit is the variable used to index through the SVIT array. The first if statement resets index_svit to 0 if it exceeds the size of the array and also increments it by 1. The next two if statements increment index_svit, so the two battery components are skipped over. The next couple statements use the MCU's analog-to-digital converter (ADC) in the function read_VIT_helper to store the digital value of the ADC conversion into variables currentValue, voltageValue, and tempValue. Since all components do not have temperature sensors, the if statement before the tempValue line checks if there actually is a temperature sensor to sample from. The read_VIT_helper function takes the ADC pin and the input of the analog MUX—variables from this sensor's SVIT_t struct—as arguments in order to electrically connect the desired sensor to the selected ADC pin, which is carried out by selectMUX. The first argument of selectMUX indicates which analog MUX, or ADC pin, is to be selected, and the second argument decides the select input bits of that analog MUX. The rest of read_VIT_helper adjusts the ADC registers to sample in a blocking manner and returns the conversion's digital value. The 2.94 ms period of read_VIT is long enough that there should be no significant delay due to the ADC blocking since the ADC clock frequency of 16 MHz / 128 = 125 kHz implies blocking should last about 120 µs. At the bottom of read_VIT, the ADC conversion process is executed for both batteries.

In storeValue, the digital value retrieved from the ADC is first multiplied by the sensor's conversion or scaling factor—a variable in the component's SVIT_t struct for voltage and current sensors—to get a "scaled" digital value. setValue takes this "scaled" value and a pointer to a COMPONENT_t struct corresponding to the current component, or index_svit value, as

arguments. The instance of the COMPONENT_t struct holds an array of the last 8 samples taken. setValue replaces the oldest of these 8 samples with the new "scaled" digital value and computes a new return value, which is stored in the appropriate variable of the SVIT_t struct in the higher-level storeValue function. The return value is the simple moving average of the most recent 8 samples, including the newly added, "scaled" digital value. The rest of the storeValue function repeats the process for the two batteries.

Filtering for the SVIT Array's Stored Value

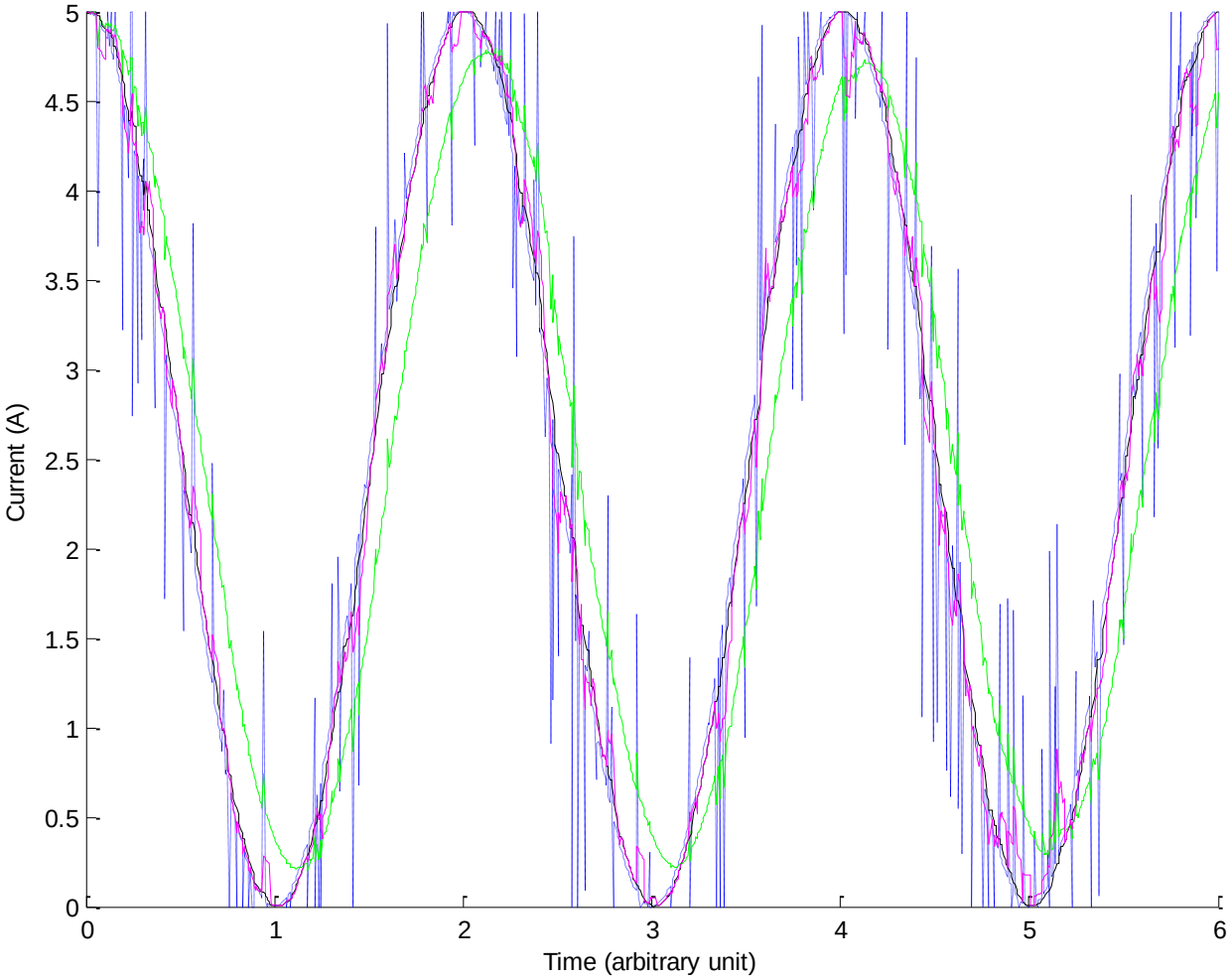


Figure 5: Different filters applied to noisy sinusoidal signal with many outliers (dotted blue); Green – SMA with window size of 7 points; Magenta – SMA excluding local outliers; Black – Simple Moving Median

It is important to store an accurate value of the measured sensor signal into the corresponding element of the SVIT array. Hence, the value stored in the SVIT array is actually a simple moving average (SMA), effectively a low pass filter on the sensor output. A SMA is still

heavily influenced by outliers, which can be caused by impulse noise, sensor malfunction, or other temporary effects. To illustrate why outliers have a significant effect, I made a Matlab simulation of a sinusoidal, time-varying signal with several discrete points that act as outliers. This signal and outputs of three different kinds of filters are shown in Figure 5. The code for the Matlab simulation appears in the Appendix. Clearly the SMA in green does a poor job of tracking the sinusoidal signal. The SMA in magenta, which excludes outliers out of the last 7 points that are more than 3 standard deviations from the mean of the sample of the last 7 points in the window, has several points shifted if there are several noise spikes in the local window. The idea of this filter is that the SMA calculation should not take into account the noise spikes; however, if there are several noise spikes, this filter's output can be affected by them. The simple moving median (SMM), which outputs the median of the last 7 points seems to be the best filter since the noise spikes are never found to be the median of the local window. This kind of filter can be used if there is a concern that noise spikes could corrupt the SMA, which is a legitimate concern for some of the Power subsystem's sensors.

Therefore, the SVIT array keeps track of the sensor output using a SMA and a SMM. If the SMA result suggests the sensor value has gone outside the acceptable bounds, then the SMM result is used to verify that the sensor value has gone out of bounds. If the SMM is indeed out of bounds, then the Power MCU is certain the sensor value is out of bounds, sets the appropriate error bit, and takes the appropriate action.

Bounds Checking on Sensor Values

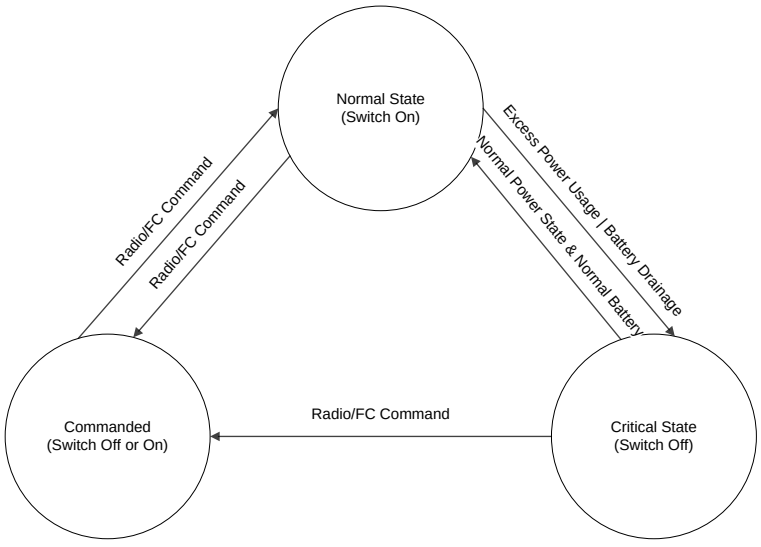


Figure 6: State diagram of switches

As suggested by the state diagram in Figure 6, the MCU controls a power switch according to how much power the corresponding subsystem/component draws. For example, if

the critical state is reached, the MCU turns that switch off because its subsystem is drawing excessive power and/or is draining the limited energy stored in the battery. Hence, the MCU must implement bounds checking on all the current and voltage sensor data.

If the sensor value is within given minimum and maximum allowable values, then the MCU will not turn the switch off, i.e. the switch is in the normal state. Exceeding these allowable value bounds causes the MCU to turn the switch off and enter the critical state. If the sensor value is within the given minimum and maximum allowable values and outside a more narrow set of bounds, then the MCU will raise a flag as a sign that the corresponding subsystem is consuming less or more power than is expected under normal operation. The Flight Computer will view this as a warning, and the MCU will not immediately take control of that switch. If the sensor value is within the narrow set of bounds, then the subsystem's power usage corresponds to the conditions of its normal operation.

Design of Control Service

The top-level control functions switchControl and torqueControl are written in *power_control.c*. The switchControl function sets the value of a port pin that is tied to a component's power switch based on the first argument, which is the index of the component, and the second argument, which decides if the switch is to be on or off. If the argument onOffFlag is 1, then the port pin is set to 1, turning the component's switch on; if onOffFlag is 0, then the port pin is set to 0, turning the switch off. Most of the switchControl function consists of a switch control structure that is used to select the correct port pin based on the switchNum variable of the component's SVIT_t struct. The SET and CLR macros are used to make the actual setting of the port pin more accessible to the code's reader. The switchFound flag, which is defaulted to 1, indicates if the port pin was found in the switch control structure and set; if the port pin is not found, switchFound should be 0. The last action of switchControl is to set the value of the S variable of this SVIT_t struct, given that the correct switch was found.

The torqueControl function takes three 8-bit inputs that set how the three magnetic torquers on the Violet satellite should be driven by pulse width modulation (PWM). These 8-bit arguments must be sign and magnitude numbers. The most significant bit decides what direction to drive the torquer, i.e. {0 = positive, 1 = negative}. The lower 7 bits determine the magnitude on a scale from 0 to 127. Each torquer has a software counter used to set its PWM signal, and the if statements at the beginning of torqueControl simply reset these PWM counters if they get above 127. If the software counter is less than the input byte, the magnetic torquer can be driven in three different ways. If the sign of the last input byte, which is stored as t*n*_pwm_on (*n* is the number of the magnetic torquer), differs from the current input byte, then the torquer is driven into a brake state, so it can change directions later. Then if the sign of the input byte is negative, the MCU sets the A input of the H-bridge that drives the torquer in the counter-clockwise

direction. If the sign is positive, the MCU sets the B input of the H-bridge that drives the torquer in the clockwise direction. In the case where the software counter exceeds the input byte value, the MCU sets the H-bridge into a free-wheeling state, so no additional torque is generated. The actual setting of the H-bridge inputs is accomplished by the drive_torq*n* functions, which set the MCU port pins tied to the H-bridge inputs to achieve the torquer state specified by the argument of drive_torq*n*.

Design of Communication Service

The code related to the communication service of the MCU appears in *power_comm.c*. The top-level communication function vcp_comm implements a finite state machine that determines whether the MCU can transmit or receive a VCP packet at any given time. The Violet Communication Protocol (VCP) is a reliable communication protocol implemented for inter-chip communication on the Violet satellite. The following sub-section gives a brief introduction to VCP and its packet structure.

Primer on Violet Communication Protocol (VCP)

VCP is a KISS-based (stands for "keep it simple stupid") communication protocol that encloses data packets inside a KISS frame. This means VCP packets start with a special start byte (0xC0) and end with a special end byte (0xC0). Inside these two KISS frame bytes, a VCP command packet from the Flight Computer to the Power MCU will have the structure shown below in Table 5. The "source" byte is the address that the packet is supposed to go to if the packet originates from the Flight Computer, or the address that the packet is sent from if it is going to the Flight Computer. The length byte contains the length of the entire VCP packet. The CRC byte is the cyclic redundancy check error-detecting code that is computed based on the bytes in the data packet. For command packets sent to the Power MCU, the first byte of the data field will be command byte as shown in Table 5, and the data bytes will contain whatever additional arguments that command needs to execute. For packets sent by the Power MCU, e.g. the telemetry packet of the state of the switches and sensors, the only difference is that there is no command byte, so the data bytes occupy the space between the length and CRC fields. Appendix III shows a table of the VCP commands that the Power MCU can receive.

Table 5: Packet structure of a VCP Command Packet

Packet	**PWR Switch Control Protocol**				
Field	Source	Length	Command	Data	CRC
Subfield	1 byte	2 bytes	1 byte	N bytes	2 bytes
Packet Size	6 + N bytes				

Finite State Machine for Communication

The state machine shown below explains the behavior of the communication service. There are five states that depend on the values of rx0_ready, tx0_ready, vcp_newACK, and vcp_newTEL. rx0_ready is a flag that indicates that a new packet has been received when it is 1. tx0_ready is a flag that indicates that a new packet has been sent, and the transmitter is not busy when it equals 1. vcp_newACK is a handshake between the receive and transmit modes set to 1 after the Power MCU has received a VCP command. Similarly vcp_newTEL is a handshake between receive and transmit modes set to 1 every one second when the MCU needs to send a telemetry packet informing the Flight Computer about the status of all the switches and sensors.

