Satellite Firmware

Cube Satellite On Board Computer

Members:
Brendan Bruner bbruner@ualberta.ca
Divyank Katira katira@ualberta.ca
Jeffrey Ryan jfryan@ualberta.ca
Oleg Oleynikov oleyniko@ualberta.ca

Summary:
The design and implementation of firmware for a cube satellite's on board computer.

1. Table of Contents

2. Acronyms

ADCS = Attitude Determination and Control System
CDFSM = Command Driven Finite State Machine
CDH = Command and Data Handling
COMMs = Communications
CSP = CubeSat Space Protocol
DFGM = Digital Fluxgate Magnetometer
EPS = Electronic Power Supply
FSM = Finite State Machine
GB = Gigabyte
GPS = Global Positioning Unit
HK = Housekeeping
IC = Integrated Circuit
IO = Input / Output
MCU = Microcontroller Unit
MnLP = Multi - Needle Langmuir Probe
OBC = On Board Computer
OS = Operating System
PCB = Printed Circuit Board
SU = Science Unit
UTC = Coordinated Universal Time
WDT = Watch Dog Timer
WOD = Whole Orbit Data

3. Abstract

AlbertaSat's cube satellite collects, maintains, and communicates data from three subsystems. These subsystems are the communications board (COMMs), electronic power supply (EPS), and the attitude control and determination system (ADCS). To do this, a command and data handling (CDH) system was designed; this is often called the on board computer (OBC). The OBC uses a state machine to walk through different phases of operation: power on (system turns on and follows start up procedures), power safe (restricts certain functionality to conserve power), detumbling (physically adjusts the satellite's rotation and orientation), and science (prioritizes the collection of scientific data). FreeRTOS is used to schedule tasks that perform the required behaviour. This includes collecting housekeeping data from the subsystems, logging collected data in non volatile memory, decoding commands sent to the satellite, and downlinking collected data to a ground station. In order to protect expensive hardware, mock up devices are used to imitate the satellites subsystems. This allows development and testing to continue without damaging the end-product hardware.

4. Functional Requirements

4.1. File System

An external, low storage (2 GB), SD card is used as the satellite's non volatile memory resource. A FAT file system is used to write and read from the SD card. A low storage card is used because it has a higher tolerance to the radiation present in space.

4.2. House Keeping

Every 60 seconds, housekeeping data is collected from the subsystems and saved to nonvolatile memory. This data is transmitted to a ground station when possible (i.e. a ground station is nearby and the satellite is not in its power safe state). Housekeeping data includes:

- subsystem diagnostic information
- state transistions

4.3. Subsystem Communication

The EPS mockup exists but is unused. The OBC does not have the code written which would enable it to talk to the EPS mockup running on a separate board. Instead, the EPS is simulated directly on the OBC. For example, their is a function to ask the EPS what the voltage is across its batteries. Instead of doing IO with the mock EPS board, this function just returns a number in large array of possible battery voltages. Below is a summary of the communication protocols used with each subsystem:

- EPS: I2C
- COMMs: UART
- ADCS: I2C
- SD card: SPI

The COMM board (the real, expensive COMM board, not the mock up) is communicated with over I2C, but the mockup COMM board uses UART. This decision was made because implementing slave mode on the cheap development boards which simulate the OBC was consuming too much time and would not be a part of the end-product anyway.

4.4. Telecommand

A telecommand is a command transmitted from a ground station to the cube satellite. Telecommands represent every type of action the OBC can be instructed to take. This includes:

- Changing the state of the OBC
- Gathering specific diagnostic information from subsystems
- Printing to the JTAG serial port

The execution status of telecommands is stored in the SD card's file system. For example, an incoming telecommand will be logged as "received" at the time it is received. When the command is executed it gets logged as "being executed" at that time. When it finished execution it gets logged as "finished execution" at that time. This execution status is treated as telemetry.

4.5. Finite State Machine

AlbertaSat's proposal to QB50 included a state machine which would be implemented in software. As such, a state machine is a functional requirement.

4.6. Autonomous Operation

The satellite must autonomously collect and save housekeeping data. It must be capable of deciding when to downlink data, buffer incoming telecommands, and make state transitions.