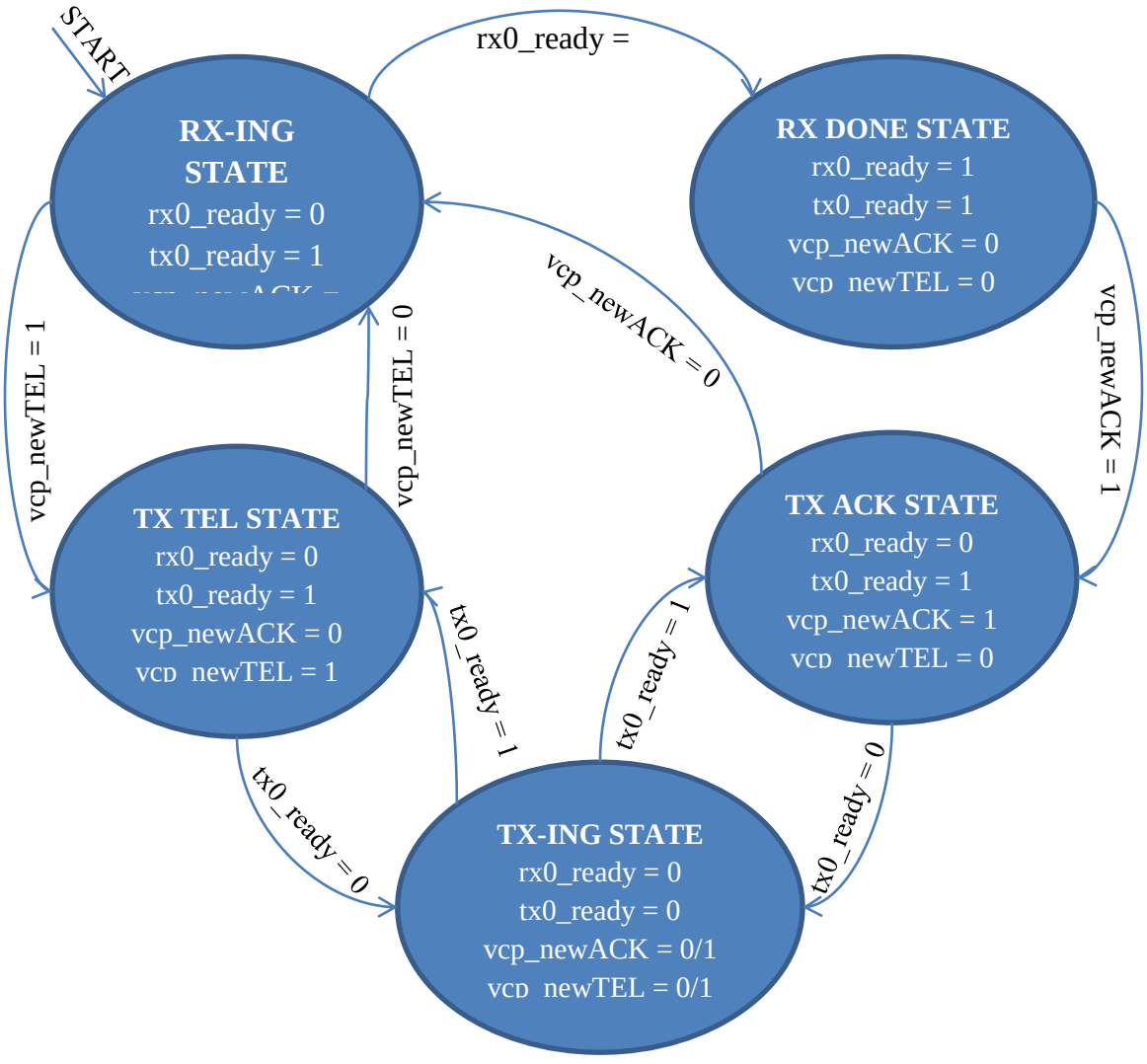


Figure 7: Finite State Machine of VCP Communication over a single USART

Initially, the MCU is in the RX-ing state, i.e. rx0_ready = 0, tx0_ready = 1, and vcp_newACK = vcp_newTEL = 0. The states of the communication service are:

- RX-ING: the MCU is ready to receive a VCP packet and will enter the UART RX ISR when prompted by a newly received byte
- RX DONE: the MCU is done receiving a VCP packet, i.e. the last byte has been received
- TX ACK: the MCU is about to transmit an ACK (acknowledgement) packet in response to a VCP command packet that was recently received and executed
- TX TEL: the MCU is about to transmit a TEL (telemetry) packet, which it periodically sends to inform the Flight Computer about the status of the Power subsystem
- TX-ING: the MCU is ready to transmit a packet and enters the UART UDRE ISR

Receiving a VCP Packet

In the USART0 RX ISR, bytes that are received in the UDR0 register are written to the message field of the vcp_ptrbuffer instance PWR0 using the vcpptr_rx function, which is defined in *vcplib.c*. When the last (0xC0) byte of the packet is detected by the if statement in the ISR, the RX ISR is disabled, and the rx0_ready flag is set to 1. So, the next time vcp_comm is called, the MCU executes the rx0_ready branch of the control structure, calling on the parseMessage and getPacket functions in that order. The parseMessage function extracts the data bytes of the packet and adjusts global variables to execute the desired command. The switch control structure in parseMessage accounts for the various VCP commands shown in Appendix III. getPacket clears the rx0_ready flag since the receiver is no longer busy and then enables the receive ISR. Because the Power MCU ought to send an acknowledgement (ACK) of all the commands that it has received to the Flight Computer, the handshake variable vcp_newACK must be set to 1 after getPacket is called, so the service can enter the next state where it sends an ACK packet.

Sending a VCP Packet

Whether an ACK packet or a telemetry packet is being sent, the sequence of events is pretty similar. The major difference is the vcp_newACK handshake is used to enable transmission of ACK packets, and the vcp_newTEL handshake is used to enable transmission of TEL packets. Note that, due to the if/else if control structure of the state machine, the TX ACK state always takes precedence over the TX TEL state; that is, if vcp_newACK and vcp_newTEL are both 1, then the TX ACK code that executes for vcp_newACK = 1 will execute first. The functions writeACKMessage and writeTELMessage write bytes of the message into a finite sized message buffer, tx0_buff.

The function putPacket writes whatever is in tx0_buff to the message field of the vcp_ptrbuffer PWR0 and writes one-byte-at-a-time to the UART using the USART UDRE ISR. tx0_ready is set to 0 to indicate that the transmitter is going to transmit, and tx0_index is reset in order to start reading from the beginning of the tx0_buff array the next time it is read. The instance of vcp_ptrbuffer PWR0 must be cleared in case there is still data from a received packet, and the address of the VCP packet to be transmitted must be set before calling the vcpptr_tx function. In the first while loop, the bytes of tx0_buff are then written into the message field of PWR0 using the vcpptr_tx function. Then in the second while loop, the bytes of PWR0's message are written to UDR0 using the writeByte function that enables the USART0 transmit ISR. PWR0's message buffer is set up to be circular in order to write bytes to it without fear of bytes being dropped. Lastly the appropriate handshake variable is cleared, and the instance of the vcp_ptrbuffer is cleared in the vcp_comm function.

Communication from the Power MCU to the Flight Computer

The MCU generates a report about once per second or based on the VCP command bytes 0x06 to 0x08. If it's sending out a packet based on a VCP command, the MCU echoes back the command byte followed by the requested data. When the MCU gets a control command from the Flight Computer rather than report command, the MCU will still transmit an acknowledgement message to the Flight Computer, which a byte that shows whether operation was successful or not followed by the received command.

When the MCU is commanded to report all the sensor output, it generates a packet for each of the 36 subsystems/components instead of a single packet containing all the sensor outputs. Such smaller packets must be used because the MCU cannot handle packets larger than 255 bytes. Each packet contains all the data values stored for that subsystem/component at the last sampling time before the incoming command was processed by the MCU. Thus, the flight computer will receive data at the same level of detail as seen by the Power MCU although it will be delayed.

Discussion

The controller functions of the Power MCU have been verified to be functioning correctly. The recently-added reader and communicator functions have been tested on the STK500 board using test setups that model the interfaces of flight hardware. Several necessary features, such as bounds checking and scaling factor for the ADC conversion, have been included in the sampling function code although specific numerical values have been excluded. These numerical values can only be determined through sampling actual sensor values, which lies outside the scope of this project and my responsibilities on the Violet team.

The battery charge-monitoring algorithm and associated functions need to be added to the MCU code since the state-of-charge algorithm has yet to be determined. Further testing must be done with the Power MCU under "flight-like" conditions. That is, the Power MCU should be programmed on a fully-populated Power Board, and the Power Board needs to be interfaced with the sensors and the CDH interface board. The CDH interface board should be connected to the Flight Computer or another machine acting like a Flight Computer. Since Revision 2 of the Power Board will arrive soon, this "flight-like" test should be done with the Power MCU on the Revision 2 board. I will be staying over the summer to hand off the Power MCU project to another Violet team member, who will most likely be the person to conduct these tests.

TESTING RESULTS

Testing of Sampling Service

Originally I had planned to do all my testing on the Power Board Rev. 1 with a MCU programmed with my final version of the code. But because some of the components—for example the off-board connectors and the analog MUXs—are pretty expensive, the Violet team used some critical Rev. 1 components on the Rev. 2 board. Since the Rev. 2 board is considered flight hardware, special handling and tests must be done before it is used for system-level tests due to the AFRL's University Nanosat Program rules. The team members trained to complete these tests were not able to finish the flight hardware qualification procedures before they left Ithaca for the summer because the Rev. 2 populated PCB did not arrive. Unfortunately this limited my access to the hardware that I would have liked to test on. Therefore, I conducted tests on an STK500 and made interfaces that modeled Violet's actual interfaces when needed.

I was not able to set up the actual hardware structure of the sampling service, i.e. the 100+ current, voltage, and temperature sensors connected to the ADC0, ADC1, and ADC2 ports of the MCU via 32-input analog multiplexers, since I was not able to use the analog multiplexers and all the sensors. Two simple tests were completed that showed the proper operation of the MCU with regard to sampling the sensors.

First, I recorded the signals on the output ports of the MCU, which will be tied to the ADC0 analog multiplexer's select inputs, and I found that the select bit values were being adjusted at the correct rate corresponding to how often the read_VIT function is called. The traces of these ADC0 MUX select bits is shown in Figure 8. Note that the time-varying pattern of how the five bits count is not from 0 to 31. The counting pattern of these bits depends on the SVIT table's entries. Remember that each time the read_VIT function executes, it increments the SVIT index by one, and each read_VIT function call samples a current sensor, a voltage sensor, and perhaps a temperature sensor. Each of these sensors could be on any one of the three analog MUXs, so sometimes read_VIT never adjusts the MCU outputs that connect to the ADC0 mux;

hence, the relatively long flat-line portions in Figure 8. In summary, the signals correspond to the MUX select bit values reading horizontally (each read_VIT call) and going down (increment the index variable) on the SVIT table, only including the ADC0 MUX's sensors.

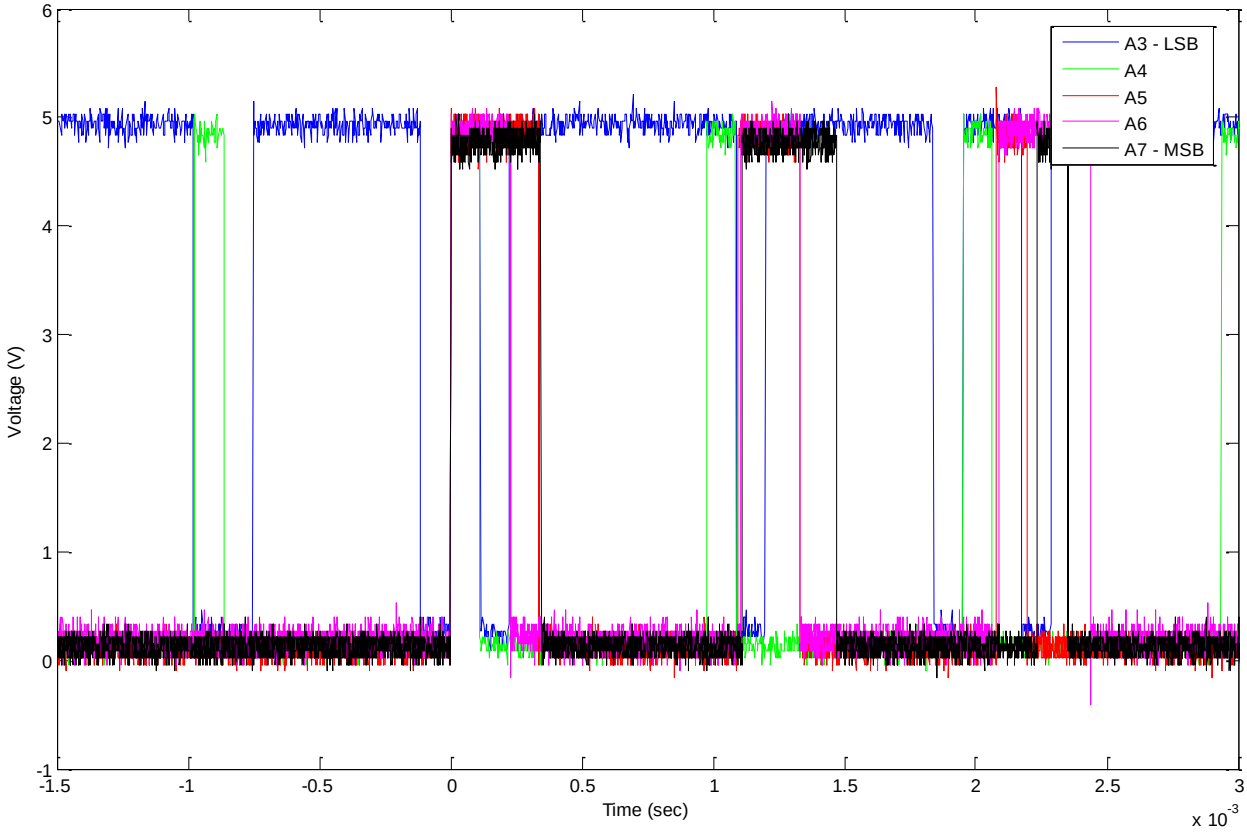


Figure 8: MCU output signals that connect to ADC0 MUX's select inputs measured with scope

Second, a voltage sensor circuit was connected to the MCU's ADC0 pin. This sensor basically looks like a voltage divider made of resistors to the MCU. The actual voltage range measured by the sensor must be scaled down by the voltage divider to a level acceptable to the MCU. An oscilloscope was used to measure the value of this sensor's output as the input of the sensor was turned on and off over time. The blue line in Figure 9 represents the sensor's output voltage, and the green line corresponds to the value computed by the simple moving average filter and stored in the SVIT table of MCU memory. The green line is actually the SVIT table's stored sensor value after dividing out that sensor's voltage scaling factor, so that it is in the vicinity of 2 V. As you can see, the ADC0's digital value closely tracks the sensor output signal and is less susceptible to noise thanks to the simple moving average filter. In addition, the fact that the scaled-down version of the SVIT table's value can track the sharp transitions seen in Figure 9 is a good feature because the sampling service should have a very fast response to changes in the sensor's input value.

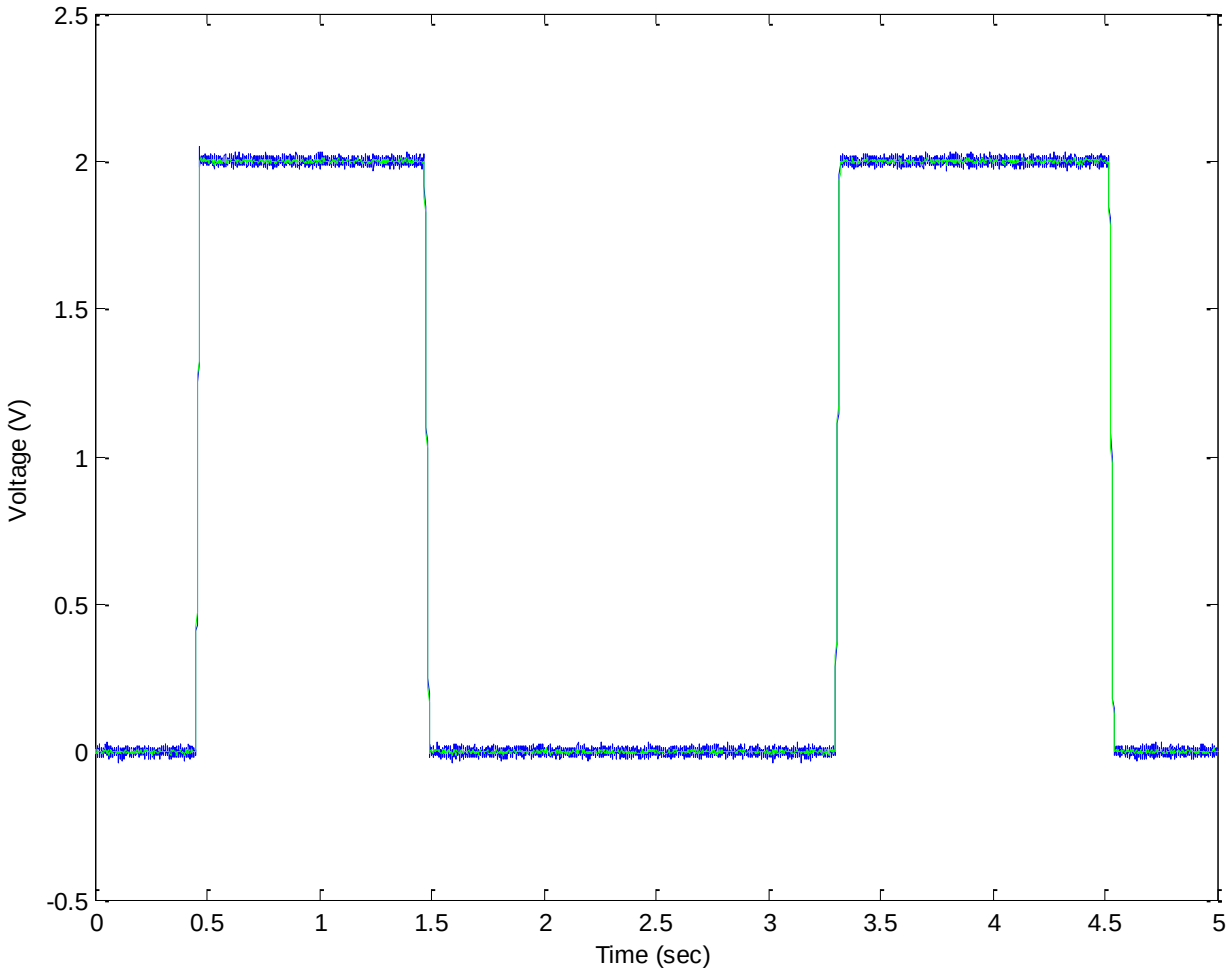


Figure 9: Sensor output in blue; This sensor's rescaled, stored value from the SVIT table in green

Testing of Control Service

The correct functionality of the switch and magnetic torquer functions needed to be verified. I tested that the switch_control function executed correctly by sending VCP commands from my computer over a serial line to the STK500. I will discuss this test in detail in the next sub-section of testing of the communication system.

A preliminary MCU function test was done on the Rev. 1 Power Board in January 2011 in preparation for the University Nanosat Program's Final Competition Review (FCR). The switch_control function was completed by then and has remained unmodified. The Violet team members who were preparing for FCR made a "Flat-sat" ("sat" for satellite) setup for testing. This basically means all the computers and data handling hardware on the satellite was set up on the lab bench in order to mimic the actual electrical system of Violet. The Power MCU was actually on the Rev. 1 board and was programmed via JTAG with the MCU code that I had

completed by then. By making a turn-on call to switch_control in the MCU's main method, Luke Ackerman and I observed that the voltage on the selected switch's line rose to 12 V as expected. A DC-DC converter on the Revision 1 board served as the 12 V source. I used a multimeter to make this measurement and unfortunately did not save the transient signal with an oscilloscope. At the time, I was expecting to work more with the Rev. 1 Board after FCR.

Here, I want to explain some results that show correct execution of the pulse width modulating function that controls the magnetic torquers. The MCU drives the magnetic torquers, which are inductive elements, through an H-bridge interface chip. Due to my limited access to the flight hardware, I was not able to set up the actual torquers connected with the H-bridges. I wanted to be sure that the MCU output would drive the H-bridge inputs correctly, so I made a Matlab simulation of the code that generates the PWM signal instead. The Matlab code appears in the Appendix. In the Matlab simulation, only one torquer's PWM signal is produced since the code for control of each of the 3 torquers is identical. The torque control function actually generates two PWM signals for each magnetic torquer because the H-bridge has two inputs that each relate to one direction of the torquer's rotation. The three figures in Appendix V show correct modulation of simulated input A (shown in blue) and input B (green) of the H-bridge chip. In Appendix V.i shows the PWM signal in response to an input sequence of +32, +64, +127, +64. Since all the numbers are positive, input A is on the whole time, and input B is only turned on when the software PWM counter has reached the current input sequence value, i.e. the torquer is in the TORQUER_FREE state. That is why input B is low for most of the time in the 2-3 second window for input +127.

Testing of Communication Service

Initially it was difficult to figure out a good way to test the communication functions in the MCU code because the CDH microcontroller code was not completely written. Remember that the Power MCU talks to the Flight Computer through VCP packets, and the CDH board with its microcontroller acts as an interface that passes VCP packets to the correct destination. Because the communication functions adhered to a strict packet-based protocol, i.e. Violet Communication Protocol (VCP), the MCU communicator functions could be functionally verified by checking if the Power MCU handled messages, which resembled VCP packets, sent over a serial line correctly. Fortunately, Violet already had a Python script that had been written to send bytes across a serial line. This Python script was originally written by Jesse Thompson, a member of the CDH team, to send commands to control the CMGs (control moment gyroscopes) that steer the Violet spacecraft in space. To perform the test, I ran the script on my computer, which was connected to the Power MCU on the STK500 via a RS-232 line. I input a sequence of bytes that is binary-equivalent to a realistic VCP packet that the Power MCU could see. The script shows the packet that I sent, and it listens to the serial line for a response from the Power MCU. As shown in Figure 10, the Power MCU sends a packet in response to the sent command.

Thus, the various cases of commands/messages to and from the Power MCU were verified by employing this Python script. Appendix IV contains a table of the different types of VCP commands and the expected ACK packet if the command is interpreted correctly.

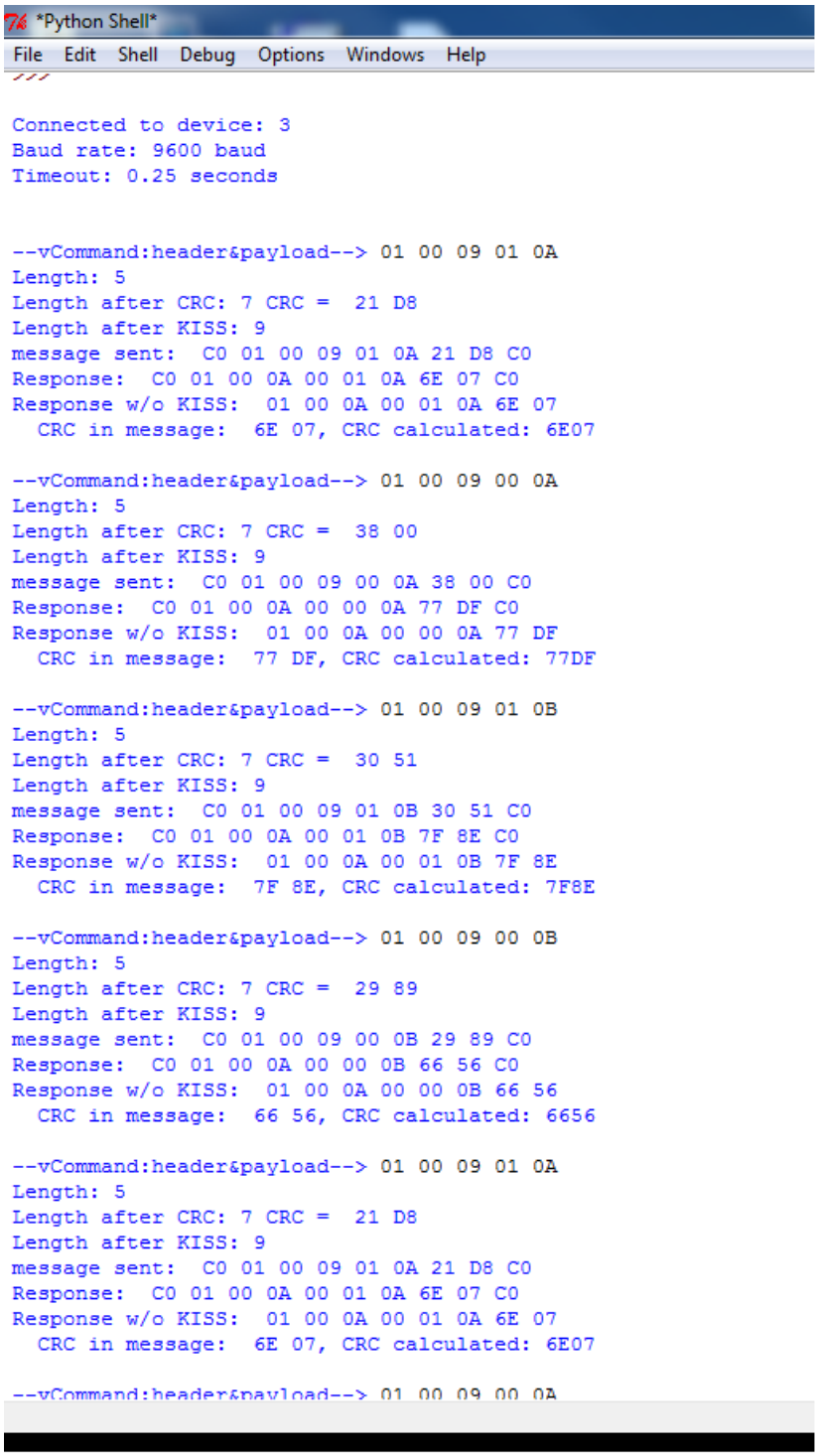


Figure 10: Screenshot of Python script sending and receiving packets from Power MCU

Because the commands sent by this Python script did not work reliably at first, I was concerned that there was a problem with the Power MCU's communication service code. I thought it could be a problem with the state machine, or it could be a problem with the transmission buffer. The MCU's response to commands was initially a little buggy. The MCU would execute a single turn-switch-on command, and this could be easily verified by connecting the chosen port pin to an LED on the STK500. But after that first command, it would execute the other commands and would not send any ACK packet or an incorrect packet. Because the switch behavior was correctly executed, I did not think there was an issue with the state machine and instead focused on the transmission buffer code. The bug in my code had to do with the tx0_buff's index counter, tx0_index. I did not reset it back to 0 at the beginning of the putPacket function, even though I wrote bytes into tx0_buff starting from the zeroth byte. Hence, the transmitter was writing a lot of blank bytes from the tx0_buff into PWR0's message and sending that out the UART for packets after the first ACK packet.

When I had corrected this bug in the transmission buffer code, I ran the Python script with the inputs and outputs shown in Figure 10. As you can see, a sequence of several VCP commands was executed correctly and the corresponding ACK packets were sent. I also took a snapshot of the signal sent out the USART0 TX pin with an oscilloscope. The two screenshots in Appendix VI were for two instances of the same command, 0x01 0x0A (close the switch of the component with SVIT array index = 10). As you can see, the transmitted ACK signals appear to be identical.

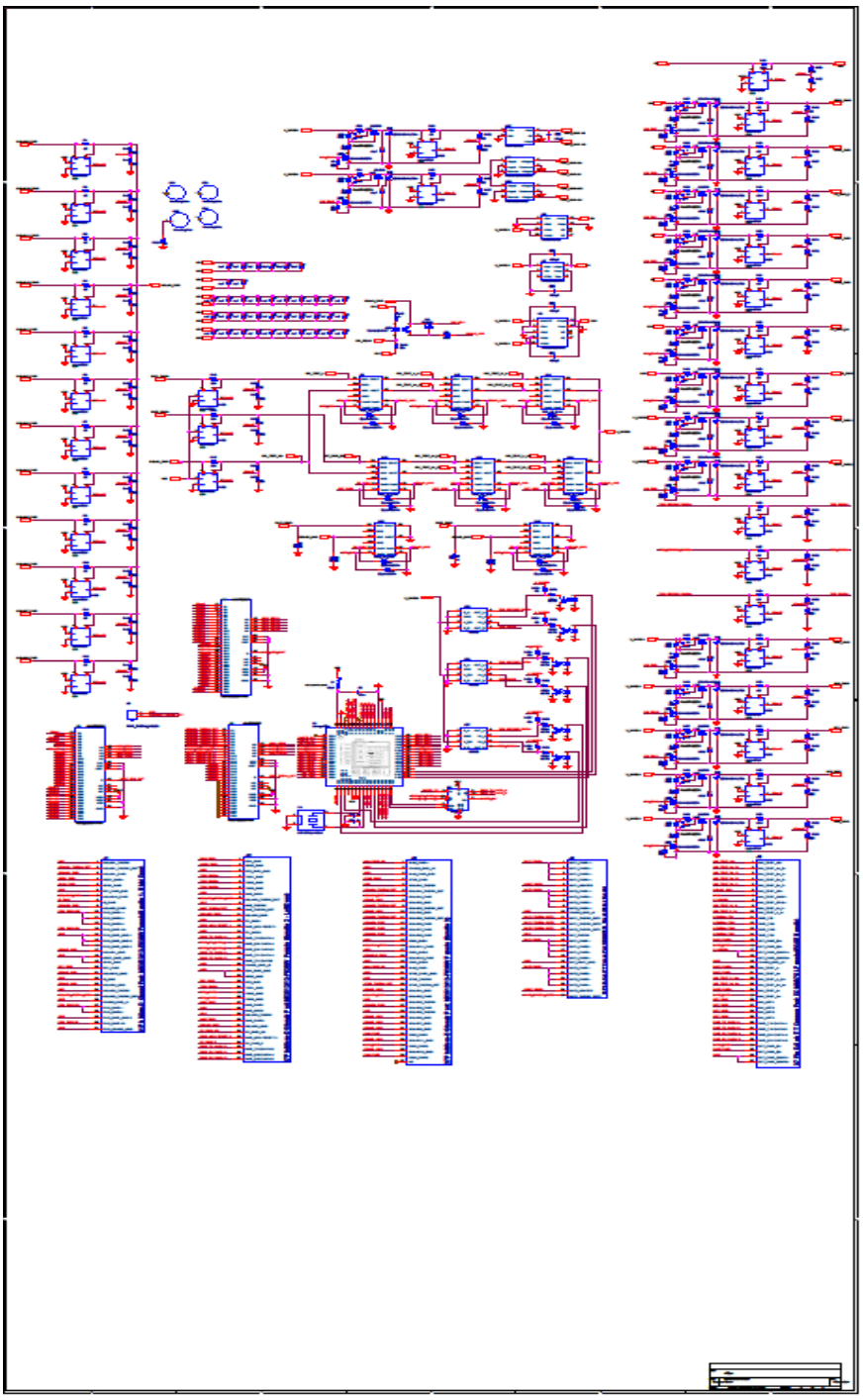
CONCLUSION

In conclusion, several key deliverables were achieved as part of my MEng project. The layout of the first version of the Violet Power Board was completed in the Fall 2010 semester, and this Revision 1 Board was fabricated and populated with most components. By performing hardware verification tests, a few mistakes were found in the Revision 1 Board, and subsequently, a Revision 2 Board layout was designed by other Violet team members. In terms of software, the latest version of the Power MCU code was verified for correct functionality on the STK500 board using test conditions modeled on operations in space. The battery charge-monitoring algorithm still needs to be added to the MCU code once that algorithm is fully determined by members of Violet's Power sub-team. The latest version of the MCU code needs to be tested under more flight-like conditions, such as testing a programmed MCU on the Revision 2 Board. The initial Power Board layout and current version of the Power MCU code represent important milestones in a reliable, centralized power system for the Violet satellite.