5. Design and Description of Operation

The software operates on a microcontroller with the high level IO diagram in figure 5.1. In the diagram, each block represents a board (ie, a PCB with ICs, conductive electrical paths, IO ports to connect to the board, etc). The arrows between boards represent a type of connection. The descriptive text explains what the connection is. There are some blocks within other blocks (GPS and EPS), these have their own individual code, but were never ported to their own boards. This diagram can be compared to figure A.2.1 (in the Future Work section) and some differences can be noted. First, the two scientific payload boards are missing, this is due a decision part way through the term to prioritize the development of code for the support systems first. The second thing to notice is that some of the connections are a different type. This is due to the testing boards being used. Despite a lot of effort, having CSP over I2C with the given hardware was proving to be a goal that could not be reached in the given time, and so some changes were made in order to have demonstrable communications.

It can also be noted that a 2GB SD card is used, which, when considering that communications may be limited for long periods of time, a larger memory size would be better. This smaller card is used because smaller memory cards have been proven to be more resilient to errors due to space (high energy particles causing bits to flip, etc.).

The OBC is a state driven machine, as described in 11. Software Design. There are 4 states used solely for the initial startup which comply with requirements set by QB50. There are 3 other operation states with different focuses. There is a state that prioritizes saving power by only allowing critical functions to execute, there is a state that prioritizes physically orientating the satellite, and lastly, a state that prioritizes the collection of science data from the (currently non-existent) payloads. It runs through these states due the HK retrieved from the EPS and ADCS. If the EPS is in a low power state, the OBC will switch to the power safe state, if the ADCS is in one of it's multiple detumbling states, the OBC will switch to the detumble state, and so on.

The OBC's main operation is to retrieve HK data from the EPS, ADCS, and COMM. The HK is saved as a telemetry packet on the SD card. When the ground station beacons the OBC (in this version, the "ground station" is a PC with a USB connection to the COMM board), and the signal is received the OBC will begin sending the telemetry to the ground station. In addition,

the ground station may send the OBC commands. The OBC will take some action based on the command and also save the execution status of the telecommand as telemetry data. The required range of commands is still not completely defined by AlbertaSat, but some basic commands for demonstration purposes were created, e.g., the OBC echos the commands it receives, and the OBC sends what state it is in.

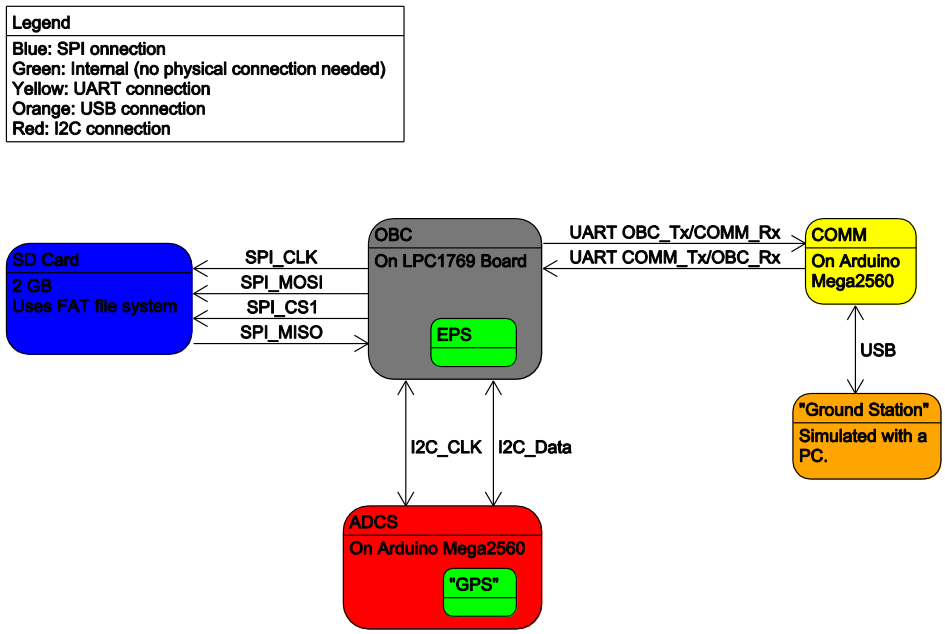


Fig. 5.1: Current board data flow.

A more detailed description of operations is provided in section 12. Software Design.

6. Bill of Materials

Count	Material	Cost ($)
1	LPC1769 Development Board	$33.25
2	Arduino Mega 2560	$103.82
2	4kOhm Resistors	$0.08
N/A	Wires	$0.67
	Total	$137.82

- 5x LPC1769 boards
- 1x SD Card breakout board to simulate SD card on the LPC1769.
- 1x SD Card for breakout board
- 6x USB - USB mini cables
- 1x UART <-> USB converter

7. Available Resources

7.1. FreeRTOS

The FreeRTOS operating system will run on the OBC.
Compile Size: 10 KiloBytes(Depends on compiler, architecture, and kernel configuration)
Performance: Minimum configuration uses 236 bytes of RAM, a more realistic number can be obtained if it becomes necessary.

7.2. CSP Library

Communications between subsystems use CSP packets as transport layer protocol (The implementation is a part of the Gomspace code repository). Work must be done to this library for it to run on anything that isn't Gomspace's Nanomind.

7.3. SwissCube Housekeeping Parameters

The SwissCube Housekeeping parameters describe the health and status of the satellite(temperatures, voltages, currents etc.). They will be organised in the Whole Orbit Data format.

8. Reusable Solutions

Device drivers for I2C and SPI have already been developed for the LPCxpresso 1769 with FreeRTOS compatibility. A C testing framework based off of munit (a two line testing framework) was written and used (the uncompiled header file is less than 2 KB).

9. Datasheet

Vdd = 3.3V

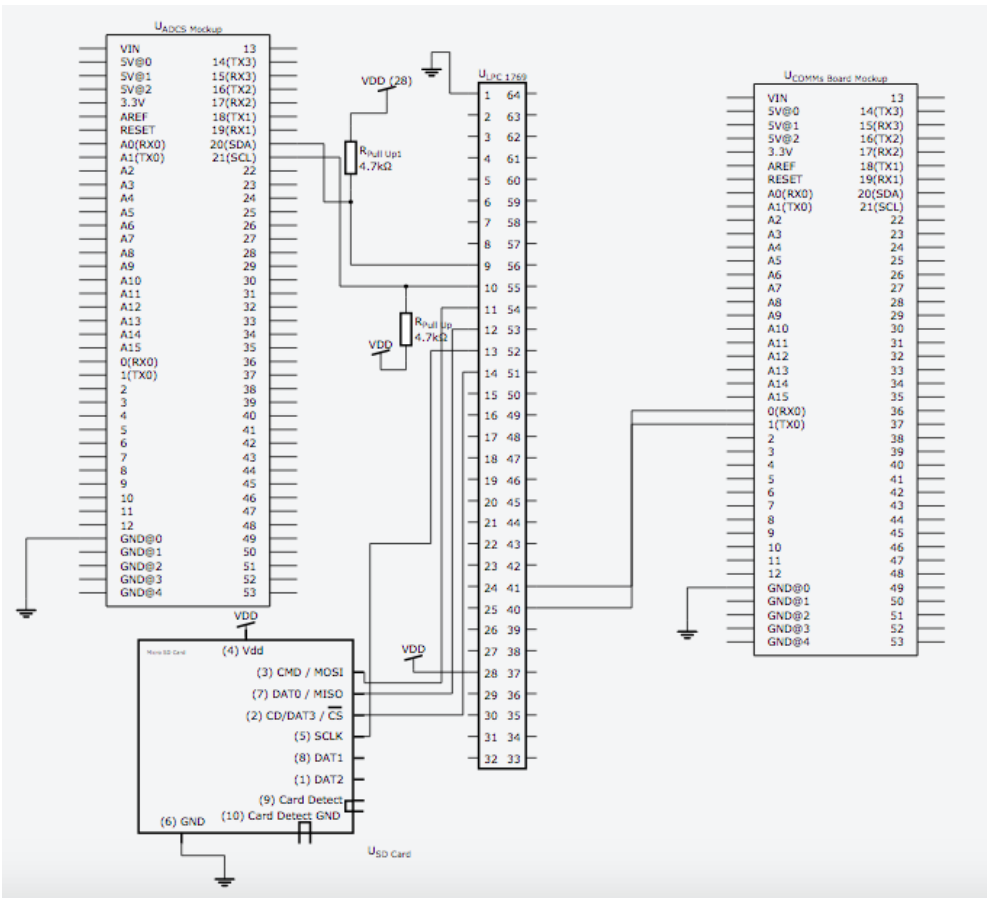


Figure 9.1: The 2 Arduino's connected to the LPC board.