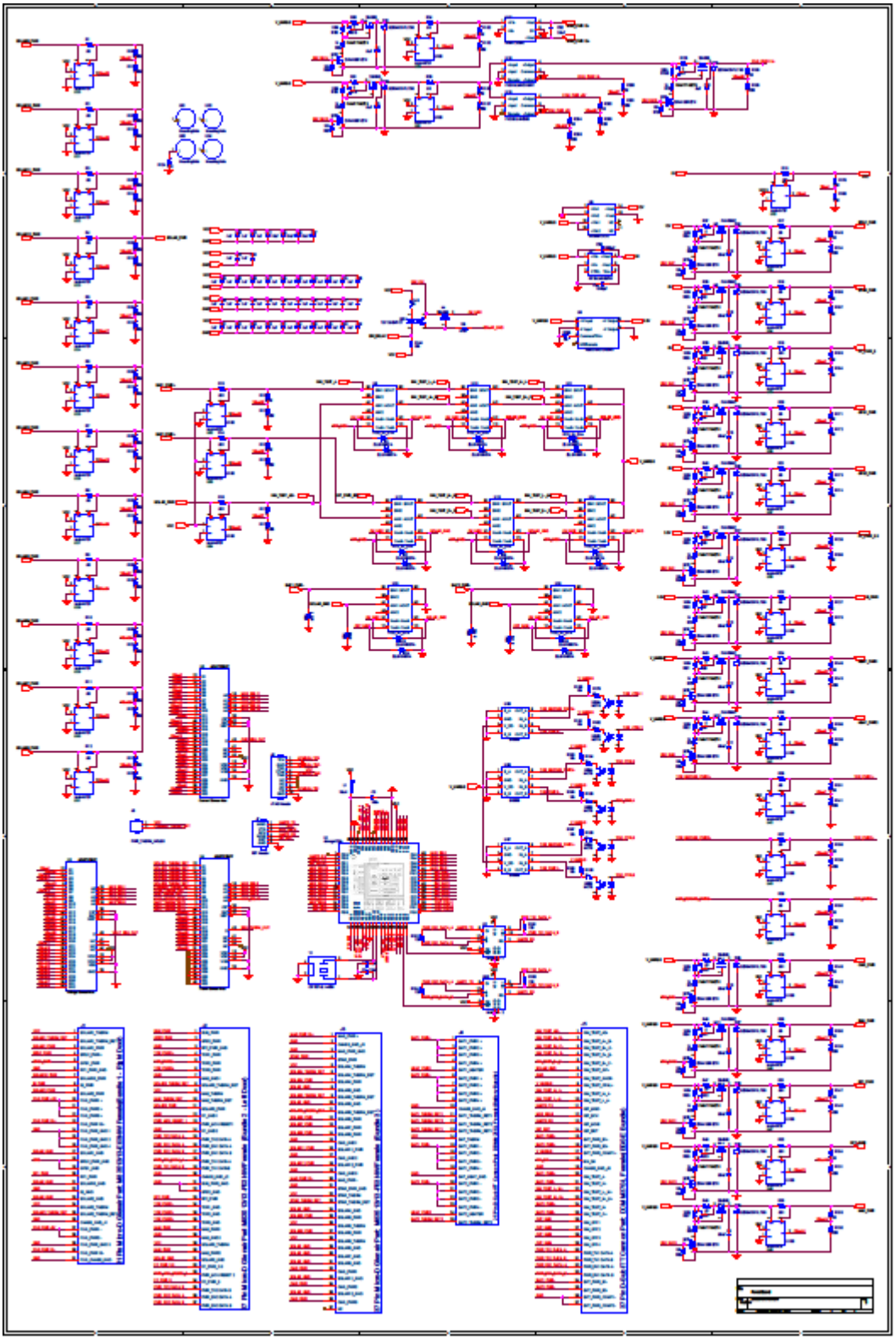
APPENDIX

I. Power Board Schematic & Mechanical Specification Drawing

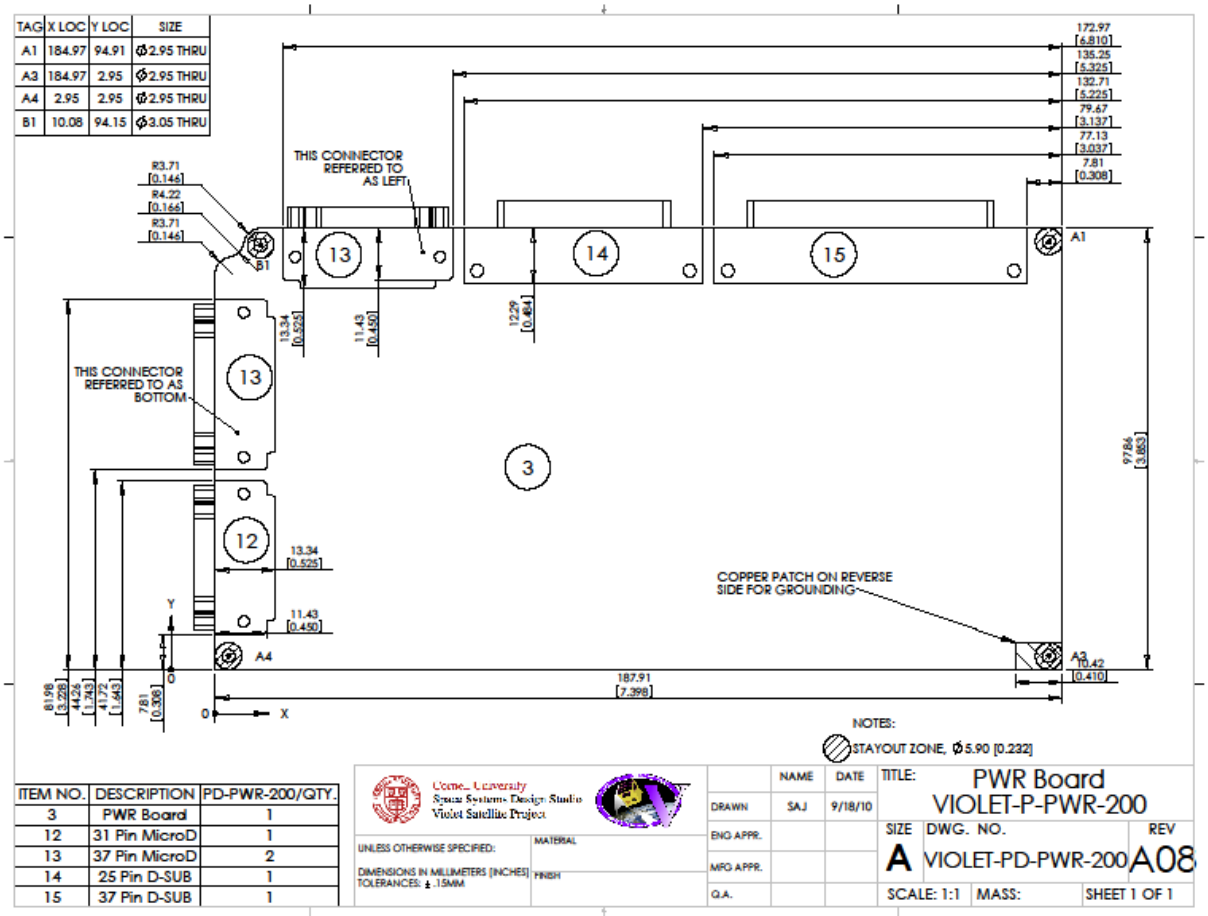
Schematic of Power Board, Revision 1



Schematic of Power Board, Revision 2



Mechanical Specification Drawing, VIOLET-PD-PWR-200



II. SVIT Table

			Switch		Voltage		Current		Temperature	
Master Index	**Component**	**Name**	*Port*	*Pin*	*Mux*	*Bit*	*Mux*	*Bit*	*Mux*	*Bit*
0	Spectrometer	spec	D	4	1	16	1	6	X	X
1	Star Tacker	star	D	5	1	23	1	7	2	7
2	FC (5V)	fc5	A	0	1	19	1	1	X	X
3	FC (3.3V)	fc3.3			2	13	2	14	X	X
4	GPS 1	gps1	A	1	1	21	1	1	X	X
5	GPS 2	gps2	A	2	1	26	1	2	X	X
6	IB	ib	B	5	1	29	1	3	X	X
7	Heater 1	heat1	B	6	1	14	1	31	X	X
8	Heater 2	heat2	B	7	1	27	1	4	X	X
9	CMG	cmg	G	2	1	15	1	8	X	X
10	Sun Sensor	sunsen	C	7	1	25	1	9	X	X
11	Radio 1	radio1	C	6	1	24	1	10	X	X
12	Radio 2	radio2	C	5	1	17	1	11	X	X
13	Maestro	maestro	C	4	1	22	1	12	2	8
14	Magnetometer	mag	C	3	0	4	0	6	X	X
15	FOG (15V)	fog15	C	1	1	18	1	13	X	X
16	FOG (5V)	fog5	C	2					X	X
17	Torquer 1	torq1	X	X	1	28	1	5	X	X
18	Torquer 2	torq2	X	X	2	15	2	16	X	X
19	Torquer 3	torq3	X	X	2	17	2	18	X	X
20	Battery 1	batt1	X	X	0	1	0	2	2	9,10
21	Battery 2	batt2	X	X	1	20	1	30	2	11,12
22	Solar (Full)	solar	X	X	0	7	0	3	X	X
23	Solar 1	solar1	X	X	0	18	0	20	2	0
24	Solar 2	solar2	X	X	0	19	0	21	2	1
25	Solar 3	solar3	X	X	0	10	0	22	2	2
26	Solar 4	solar4	X	X	0	15	0	23	2	3
27	Solar 5	solar5	X	X	0	9	0	24	2	4
28	Solar 6	solar6	X	X	0	14	0	25	X	X
29	Solar 7	solar7	X	X	0	16	0	26	X	X
30	Solar 8	solar8	X	X	0	8	0	27	X	X
31	Solar 9	solar9	X	X	0	13	0	28	2	5
32	Solar 10	solar10	X	X	0	12	0	29	X	X
33	Solar 11	solar11	X	X	0	11	0	30	X	X
34	Solar 12	solar12	X	X	0	17	0	31	X	X
35	Power Board	pb	X	X	0	5	0	0	2	6

III. VCP Commands

Command	Byte Code	Description
CS0	0x00	Turns off the specified component or subsystem (opens switch).
CS1	0x01	Turns on the specified component or subsystem (closes switch).
CST	0x02	Resets the specified component or subsystem (toggles switch).
CSF	0x03	Forces the switch of the specified component or subsystem to close.
CT	0x04	Control amount of current flowing in the Torquers. Three data bytes have input values for all three Torquers.
BE	0x05	This command is for CDH only. The Power Board reports state of the board to the beacon of the CDH Chip. Detailed functionality of beacon is explained in VIOLET-CDH-012, CDH Chip Design.
RTP	0x06	Sends a telemetry packet that contains the most recent SVIT-array data stored in the MCU memory.
RSS	0x07	Sends a packet that contains the most recent state of all the component or subsystem switches in the SVIT array.
RES	0x08	Sends a packet that contains the most recent values for the error bytes of the components in the SVIT array.

IV. Table of VCP Command Packets and Expected ACK Packets

Command {command byte in hex}	**Command Packet sent to Power MCU**	**Acknowledgement (ACK) Packet sent from Power MCU**
SWITCH OFF {0x00}	0xC0 0x01 0x00 0x09 0x00 0xSN 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x0A 0xER 0x00 0xSN 0xCR 0xCR 0xC0
SWITCH ON {0x01}	0xC0 0x01 0x00 0x09 0x01 0xSN 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x0A 0xER 0x01 0xSN 0xCR 0xCR 0xC0
SWITCH RESET {0x02}	0xC0 0x01 0x00 0x09 0x02 0xSN 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x0A 0xER 0x02 0xSN 0xCR 0xCR 0xC0
FORCE SWITCH ON {0x03}	0xC0 0x01 0x00 0x09 0x03 0xSN 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x0A 0xER 0x03 0xSN 0xCR 0xCR 0xC0
TORQUE CONTROL {0x04}	0xC0 0x01 0x00 0x0B 0x04 0xT1 0xT2 0xT3 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x0C 0xER 0x04 0xT1 0xT2 0xT3 0xCR 0xCR 0xC0
BEACON {0x05}	0xC0 0x01 0x00 0x08 0x05 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x09 0xER 0x05 0xCR 0xCR 0xC0
SEND TELEMETRY PACKET {0x06}	0xC0 0x01 0x00 0x08 0x06 0xCR 0xCR 0xC0	0xC0 0x01 0xLL 0xLL 0xER 0x06 [*SVIT array data as bytes*] 0xCR 0xCR 0xC0
REPORT SWITCH STATUS {0x07}	0xC0 0x01 0x00 0x08 0x07 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x19 0xER 0x07 [*switch values from SVIT array as one-bits, 4 bytes total*] 0xCR 0xCR 0xC0
REPORT ERROR STATUS {0x08}	0xC0 0x01 0x00 0x08 0x08 0xCR 0xCR 0xC0	0xC0 0x01 0x00 0x2D 0xER 0x08 [*error bytes of SVIT array's components, 36 bytes total*] 0xCR 0xCR 0xC0

Key for byte values that represent non-numerical quantities (not actual hexadecimal value):

Note: 0x01 byte after first 0xC0 is the address byte.

0xSN = switch number byte

0xCR = cyclic redundancy check byte

0xT1 = torquer 1 input byte

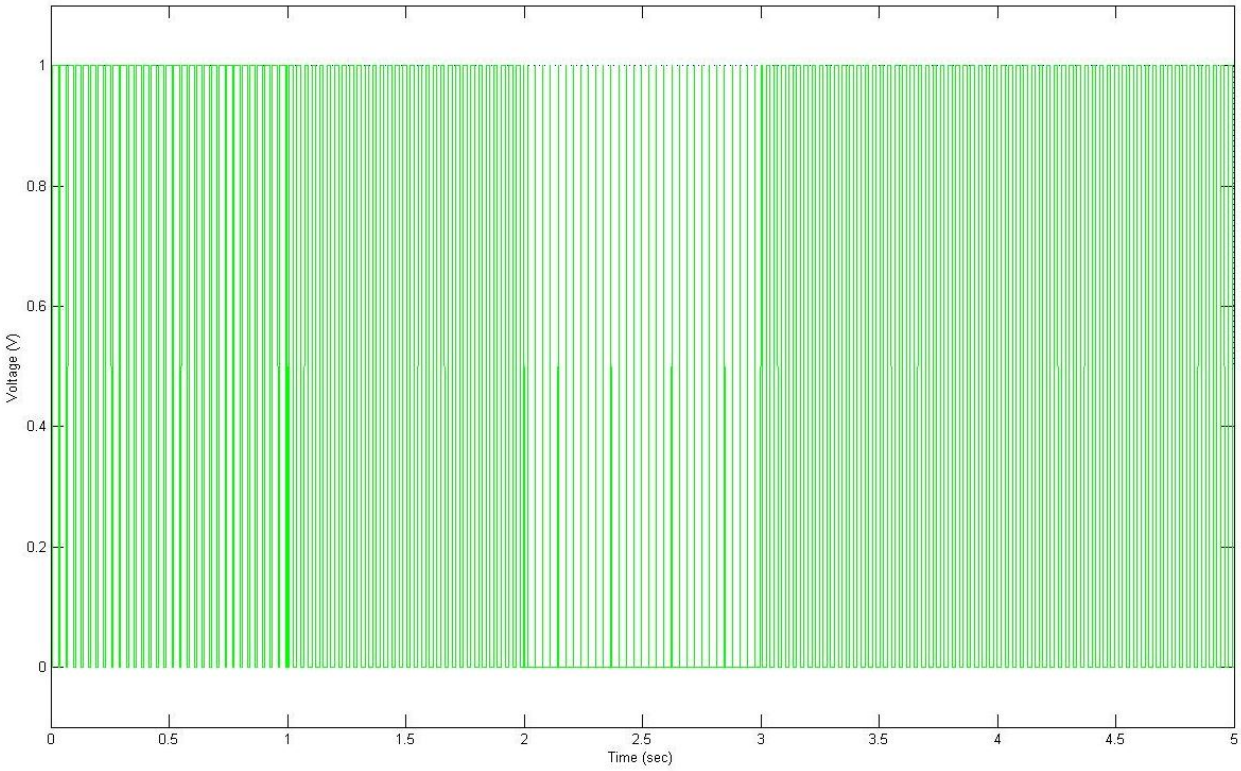
0xER = error byte

0xT2 = torquer 2 input byte

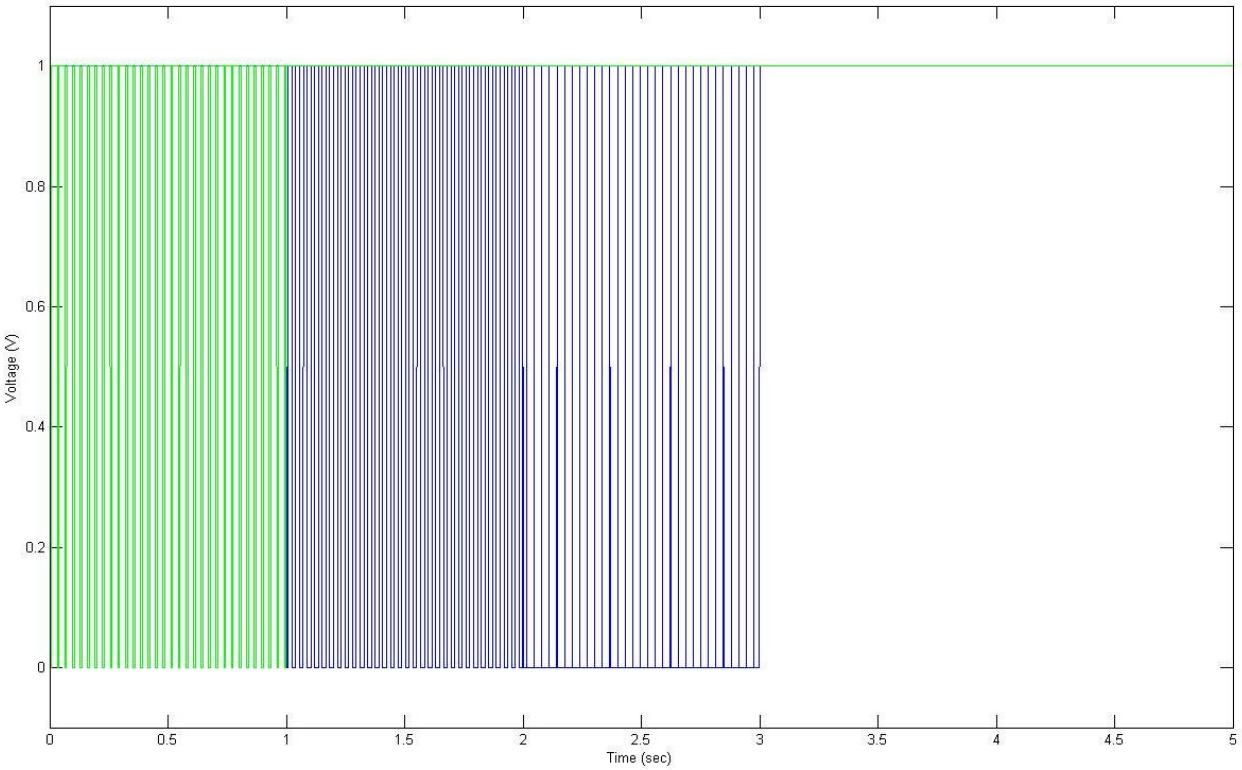
0xLL = length byte, two bytes total

0xT3 = torquer 3 input byte

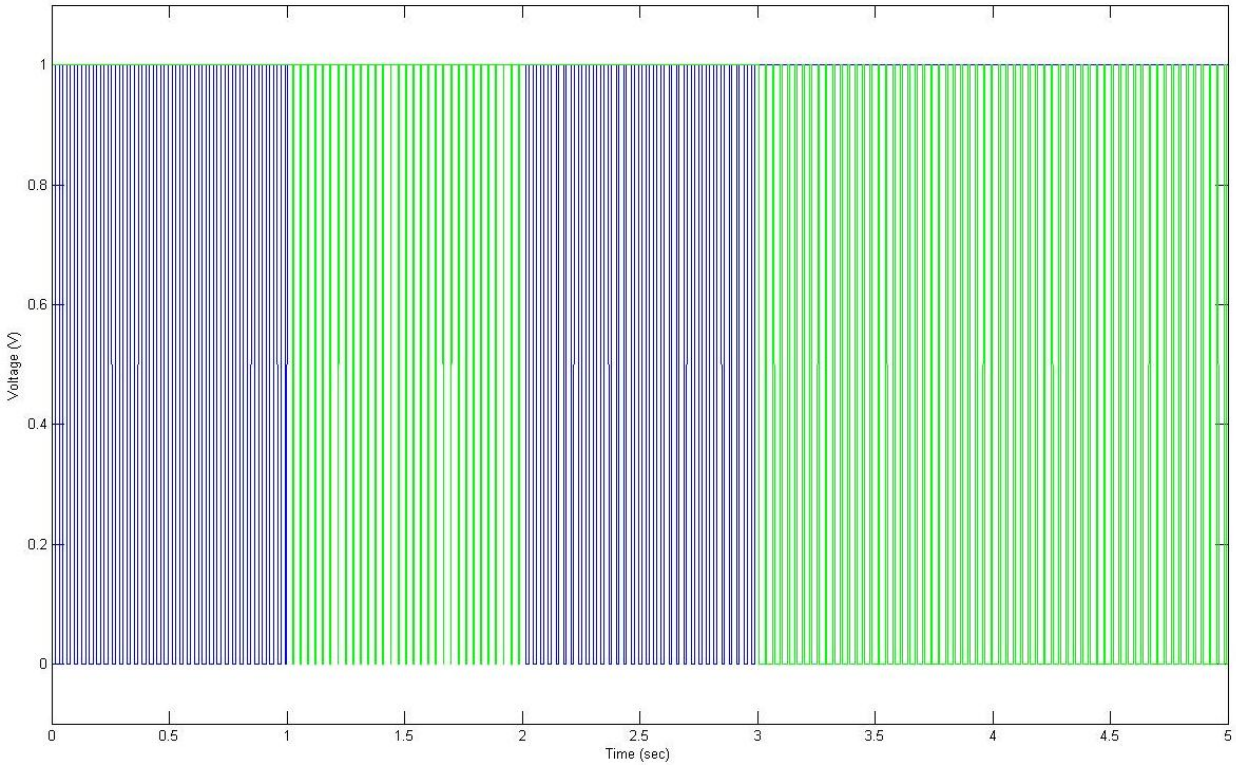
V. Simulated PWM Outputs for Magnetic Torquer Control



i. Input A and Input B of Torquer's H-bridge from simulation input {+32, +64, +127, +64}



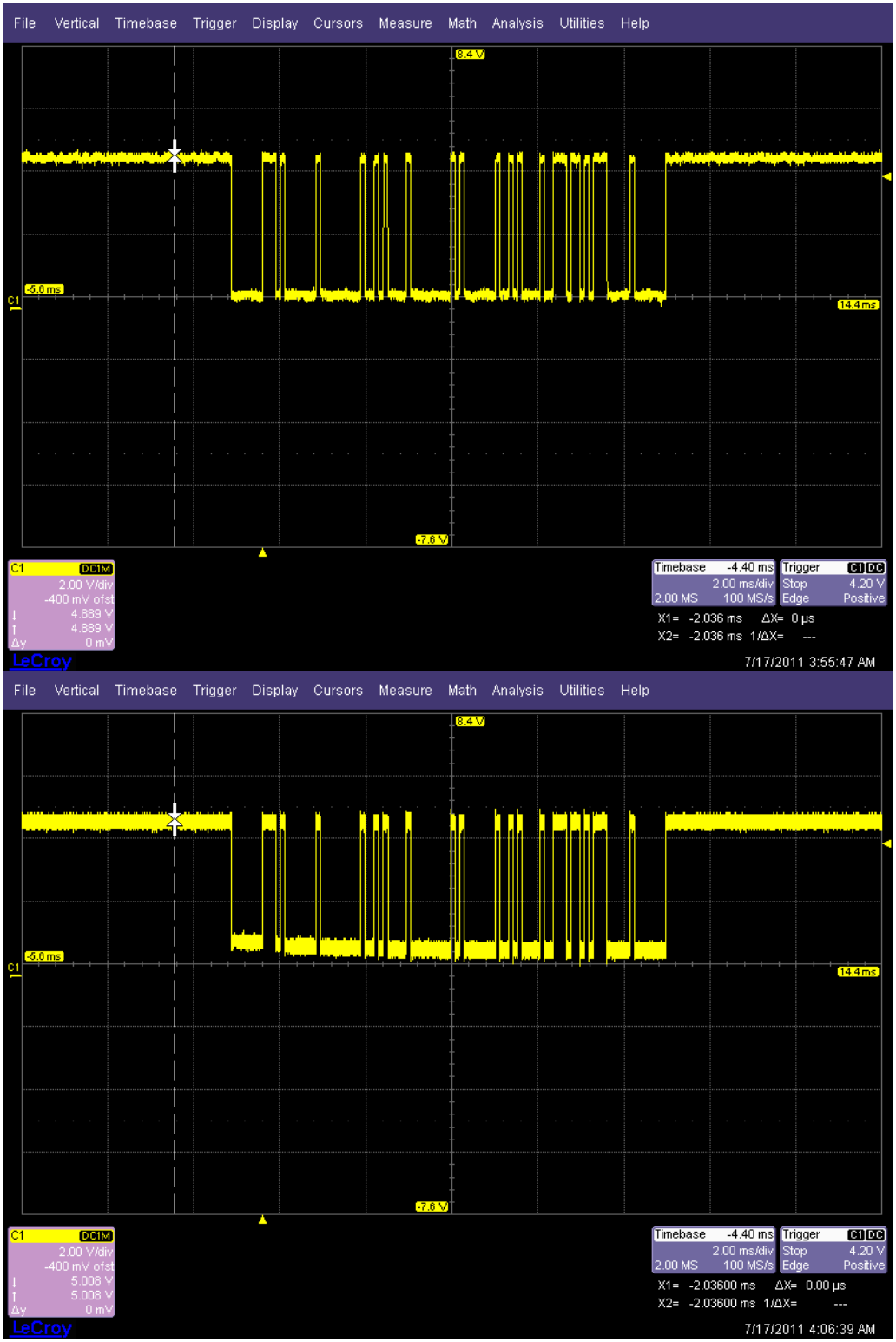
ii. Input A and Input B of Torquer's H-bridge from simulation input {+32, -64, -127, 0}



iii. Input A and Input B of Torquer's H-bridge from simulation input {-64, +16, -48, +96}

VI. ACK Packet In Response to Same Input

Oscilloscope images of the ACK packet transmitted on the USART0 TX pin after the MCU received a 0x01 0x0A command packet



VII. Matlab Code for Simulation of Outlier Problem

```
t = linspace(0, 6, 1001);
x = zeros(size(t));

x = (5/2)*cos(t*2*pi*(1/2)) + 5/2;
%plot(t, x, 'g');

rho = 0.2; % ratio of perturbated values
sigma = 2; % amplitude of perturbation
p = round(rho * size(t,2)); % number of pertubated values
% indexes of the perturbated values
sel = randperm(size(t,2));
sel = sel(1:p);
sel = sel(:);
% perturbation of the signal
f = x(:);
for i = 1 : length(sel)
    f(sel(i)) = f(sel(i)) + 2*(rand() - 1/2)*sigma;
    if (f(sel(i)) > 5)
        f(sel(i)) = 5;
    end
    if (f(sel(i)) < 0)
        f(sel(i)) = 0;
    end
end
hold on
plot(t, f, ':');

% f_sma = the output of SMA with window size 7
f_sma = f(:);
Wa = 7; % window size of last 7 points
for i = Wa : length(f_sma)
    sum = 0;
    for j = i-(Wa-1) : i
        sum = sum + f_sma(j);
    end
    f_sma(i) = sum / Wa;
end
plot(t, f_sma, 'g')

% f_smm = the output of SMM with window size of 7
f_smm = f(:);
k = 3; % half width
Wm = 2*k + 1; % width
for i = Wm : length(f_smm)
    sorted = f(i-(Wm-1):i);
    sorted = sort(sorted, 1, 'ascend');
    f_smm(i) = sorted(k+1);
end
plot(t, f_smm, 'k')
```

```
% f_exo = the output of SMA with local outliers excluded
f_exo = f(:);
We = 7;
for i = We : length(f_exo)
    sum = 0;
    sqsum = 0;
    for j = i-(We-1) : i
        sum = sum + f(j);
        sqsum = sqsum + (f(j) * f(j));
    end
    average = sum / We;
    stddev = sqrt((sqsum / We) - (average * average));
    exsum = 0;
    num_ex = 0;
    for j = i-(We-1) : i
        if (f(j) > average + (3*stddev))
            num_ex = num_ex + 1;
            continue;
        elseif (f(j) < average - (3*stddev))
            num_ex = num_ex + 1;
            continue;
        else
            exsum = exsum + f(j);
        end
    end
    f_exo(i) = exsum / (We - num_ex);
end
plot(t, f_exo, 'm')
xlabel('Time (arbitrary unit)')
ylabel('Current (A)')
```

VIII. Matlab Function for Simulating PWM-based Torquer Control

```
function pwm_sim(t1_part1, t1_part2, t1_part3, t1_part4);

t = linspace(0,5,20001);
i = 1;
time = t(i);

PC0 = zeros(size(t));
PG1 = zeros(size(t));

t1_counter = 0;
i = 1;
t1_pwm_on = t1_part1;
t1Input = t1_part1;
while (i <= length(t))
    if (t(i) < 1)
        t1Input = t1_part1;
    elseif (t(i) < 2)
        t1Input = t1_part2;
    elseif (t(i) < 3)
        t1Input = t1_part3;
    else
        t1Input = t1_part4;
    end

    t1_counter = t1_counter + 1;
    if (abs(t1_counter) > 127)
        t1_counter = 0;
    end

    if (t1_counter < abs(t1Input))
        if (sign(t1_pwm_on) ~= sign(t1Input))
            %drive_torq1(TORQUER_BRAKE);
            PC0(i) = 0;
            PG1(i) = 0;
        end
        if (sign(t1Input) > 0)
            %drive_torq1(TORQUER_SETA);
            PC0(i) = 1;
            PG1(i) = 0;
        else
            %drive_torq1(TORQUER_SETB);
            PC0(i) = 0;
            PG1(i) = 1;
        end
    else
        %drive_torq1(TORQUER_FREE);
        PC0(i) = 1;
        PG1(i) = 1;
    end
```

```
    t1_pwm_on = t1Input;
    i = i + 1;
end

%subplot(2,1,1), plot(t,PC0,'b')
%subplot(2,1,2), plot(t,PG1,'g')
plot(t,PC0,'b', t,PG1,'g')
axis([0 5 -0.1 1.1])
xlabel('Time (sec)')
ylabel('Voltage (V)')
```

IX. Power Microcontroller Code

power.h

```
/********************** INCLUDED LIBRARIES *************************/
#include <stdio.h>
#include <inttypes.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <avr/pgmspace.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "uart.h"
#include <stdint.h>
#include "vcp/vcplib.h"
#include "vcp/common.h"


/******************************** CONSTANTS ************************/
#define CLK          16000000L
#define CONST_TIMER0_COMPA  187
#define CONST_TIMER1_COMPA  249
#define TIME_CONST   64*CONST_COMPA*TIME_SENSE/CLK  //scalar by 64, 249 COMPA, 16 MHZ

#define TIME_SENSE        249
#define TIME_UPDATE_SOC   499
#define TIME_COMM         9
#define TIME_WRITE        9
#define TIME_SWITCH       49

#define MUX_NULL    3
#define MUXBIT_NULL 32
#define SVIT_SIZE   36
#define TEMP_SIZE   12
#define INDEX_BATT1 20
#define INDEX_BATT2 21

#define READ(U, N) ((U) >> (N) & 1u)
#define SET(U, N) ((void)((U) |= 1u << (N)))
#define CLR(U, N) ((void)((U) &= ~(1u << (N))))
#define FLIP(U, N) ((void)((U) ^= 1u << (N)))

#define CMD_BYTE_INDEX 2
#define COMP_BYTE_INDEX 3
#define TQ1_BYTE_INDEX 3
#define FEND_B_COUNT 2
#define ADDR_B_COUNT 1
#define CRC_B_COUNT 2
#define WRAPPER_BYTE_COUNT (FEND_B_COUNT + ADDR_B_COUNT + CRC_B_COUNT)
#define PWR0_BUF_SIZE 128
```

```
//#include "v2temp.h"
//#include <avr/pgmspace.h>
// used 22Kohms
#define starting_index    12
#define ending_index      990
#define len                     ending_index - starting_index  + 1
/*****************************************************************/


// SAMPLING STRUCTURE DEFINITION
typedef struct COMPONENT_struct
{
      unsigned char       S_INDEX;
      unsigned short      SAMPLE[8];
} COMPONENT_t;

// BOUNDS STRUCTURE DEFINITION
typedef struct BOUNDS_struct
{
      const int LowerBound;
      const int UpperBound;
      const int AbsLowerBound;
      const int AbsUpperBound;
} BOUNDS_t;

// SENSOR COMPONENT STRUCTURE DEFINITION
typedef struct SVIT_struct
{
      char name[8];             // the shorthand subsystem name as shown in the SVIT
spreadsheet
      char switchNum[3]; // the port pin (lowercase) tied to this subsystem's
switch
      char S;                         // the current state of the switch (1 = on, 0
= off)
      char error;               // the byte used to encode the detected errors for
this subsystem/component

      char   Vmux;        // which analogMUX/PORTF pin is this signal on?
      char   VmuxBit;     // which of the analogMUX's inputs is this signal?
      int          V;                 // most recently stored value for this
voltage
      int          ScaleFactorV;      // conversion facter for this voltage signal
      //BOUNDS_t   BoundsV;

      char   Imux;        // which analogMUX/PORTF pin is this signal on?
      char   ImuxBit;     // which of the analogMUX's inputs is this signal?
      int          I;                 // most recently stored value for this
current
      int          ScaleFactorI;      // conversion factor for this current signal
      //BOUNDS_t   BoundsI;

      char   Tmux;        // which analogMUX/PORTF pin is this signal on?
```

```
	char	TmuxBit;	// which of the analogMUX's inputs is this signal?
	int		T;			// most recently stored value for this
temperature
	//BOUNDS_t	BoundsT;
} SVIT_t;

// BATTERT STATE-OF-CHARGE STRUCTURE DEFINITION
typedef struct BATTERY_struct
{
	COMPONENT_t		METHOD0;
	COMPONENT_t		METHOD1;
	unsigned longSOC;
	COMPONENT_t		TEMP_SAMPLE;
	int				TEMPERATURE;
} BATTERY_t;


//unsigned int currentValue, voltageValue, tempValue;
COMPONENT_t I_SAMPLE[SVIT_SIZE]; // number of components and 8 samples each!
COMPONENT_t V_SAMPLE[SVIT_SIZE]; // number of components and 8 samples each!
COMPONENT_t T_SAMPLE[TEMP_SIZE]; // number of components and 8 samples each!
SVIT_t SVIT[SVIT_SIZE]; // two-dimensional SVIT array declaration
BATTERY_t BATTERY0, BATTERY1; // two batteries' SOC struct declarations


// POWER.C function declarations
void initialize(void);

// POWER_SAMPLE.C function declarations
void read_VIT(void);
unsigned int read_VIT_helper(char MUX_NUM, char MUX_SEL);
void selectMUX(char MUX_NUM, char MUX_SEL);
void storeValue(void);
int setValue(COMPONENT_t * c, unsigned int v, char ivtFlag);
void updateSOC(BATTERY_t * batt);
unsigned int filter(int sample, char SVITindex);
int getTemp(int voltage);

// POWER_CONTROL.C function declarations
void switchControl(int num, char onOffFlag);
void torqueControl(char torq1Input, char torq2Input, char torq3Input);
void drive_torq1(char state);
void drive_torq2(char state);
void drive_torq3(char state);

// POWER_COMM.C function declarations
void uart_read0_term(void);
void uart_write0_term(void);
void vcp_comm(void);
void vcp_write(void);
void putPacket(void);
void getPacket(void);
```

```
char parseMessage();
void writeACKMessage(void);
void sendTelemetry(void);
void sendSwitchStatus(void);
void sendErrors(void);
void sendBeacon(void);
void writeTELMessage(void);
void clearTxBuffer(void);

volatile int time1, time2, time3, time4, time5; //task scheduling timeout counters

vcp_ptrbuffer PWR0;     // pointer to vcp buffer used for RXing/TXing VCP packet
over UART

// USART0 ISR variables
//           RXC ISR variables
volatile char rx0_status;// return value of vcpptr_rx function
volatile unsigned int rx0_index;      // current string index
volatile char rx0_buff[20];     // input string
volatile char rx0_ready; // flag for receive done
volatile char rx0_char;  // current character
//           TX ISR variables
volatile char tx0_status;      // return value of vcpptr_tx function
volatile unsigned int tx_in, tx_out;  //
volatile unsigned int tx0_index;      // current string index
volatile char tx0_buff[PWR0_BUF_SIZE];      // output string
volatile char tx0_ready; // flag for transmit done
volatile char tx0_char;  // current character
char vcp_newACK;  // vcp_read-->vcp_write handshake
char vcp_newTEL;  // flag to indicate new Telemetry/Errors/SwitchStatus message is
ready to TX

char* telMessage;  // pointer to memory location of start of TEL packet message
int v;
char ack, command; // acknowledgement byte, command byte
int index_svit, index_temp;    // index for the SVIT array, index for the T_SAMPLE
array
int componentNum; // SVIT index of the component to be controlled by switch_control
in while(1) loop
char endSwitchState; // final state of switch being controlled
char t1value, t2value, t3value; // input values for torque_control of the 3
torquers
```

power.c

```
/*****************************************************
 Project : Power MCU Code
 Version : 3.0.1
 Date    : 4/1/2011
 Author  : Rajesh Atluri
 Company : Cornell Violet Satellite Project

 Chip type           : ATmega128
 Program type        : Application
 Clock frequency     : 16.000000 MHz
 Memory model        : -
 External SRAM size  : -
 Data Stack size     : -
 *****************************************************/

#include "power.h"


// UART file descriptor, putchar and getchar are in uart.c
FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);

 /* NOTE:    MUST MAKE SIGNIFICANT CHANGES TO SVIT ARRAY FOR ERROR HANDLING (e.g.
FLAG BITS) */
// MOD: THIS svit ARRAY HAS fc5'S CURRENT ON MUX1.0 AND gps1'S CURRENT ON MUX1.1
// MOD: switches of spec and star are "d4" and "d5" respectively. fog15's switch "c1"

// SVIT array, currently 36-elements of SVIT_t structs
SVIT_t SVIT[] = {
{"spec",     "d4", 0, 0x00, /*{0,256,0,256},*/  1, 16, 0, 1,   1,  6, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"star",     "d5", 0, 0x00, /*{0,256,0,256},*/  1, 23, 0, 1,   1,  7, 0, 1,   2,  7,
0},
{"fc5",      "a0", 0, 0x00, /*{0,256,0,256},*/  1, 19, 0, 1,   1,  0, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"fc3.3",    "a0", 0, 0x00, /*{0,256,0,256},*/  2, 13, 0, 1,   2, 14, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"gps1",     "a1", 0, 0x00, /*{0,256,0,256},*/  1, 21, 0, 1,   1,  1, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"gps2",     "a2", 0, 0x00, /*{0,256,0,256},*/  1, 26, 0, 1,   1,  2, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"ib",       "b5", 0, 0x00, /*{0,256,0,256},*/  1, 29, 0, 1,   1,  3, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"heat1",    "b6", 0, 0x00, /*{0,256,0,256},*/  1, 14, 0, 1,   1, 31, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"heat2",    "b7", 0, 0x00, /*{0,256,0,256},*/  1, 27, 0, 1,   1,  4, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"cmg",      "g2", 0, 0x00, /*{0,256,0,256},*/  1, 15, 0, 1,   1,  8, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"sunsen",   "c7", 0, 0x00, /*{0,256,0,256},*/  1, 25, 0, 1,   1,  9, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
```

```
{"radio1",   "c6", 0, 0x00, /*{0,256,0,256},*/  1, 24, 0, 1,   1, 10, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"radio2",   "c5", 0, 0x00, /*{0,256,0,256},*/  1, 17, 0, 1,   1, 11, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"maestro", "c4", 0, 0x00, /*{0,256,0,256},*/  1, 22, 0, 1,   1, 12, 0, 1,   2,  8,
0},
{"mag",      "c3", 0, 0x00, /*{0,256,0,256},*/  0,  4, 0, 1,   0,  6, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"fog15",    "c1", 0, 0x00, /*{0,256,0,256},*/  1, 18, 0, 1,   1, 13, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"fog5",     "c2", 0, 0x00, /*{0,256,0,256},*/  1, 18, 0, 1,   1, 13, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"torq1",    "",   0, 0x00, /*{0,256,0,256},*/  1, 28, 0, 1,   1,  5, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"torq2",    "",   0, 0x00, /*{0,256,0,256},*/  2, 15, 0, 1,   2, 16, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"torq3",    "",   0, 0x00, /*{0,256,0,256},*/  2, 17, 0, 1,   2, 18, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"batt1",    "",   0, 0x00, /*{0,256,0,256},*/  0,  1, 0, 1,   0,  2, 0, 1,   2,  9,
0}, // says 9, 10 for Bit on SVIT.xlsx; average them
{"batt2",    "",   0, 0x00, /*{0,256,0,256},*/  1, 20, 0, 1,   1, 30, 0, 1,   2, 11,
0}, // says 11, 12 for Bit on SVIT.xlsx; average them
{"solar",    "",     0, 0x00, /*{0,256,0,256},*/  0,  7, 0, 1,   0,  3, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar1",   "",   0, 0x00, /*{0,256,0,256},*/  0, 18, 0, 1,   0, 20, 0, 1,   2,  0,
0},
{"solar2",  "",   0, 0x00, /*{0,256,0,256},*/  0, 19, 0, 1,   0, 21, 0, 1,   2,  1,
0},
{"solar3",  "",   0, 0x00, /*{0,256,0,256},*/  0, 10, 0, 1,   0, 22, 0, 1,   2,  2,
0},
{"solar4",  "",   0, 0x00, /*{0,256,0,256},*/  0, 15, 0, 1,   0, 23, 0, 1,   2,  3,
0},
{"solar5",  "",      0, 0x00, /*{0,256,0,256},*/  0,  9, 0, 1,   0, 24, 0, 1,   2,
4, 0},
{"solar6",  "",   0, 0x00, /*{0,256,0,256},*/  0, 14, 0, 1,   0, 25, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar7",  "",   0, 0x00, /*{0,256,0,256},*/  0, 16, 0, 1,   0, 26, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar8",  "",   0, 0x00, /*{0,256,0,256},*/  0,  8, 0, 1,   0, 27, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar9",  "",   0, 0x00, /*{0,256,0,256},*/  0, 13, 0, 1,   0, 28, 0, 1,   2,  5,
0},
{"solar10", "",   0, 0x00, /*{0,256,0,256},*/  0, 12, 0, 1,   0, 29, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar11", "",   0, 0x00, /*{0,256,0,256},*/  0, 11, 0, 1,   0, 30, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"solar12", "",   0, 0x00, /*{0,256,0,256},*/  0, 17, 0, 1,   0, 31, 0, 1,
MUX_NULL, MUXBIT_NULL, 0},
{"pb",              "",   0, 0x00, /*{0,256,0,256},*/  0,  5, 0, 1,   0,  0, 0, 1,
2,  6, 0}                                 };


 // TIMER 0 ISR
```

```
ISR (TIMER0_COMP_vect)
{
	if (time1 > 0)	--time1;
}

// Timer 1 ISR
ISR (TIMER1_COMPA_vect)
{
	if (time2 > 0)  --time2;
	if (time3 > 0)  --time3;
	if (time4 > 0)  --time4;
	if (time5 > 0)	--time5;
}


// PROGRAM'S MAIN METHOD
int main(void)
{
	initialize();

	// TASK SCHEDULING INFINITE WHILE LOOP
	while(1)
	{
		if (time1 == 0)
		{
			time1 = TIME_SENSE;
			read_VIT();
			storeValue();
		}
		if (time2 == 0)
		{
			time2 = TIME_COMM;
			vcp_comm();

			//uart_read0_term(); FOR READING STRINGS TO HYPERTERM/TERMINAL
		}
		if (time3 == 0)
		{
			time3 = TIME_WRITE;
			//vcp_write();

			//uart_write0_term(); FOR WRITING STRINGS TO HYPERTERM/TERMINAL
			//uart_write1();
		}
		if (time4 == 0)
		{
			time4 = TIME_UPDATE_SOC;
			// UNDEVELOPED FUNCTIONS
			//updateSOC(&BATTERY0);
			//updateSOC(&BATTERY1);

			// de-package the data and call right function
```

```
                  //RAS_package(&PWR2, I_SAMPLE, V_SAMPLE, T_SAMPLE);
                  //UCSR1B |= (1 << UDRIE1);// start TX ISR
                  //t_index1 = 0;
            }
            if (time5 == 0)
            {
                  time5 = TIME_SWITCH;
                  if (endSwitchState == 2)
                  {
                        if (SVIT[componentNum].S)
                              switchControl(componentNum, 0);
                        endSwitchState = 1;
                  }
                  //else if (SVIT[componentNum].S != endSwitchState)
                        switchControl(componentNum, endSwitchState);
            }
            torqueControl(t1value, t2value, t3value);
      }
}

void initialize(void)
{
      cli();

      // initialize input pins and output port registers
      DDRA =      0xff;
      PORTA = 0x00;
      DDRB =      0xff;
      PORTB = 0x00;
      DDRC =      0xff;
      PORTC = 0x00;
      DDRD =      0b11111011;
      PORTD = 0b11000000;
      DDRE =      0b11111110;
      PORTE = 0b00000000;
      DDRF =0xf0;
      PORTF = 0x00;
      DDRG =0x1f;
      PORTG = 0x00;

      // set up uart
      uart_init();
      stdout = stdin = stderr = &uart_str;
      //fprintf(stdout, "%x%x%x", 0xC0, 0xFF, 0xC0);

      // TIMERS AND ADC initialization

      /********************** T I M E R S **********************/
      /*     Timer 0 initialization
      *      Clock source: External 16 MHz clock
      *      Mode: Normal top = CTC (Clear Timer on Compare)
      *      COMP value = 16
```

```
	*	Prescalar = 1
	*	Compare Match Interrupt Period = 1 us
	*/
	//TCNT0 = 0x00;
	//ASSR = (1 << AS0);		// External Clock on
	TIMSK = (1 << OCIE0);	// Enable CTC interrupt
	OCR0 = CONST_TIMER0_COMPA;	//set the compare reg. to 16 timer ticks
	TCCR0 = (1 << WGM01) | (1 << WGM00) | (1 << CS00); // start timer at Fcpu/1

	/*	Timer 1 initialization
	*	Clock source: External 16 MHz clock
	*	Mode: Normal top = CTC (Clear Timer on Compare)
	*	COMPA value = 249
	*	Prescalar = 1/64
	*	COMPA Match Interrupt Period = 1 ms approx.
	*/
	TIMSK |= (1 << OCIE1A);  // Enable CTC interrupt
	OCR1A = CONST_TIMER1_COMPA;	// set the compare reg. to 250 timer ticks
	TCCR1B = (1 << WGM12) | (1 << CS11) | (1 << CS10); // start timer at Fcpu/64
	/********************** T I M E R S **********************/

	/********************** ADC0-ADC2 initialization **********************/
	//ADMUX = (1 << REFS1) | (1 << REFS0) | (0 << ADLAR);	// ADC ref of
2.56V, Right justified values
	ADCSRA = ((1 << ADEN) | (1 << ADSC)) + 7;	// enable ADC, start first
conversion (takes longer than others),
                                                                    // and
make ADC clock 16E6/128 = 125 kHZ

	index_svit = 0;
	index_temp = 0;

	// ASSUME BATTERY0 and BATTERY1 are 100% charged initially
	BATTERY0.SOC = 0xffffffff;
	BATTERY1.SOC = 0xffffffff;

	// initalize VCP pointer buffer
	uint8 BUFFER0[PWR0_BUF_SIZE];
	vcpptr_init(&PWR0, BUFFER0, PWR0_BUF_SIZE);
	// set up circular buffer state variables
	tx_in = 0;
	tx_out = 0;
	rx0_index = 0;
	tx0_index = 0;
	rx0_ready = 0;	// no (RX buffer) input ready initially
	tx0_ready = 1;	// the (TX buffer) output and the transmitter are ready
initially
	vcp_newACK = 0;	// initialize ACK handshake to its RX-state value
	vcp_newTEL = 0; // initialize TEL handshake to its RX-state value

	// initialize the task timers/counters
	time1 = TIME_SENSE;
```

```
        time2 = TIME_COMM;
        time3 = TIME_WRITE;
        time4 = TIME_UPDATE_SOC;
        time5 = TIME_SWITCH;

        sei();// crank up the ISRs

        getPacket(); // initialize comm. service to receive packets
}
```

power_control.c

```
#include "power.h"

#define TORQUER_FREE 0
#define TORQUER_BRAKE 1
#define TORQUER_SETA 2
#define TORQUER_SETB 3

// NOTE TO SELF: try using "#define paste(front, back) front ## back" to reduce the
number
// of lines taken up by the switchControl function

/*	function that turns on/off the switch with index of num (orig. int, now char)
	in the SVIT struct array. The outer if statement is included to make sure the
	port pin is only changed if its current state differs from the final state
	indicated by onOffFlag. (turns on if onOffFlag = 1, turns off if onOffFlag =
0) */
void switchControl(int num, char onOffFlag)
{
	char switchFound = 1;
	//if (SVIT[num].S != onOffFlag) {
		switch (SVIT[num].switchNum[0]) {
			case 'a':
				switch (SVIT[num].switchNum[1]) {
					case '2':
						if (onOffFlag) SET(PORTA, PA2);
						else CLR(PORTA, PA2);
						break;
					case '1':
						if (onOffFlag) SET(PORTA, PA1);
						else CLR(PORTA, PA1);
						break;
					case '0':
						if (onOffFlag) SET(PORTA, PA0);
						else CLR(PORTA, PA0);
						break;
					default:
						switchFound = 0;
						break;
				}
				break;
			case 'b':
				switch (SVIT[num].switchNum[1]) {
					case '7':
						if (onOffFlag) SET(PORTB, PB7);
						else CLR(PORTB, PB7);
						break;
					case '6':
						if (onOffFlag) SET(PORTB, PB6);
						else CLR(PORTB, PB6);
						break;
```

```
                case '5':
                        if (onOffFlag) SET(PORTB, PB5);
                        else CLR(PORTB, PB5);
                        break;
                default:
                        switchFound = 0;
                        break;
        }
        break;
case 'c':
        switch (SVIT[num].switchNum[1]) {
                case '7':
                        if (onOffFlag) SET(PORTC, PC7);
                        else CLR(PORTC, PC7);
                        break;
                case '6':
                        if (onOffFlag) SET(PORTC, PC6);
                        else CLR(PORTC, PC6);
                        break;
                case '5':
                        if (onOffFlag) SET(PORTC, PC5);
                        else CLR(PORTC, PC5);
                        break;
                case '4':
                        if (onOffFlag) SET(PORTC, PC4);
                        else CLR(PORTC, PC4);
                        break;
                case '3':
                        if (onOffFlag) SET(PORTC, PC3);
                        else CLR(PORTC, PC3);
                        break;
                case '2':
                        if (onOffFlag) SET(PORTC, PC2);
                        else CLR(PORTC, PC2);
                        break;
                case '1':
                        if (onOffFlag) SET(PORTC, PC1);
                        else CLR(PORTC, PC1);
                default:
                        switchFound = 0;
                        break;
        }
        break;
case 'd':
        switch (SVIT[num].switchNum[1]) {
                case '5':
                        if (onOffFlag) SET(PORTD, PD5);
                        else CLR(PORTD, PD5);
                        break;
                case '4':
                        if (onOffFlag) SET(PORTD, PD4);
                        else CLR(PORTD, PD4);
```

```
                                        break;
                                default:
                                        switchFound = 0;
                                        break;
                                }
                                break;
                        case 'g':
                                switch (SVIT[num].switchNum[1]) {
                                case '2':
                                        if (onOffFlag) SET(PORTG, PG2);
                                        else CLR(PORTG, PG2);
                                        break;
                                default:
                                        switchFound = 0;
                                        break;
                                }
                                break;
                        default:
                                switchFound = 0;
                                break;
                }
                if (switchFound) {
                        SVIT[num].S = onOffFlag;
                }
        //}
}


// NOTE: Does torqueControl require a timer to be set into PWM mode or just a task
timer?
unsigned char t1_pwm_counter, t2_pwm_counter, t3_pwm_counter; // initialize to 0
(counts 0 to 0x7f)
unsigned char t1_pwm_on, t2_pwm_on, t3_pwm_on;    // initialize to +0

/* function that controls the magnetic torquer using software-counter-based PWM */
void torqueControl(char torq1Input, char torq2Input, char torq3Input)
{
        t1_pwm_counter++;
        t2_pwm_counter++;
        t3_pwm_counter++;
        if ((t1_pwm_counter & 0x7f) > 127) { t1_pwm_counter &= 0x80; }
        if ((t2_pwm_counter & 0x7f) > 127) { t2_pwm_counter &= 0x80; }
        if ((t3_pwm_counter & 0x7f) > 127) { t3_pwm_counter &= 0x80; }
        /* SEMAPHORE LOCK */
        if (t1_pwm_counter < torq1Input) {
                if ((t1_pwm_on & 0x80) != (torq1Input & 0x80))
                {
                        drive_torq1(TORQUER_BRAKE);
                        //delay_us(10);
                }
                if (torq1Input & 0x80)    drive_torq1(TORQUER_SETA);
                else   drive_torq1(TORQUER_SETB);
```

```
	} else drive_torq1(TORQUER_FREE);
	if (t2_pwm_counter < torq2Input) {
		if ((t2_pwm_on & 0x80) != (torq2Input & 0x80))
		{
			drive_torq2(TORQUER_BRAKE);
			//delay_us(10);
		}
		if (torq2Input & 0x80)	drive_torq2(TORQUER_SETA);
		else	drive_torq2(TORQUER_SETB);
	} else drive_torq2(TORQUER_FREE);
	if (t3_pwm_counter < torq3Input) {
		if ((t3_pwm_on & 0x80) != (torq3Input & 0x80))
		{
			drive_torq3(TORQUER_BRAKE);
			//delay_us(10);
		}
		if (torq3Input & 0x80)	drive_torq3(TORQUER_SETA);
		else	drive_torq3(TORQUER_SETB);
	} else drive_torq3(TORQUER_FREE);
	t1_pwm_on = torq1Input;
	t2_pwm_on = torq2Input;
	t3_pwm_on = torq3Input;
	/* SEMAPHORE UNLOCK */
}
/*	the following 3 functions set the output ports connected to
	each of the 3 torquer's H-bridge interface chips	*/
void drive_torq1(char state)
{
	if (state == TORQUER_FREE) {
		SET(PORTC, PC0);
		SET(PORTG, PG1);
	} else if (state == TORQUER_BRAKE) {
		CLR(PORTC, PC0);
		CLR(PORTG, PG1);
	} else if (state == TORQUER_SETA) {
		SET(PORTC, PC0);
		CLR(PORTG, PG1);
	} else if (state == TORQUER_SETB) {
		CLR(PORTC, PC0);
		SET(PORTG, PG1);
	}
}
void drive_torq2(char state)
{
	if (state == TORQUER_FREE) {
		SET(PORTG, PG0);
		SET(PORTD, PD0);
	} else if (state == TORQUER_BRAKE) {
		CLR(PORTG, PG0);
		CLR(PORTD, PD0);
	} else if (state == TORQUER_SETA) {
		SET(PORTG, PG0);
```

```
            CLR(PORTD, PD0);
      } else if (state == TORQUER_SETB) {
            CLR(PORTG, PG0);
            SET(PORTD, PD0);
      }
}
void drive_torq3(char state)
{
      if (state == TORQUER_FREE) {
            SET(PORTG, PG3);
            SET(PORTG, PG4);
      } else if (state == TORQUER_BRAKE) {
            CLR(PORTG, PG3);
            CLR(PORTG, PG4);
      } else if (state == TORQUER_SETA) {
            SET(PORTG, PG3);
            CLR(PORTG, PG4);
      } else if (state == TORQUER_SETB) {
            CLR(PORTG, PG3);
            SET(PORTG, PG4);
      }
}
```

power_comm.c

```
#include "power.h"

char newPacketLength;
int telSentCount;

// UART 8-bit-character-ready receive ISR
ISR(USART0_RX_vect)
{
	rx0_char = UDR0; // get an 8-bit char
	rx0_status = vcpptr_rx(&PWR0, rx0_char);
	rx0_buff[rx0_index++] = rx0_char;
	if ((rx0_status & VCP_TERM) == VCP_TERM)
	{
		UCSR0B ^= (1 << RXCIE0); // clear RX enable
		rx0_ready = 1;	// receive done
	}
}

// UART 8-bit-character-ready transmit ISR
ISR(USART0_UDRE_vect)
{
	if (tx_in == tx_out)
		UCSR0B &= ~(1 << UDRIE0); // disable TX interrupt
	else
	{
		//loop_until_bit_is_set(UCSR0A, UDRE0);
		UDR0 = tx0_char = (&PWR0)->message[tx_out];
		tx_out++;
		if (tx_out == PWR0_BUF_SIZE)
			tx_out = 0;
		/*if (tx_out >= newPacketLength)
		{
			UCSR0B ^= (1 << UDRIE0); // clear TX enable
			tx0_ready = 1;	// transmit done
		}*/
	}
}

/*	Writes the input byte into the UART using a circular buffer approach
	and activates the UART transmit ISR	*/
int writeByte(char c)
{
	char i = tx_in;
	i++;
	if (i == PWR0_BUF_SIZE)
		i = 0;
	(&PWR0)->message[tx_in] = c;

	while (i == tx_out);	// until at least 1 byte free
		// tx_out modified by interrupt!
```

```
	tx_in = i;
	UCSR0B |= (1 << UDRIE0);
	//UCSR0A |= (1 << UDRE0);
	loop_until_bit_is_set(UCSR0A, UDRE0);

	return 0;
}

/*	Writes the message in tx0_buff to the message field of the vcpptr_buffer
	and sends the message one byte at a time	*/
void putPacket(void)
{
	tx0_ready = 0; // mark tranmitter as busy
	tx0_index = 0;
	vcpptr_clear(&PWR0);
	(&PWR0)->address = VCP_POWER;  // packets from Power MCU have VCP_POWER
address (see vcplib.h)

	while (tx0_index < newPacketLength - WRAPPER_BYTE_COUNT)
	{
		tx0_status = vcpptr_tx(&PWR0, tx0_buff[tx0_index++], (&PWR0)-
>flags);
	}
	tx0_status = vcpptr_tx(&PWR0, tx0_buff[tx0_index++], VCP_TERM);

	int tx_out_init = tx_out;
	while (tx_out - tx_out_init < newPacketLength)
	{
		writeByte((&PWR0)->message[tx_out]);
	}

	UCSR0B ^= (1 << UDRIE0); // clear TX enable
	tx0_ready = 1;	// transmit done
	clearTxBuffer();

}

/* writes an acknowledgement message into the tx0_buff buffer */
void writeACKMessage(void)
{
	newPacketLength = (char)((&PWR0)->index) + WRAPPER_BYTE_COUNT + 1;
	tx0_buff[0] = (char)(newPacketLength >> 8);
	tx0_buff[1] = (char)newPacketLength;
	tx0_buff[2] = ack;
	int j;
	for (j = 0; j < (&PWR0)->index - 2; j++)
	{
		tx0_buff[j+3] = ((&PWR0)->message[j+2]);
	}
}

/* writes the message in the telMessage buffer into the tx0_buff buffer */
```

```
void writeTELMessage(void)
{

	if (sizeof(&telMessage) + WRAPPER_BYTE_COUNT > PWR0_BUF_SIZE)
	{

	}
	else
	{
		char dataByteCount = sizeof(&telMessage) + WRAPPER_BYTE_COUNT;
		tx0_buff[0] = (char)(dataByteCount >> 8);
		tx0_buff[1] = (char)dataByteCount;
		int j;
		for (j = 0; j < (dataByteCount - WRAPPER_BYTE_COUNT); j++)
		{
			tx0_buff[j+2] = telMessage[j];
		}
		telSentCount = sizeof(&telMessage);
	}
	newPacketLength = telSentCount + WRAPPER_BYTE_COUNT;
}

// TOP-LEVEL FUNCTION OF THE VCP-PACKET-BASED COMMUNICATION OVER UART
void vcp_comm(void)
{
	if (rx0_ready)	// is the RX buffer string ready to read?
	{
		// extract message bytes from rx0_buff
		ack = parseMessage();
		getPacket(); // set up procedure to get the next string input and read
it using RX ISR
		vcp_newACK = 1; // signal that a new command has been received
	}
	else if (tx0_ready && vcp_newACK)	// see if the last transmit is done and
if there is a new ACK message to TX
	{
		writeACKMessage();
		//vcp_package((&PWR0)->message, &((&PWR0)->size), (&PWR0)->address,
(uint8ptr)tx0_buff, sizeof(tx0_buff));
		putPacket(); // send ACK packet using transmit ISR

		vcp_newACK = 0; // clear the handshake from the above state
		vcpptr_clear(&PWR0);
	}
	else if (tx0_ready && vcp_newTEL && !vcp_newACK)  // see if the last
transmit is done and if there is a new TEL message to TX
	{
		telSentCount = 0;
		do {
			writeTELMessage();
		} while (telSentCount < sizeof(&telMessage));
		putPacket();
```

```
			vcp_newTEL = 0;
			vcpptr_clear(&PWR0);
		}
		/*else if (tx0_ready == rx0_ready)
		{
			tx0_ready = 0;
			rx0_ready = 0;
			UCSR0B |= (1 << RXCIE0); // turn on receive ISR
		}*/
}

/* adjusts variables to enable RXing of VCP packet and enables RX ISR */
void getPacket(void)
{
		rx0_ready = 0; // mark RX buffer as not ready
		rx0_index = 0; // reset index
		UCSR0B |= (1 << RXCIE); // turn on receive ISR
}

/* breaks down PWR0's message buffer into certain fields and executes the sent
command */
char parseMessage(void)
{
		char cmdStatus = 0;
		command = (&PWR0)->message[CMD_BYTE_INDEX];
		if (command <= 0x03)
			componentNum = (&PWR0)->message[COMP_BYTE_INDEX]; //componentNum =
(int)msg[COMP_BYTE_INDEX];//componentNum = ((&PWR0)->message[COMP_BYTE_INDEX]);
		else
			componentNum = 999;
		//char t1value, t2value, t3value;
		switch(command)
		{
			case 0x00:	//"CS0" - SWITCH OFF
				endSwitchState = 0;
				cmdStatus = 0x00;
				break;
			case 0x01:	//"CS1" - SWITCH ON
				endSwitchState = 1;
				cmdStatus = 0x00;
				break;
			case 0x02:	//"CST" - SWITCH RESET
				endSwitchState = 2;
				cmdStatus = 0x00;
				break;
			case 0x03:	//"CSF" - FORCE SWITCH ON
				endSwitchState = 1;
				cmdStatus = 0x00;
				break;
			case 0x04:	//"CT" - TORQUE CONTROL
				t1value = (&PWR0)->message[TQ1_BYTE_INDEX];
				t2value = (&PWR0)->message[TQ1_BYTE_INDEX + 1];
```

```
                t3value = (&PWR0)->message[TQ1_BYTE_INDEX + 2];
                //torqueControl(t1value, t2value, t3value);
                break;
            case 0x05:   //"BE" - BEACON

                break;
            case 0x06:   //"RTP" - SEND TELEMETRY PACKET
                sendTelemetry();
                break;
            case 0x07:   //"RSS" - REPORT SWITCH STATUS
                sendSwitchStatus();
                break;
            case 0x08:   //"RES" - REPORT ERROR STATUS
                sendErrors();
                break;
            default:
                cmdStatus = 0xff;
                break;
        }
        return cmdStatus;
}


// formats and writes a telemetry packet with data from the SVIT struct into the
telMessage buffer
void sendTelemetry(void)
{
        //unsigned char vcpTXflags;
        int i;
        for (i = 0; i < SVIT_SIZE; i++)
        {
            sprintf(telMessage, "%s,%s,%x,%x\t%d\t%d\t%d\n\r",
                SVIT[i].name, SVIT[i].switchNum, SVIT[i].S, SVIT[i].error,
SVIT[i].V, SVIT[i].I, SVIT[i].T);
        }
        vcp_newTEL = 1;
}

// formats and writes the status of switches into the telMessage buffer
void sendSwitchStatus(void)
{
        int i;
        for (i = 0; i < SVIT_SIZE; i++)
        {
            sprintf(telMessage, "%x", SVIT[i].S);
            if (i < SVIT_SIZE - 1)
                sprintf(telMessage, ",");
        }

        vcp_newTEL = 1;
}
```

```
// formats and writes the components' error bytes into the telMessage buffer
void sendErrors(void)
{
	int i;
	for (i = 0; i < SVIT_SIZE; i++)
	{
		sprintf(telMessage, "%x", SVIT[i].error);
		if (i < SVIT_SIZE - 1)
			sprintf(telMessage, ",");
	}

	vcp_newTEL = 1;
}

// AS OF NOW THE BEACON FUNCTION HAS YET TO BE DEFINED
void sendBeacon(void)
{

}

// clears the tx0_buff buffer so it can get a new message into it
void clearTxBuffer(void)
{
	int i;
	for (i = 0; i <= newPacketLength; i++)
	{
		tx0_buff[i] = 0;
	}
}
```

power_sample.c

```
#include "power.h"

const int TempScaleFactor =    1;    // conversion factor for all temperature
signals

// variables that store the digital value output by the MCU's A-to-D Converter (ADC)
unsigned int currentValue, voltageValue, tempValue;
unsigned int batt1CurrentValue, batt1VoltageValue, batt1TempValue;
unsigned int batt2CurrentValue, batt2VoltageValue, batt2TempValue;
// variables that equal the ADC output value multiplied by the appropriate sensor
scaling factor
unsigned int currentScaledValue, voltageScaledValue, tempScaledValue;
unsigned int batt1CurrentScaled, batt1VoltageScaled, batt1TempScaled;
unsigned int batt2CurrentScaled, batt2VoltageScaled, batt2TempScaled;

/*    TOP LEVEL FUNCTION THAT EXECUTES AN ANALOG-TO-DIGITAL CONVERSION OF THE
      MEASURED OUTPUT OF THE SESNSORS OF THE CURRENTLY INDEXED SVIT COMPONENT
      AND OF THE BATTERIES      */
void read_VIT(void)
{
      /*
      *     We will use 10-bit precision, right justified (ADLAR = 0)
      *     PINF0/ADC0 = MUX with output VOLT_SEN_OUT
      *     PINF1/ADC1 = MUX with output CUR_SEN_OUT
      *     PINF2/ADC2 = MUX with output SEN_THERM_OUT
      */
      if (index_svit++ >= SVIT_SIZE)
            index_svit = 0;
      if (index_svit == INDEX_BATT1)
            index_svit++;
      if (index_svit == INDEX_BATT2)
            index_svit++;

      currentValue = read_VIT_helper(SVIT[index_svit].Imux,
SVIT[index_svit].ImuxBit);
      voltageValue = read_VIT_helper(SVIT[index_svit].Vmux,
SVIT[index_svit].VmuxBit);
      if (SVIT[index_svit].Tmux != MUX_NULL)
            tempValue = read_VIT_helper(SVIT[index_svit].Tmux,
SVIT[index_svit].TmuxBit);

      batt1CurrentValue = read_VIT_helper(SVIT[INDEX_BATT1].Imux,
SVIT[INDEX_BATT1].ImuxBit);
      batt1VoltageValue = read_VIT_helper(SVIT[INDEX_BATT1].Vmux,
SVIT[INDEX_BATT1].VmuxBit);
      batt1TempValue = read_VIT_helper(SVIT[INDEX_BATT1].Tmux,
SVIT[INDEX_BATT1].TmuxBit);

      batt2CurrentValue = read_VIT_helper(SVIT[INDEX_BATT2].Imux,
```

```
SVIT[INDEX_BATT2].ImuxBit);
	batt2VoltageValue = read_VIT_helper(SVIT[INDEX_BATT2].Vmux,
SVIT[INDEX_BATT2].VmuxBit);
	batt2TempValue = read_VIT_helper(SVIT[INDEX_BATT2].Tmux,
SVIT[INDEX_BATT2].TmuxBit);
}

// MUX_NUM is from {0, 1, 2} for the analog multiplexor that is being sampled from
// MUX_SEL is 0-31 for the input of the MUX that is selected and sampled

// sets the ADC registers of the MCU and actually performs the analog-to-digital
conversion
unsigned int read_VIT_helper(char MUX_NUM, char MUX_SEL)
{
	unsigned int temp;
	selectMUX(MUX_NUM, MUX_SEL);
	ADMUX = (1 << REFS0) | MUX_NUM;// sets voltage ref. to AVCC with external
capacitor at AREF
                                                        // and selects one of the
3 ADC sampling channels
	ADCSRA |= (1 << ADSC);   // start conversion
	while (ADCSRA & (1 << ADSC));   // blocking while waiting for end of
conversion
	temp = ADCL;// we must read ADCL first according to spec sheet
	temp |= (ADCH << 8);     // reading ADCH will refresh ADC register
	return temp;
}

// drives the MCU port pins, connected to the select inputs of the activated MUX,
// with MUX_SEL depending on MUX_NUM, which indicates the activated MUX
/* #define MUX_SELECT(bit, port, pbit) ((bit) ? (port |= (1 << pbit)) : (port &= ~(1
<< pbit))) OLD */
void selectMUX(char MUX_NUM, char MUX_SEL)
{
	if (MUX_NUM == 0)
	{
		//PORTA &= 0xf8;
		if(READ(MUX_SEL, 0)) SET(PORTA, PA3);
		else CLR(PORTA, PA3);
		if(READ(MUX_SEL, 1)) SET(PORTA, PA4);
		else CLR(PORTA, PA4);
		if(READ(MUX_SEL, 2)) SET(PORTA, PA5);
		else CLR(PORTA, PA5);
		if(READ(MUX_SEL, 3)) SET(PORTA, PA6);
		else CLR(PORTA, PA6);
		if(READ(MUX_SEL, 4)) SET(PORTA, PA7);
		else CLR(PORTA, PA7);
	}
	else if (MUX_NUM == 1)
	{
		//PORTE &= 0xf8;
		if(READ(MUX_SEL, 0)) SET(PORTE, PE7);
```

```
            else CLR(PORTE, PE7);
            if(READ(MUX_SEL, 1)) SET(PORTE, PE6);
            else CLR(PORTE, PE6);
            if(READ(MUX_SEL, 2)) SET(PORTE, PE5);
            else CLR(PORTE, PE5);
            if(READ(MUX_SEL, 3)) SET(PORTE, PE4);
            else CLR(PORTE, PE4);
            if(READ(MUX_SEL, 4)) SET(PORTE, PE3);
            else CLR(PORTE, PE3);
      }
      else if (MUX_NUM == 2)
      {
            //PORTB &= 0x1f;
            if(READ(MUX_SEL, 0)) SET(PORTB, PB4);
            else CLR(PORTB, PB4);
            if(READ(MUX_SEL, 1)) SET(PORTB, PB3);
            else CLR(PORTB, PB3);
            if(READ(MUX_SEL, 2)) SET(PORTB, PB2);
            else CLR(PORTB, PB2);
            if(READ(MUX_SEL, 3)) SET(PORTB, PB1);
            else CLR(PORTB, PB1);
            if(READ(MUX_SEL, 4)) SET(PORTB, PB0);
            else CLR(PORTB, PB0);
      }
}

/*    TOP LEVEL FUNCTION THAT SCALES THE ADC CONVERSION'S OUTPUT AND APPLIES
      A FILTER TO THIS SCALED VALUE BEFORE STORING IT IN THE APPROPRIATE
      ELEMENT OF THE SVIT TABLE */
void storeValue()
{
      currentScaledValue = currentValue * SVIT[index_svit].ScaleFactorI;
      SVIT[index_svit].I = setValue(&I_SAMPLE[index_svit], currentScaledValue,
0);
      voltageScaledValue = voltageValue * SVIT[index_svit].ScaleFactorV;
      SVIT[index_svit].V = setValue(&V_SAMPLE[index_svit], voltageScaledValue,
1);
      if (SVIT[index_svit].Tmux != MUX_NULL)
      {
            tempScaledValue = tempValue * TempScaleFactor;
            SVIT[index_svit].T = setValue(&T_SAMPLE[index_temp++],
tempScaledValue, 2);
            if (index_temp >= TEMP_SIZE) index_temp = 0;
      }

      batt1CurrentScaled = batt1CurrentValue * SVIT[INDEX_BATT1].ScaleFactorI;
      SVIT[INDEX_BATT1].I = setValue(&I_SAMPLE[INDEX_BATT1],
batt1CurrentScaled, 0);
      batt1VoltageScaled = batt1VoltageValue * SVIT[INDEX_BATT1].ScaleFactorV;
      SVIT[INDEX_BATT1].V = setValue(&V_SAMPLE[INDEX_BATT1],
batt1VoltageScaled, 1);
      batt1TempScaled = batt1TempValue * TempScaleFactor;
```

```
	SVIT[INDEX_BATT1].T = setValue(&T_SAMPLE[INDEX_BATT1], batt1TempScaled,
2);

	batt2CurrentScaled = batt2CurrentValue * SVIT[INDEX_BATT2].ScaleFactorI;
	SVIT[INDEX_BATT2].I = setValue(&I_SAMPLE[INDEX_BATT2],
batt2CurrentScaled, 0);
	batt2VoltageScaled = batt2VoltageValue * SVIT[INDEX_BATT2].ScaleFactorV;
	SVIT[INDEX_BATT2].V = setValue(&V_SAMPLE[INDEX_BATT2],
batt1VoltageScaled, 1);
	batt2TempScaled = batt2TempValue * TempScaleFactor;
	SVIT[INDEX_BATT2].T = setValue(&T_SAMPLE[INDEX_BATT2], batt2TempScaled,
2);
}

/* NOTE: SVIT array's I/V/T are not valid SMA values until after 8 executions of
setValue() */

// returns the newly computed SMA of the most recent 8 samples (including the one
from the previous read_VIT() call).
// Assumes that SVIT array's stored value is the last computed SMA.
// Updates 1 of the 8 samples stored in the SAMPLES array of the COMPONENT_t struct.
// ivtFlag: 0 = current, 1 = voltage, 2 = temperature
int setValue(COMPONENT_t * c, unsigned int v, char ivtFlag)
{
	int newStoredValue, oldStoredValue;
	if (ivtFlag == 0) oldStoredValue = SVIT[index_svit].I;
	else if (ivtFlag == 1) oldStoredValue = SVIT[index_svit].V;
	else if (ivtFlag == 2) oldStoredValue = SVIT[index_svit].T;
	newStoredValue = oldStoredValue + ((v - (c -> SAMPLE[c -> S_INDEX])) >>
3);
	//c -> SAMPLE[c -> S_INDEX] = v; // the dropped sample is replaced by the most
recently taken sample
	//c -> S_INDEX++;
	if (c -> S_INDEX > 7)
		c -> S_INDEX = 0;
	return newStoredValue;
}

/* VALUE_RET function carried over from old power.c */
unsigned int retValue(COMPONENT_t * c)
{
	unsigned int ret, i;
	ret = 0;
	for (i = 0; i < 8; i++)
	{
		ret += c -> SAMPLE[i];
	}
	return (ret >> 3);
}

// updates the batter SOC variables in the SVIT struct
// if SOC(Vcell) is a monotonically decreasing function, then
```

```
//     x^n/(k-x^n) can model the curve well
void updateSOC(BATTERY_t * batt)
{
	unsigned int x = retValue(&batt -> METHOD0);
	unsigned int y = retValue(&batt -> METHOD1);
	batt -> TEMPERATURE = getTemp( retValue(&batt -> TEMP_SAMPLE) );
	if (x > y)
		batt -> SOC = (batt -> SOC) + x;
	else
		batt -> SOC = (batt -> SOC) + y;
}


// voltage-to-temperature conversion array stored in Flash memory
prog_int16_t ADC2TEMP[len] = {
12498  ,
12269  ,
12056  ,

…
…
…

-5278  ,
-5332  ,
-5388
};

// returns the temperature value corresponding to the voltage input from ADC2TEMP
array
int getTemp(int voltage)
{
	if (voltage < starting_index || voltage > ending_index)
	{
		return -1;
	}
	return pgm_read_word(((uint16_t)(ADC2TEMP)) + (voltage -
starting_index)*2);
}
```

REFERENCES

Atmel Corporation. “ATmega128/ATmega128L.” Datasheet available at www.atmel.com.

Violet Satellite Project, Cornell University. “Power Board Design.” Document ID: VIOLET-PWR-003.

Violet Satellite Project, Cornell University. “Power Board Schematic.” Revision 1, December 2010.

Violet Satellite Project, Cornell University. “Power Board Schematic.” Revision 2, May 2011.

Violet Satellite Project, Cornell University. “Power Requirements.” Document ID: VIOLET-PWR-001.

Violet Satellite Project, Cornell University. “PWR Board VIOLET-P-PWR-200.” Drawing No.: VIOLET-PD-PWR-200.