[Please click here for scalable image]

IO Signals:

- ➔ Attitude Determination and Control System (Uses I2C)
  - ◆ Clock: I2C_Clk
  - ◆ Data: I2C_Data
- ➔ Communications (Uses UART)
  - ◆ UART: UART1
- ➔ Digital Fluxgate Magnetometer (Uses UART)
  - ◆ UART: UART0
- ➔ Multi-needle Langmuir Probe (Uses SPI)
  - ◆ Clock: SPI_Clk
  - ◆ MISO: SPI_Data
  - ◆ MOSI: SPI_Data
  - ◆ Chip Select: MnLP_CS
- ➔ MicroSD Card (Uses SPI)
  - ◆ Clock: SPI_Clk
  - ◆ MISO: SPI_Data
  - ◆ MOSI: SPI_Data
  - ◆ Chip Select: Mem_CS

There will be one data pin and one clock pin for all the I2C signals (ADC), There will also be one data pin and one clock pin for all the CSP signals (EPS, Comms, DFM). For the SPI signals there is an initial three pins for clock and data and then an extra select pin for each device, so that would be five in total (MnLP and MicroSD). This gives a total of nine IO pins.
There will also need to be pins to communicate with the Nanohub. Specifications (written by this capstone team) have not been made for this.

9.1. Quick Start Guide

Included in the appendix is a quick start guide. This guide walks through the process of installing the required software, flashing the hardware with the correct code, and starting the ground station.

9.2. Ground Station

To interact with OBC the ground station is used. The following operations will be helpful when using the ground station command prompt:

- help
  - Display help about the programs which can be used in the command prompt
- beacon --start
  - Send a beacon out to the satellite.
- beacon --stop
  - Stop sending a beacon to the satellite
- tc --help
  - List help info about how to telecommand the OBC.

10. Background Reading

10.1. The paper [1] describes a similar project undertaken by the University of Manitoba and provides useful insights on storage integrity, data redundancy and error reporting.

10.2. The Ground System Operations manual [2], published by the European Cooperation for Space Standardisation describes packet design and routing for telecommanding and telemetry reporting.

Please be advised, even though this list is short, the document by the European Cooperation is a very heavy and detailed read. It is by no means trivial to understand and requires significant time to be of benefit.

10.3. Given the large scope of the project, the AlbertaSat wiki [3] proved to be a valuable resource in providing us with the necessary background to get familiar with the work that had been done before we started working on the project.

11. Software Design

Software design is broken down into two sections. A command driven FSM (CDFSM) and the FreeRTOS tasks which use the CDFSM. This description is broken down to explain the CDFSM, commands, the FreeRTOS tasks, and the integration of all components.

11.1. Command Driven Finite State Machine (CDFSM)

First, the states and their transitions must be defined. In figure 12.1, the solid circle is the entry point to the state machine, boxes are the states, arrows are the requirements for a state transition, and diamonds represent a decision.

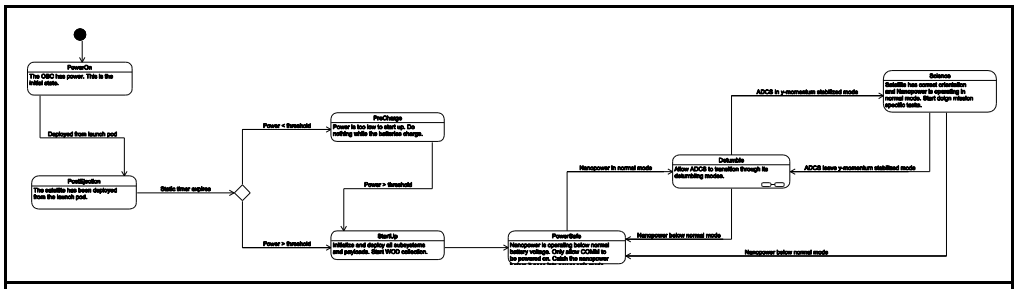


Figure 12.1: State Diagram

Next, this state machine is implemented in software such that it is command driven. This means that commands are sent to the state machine for execution. A command may or may not get executed depending on the current state of the machine. Figure 12.2 is a data flow diagram for the CDFSM. In the diagram, the state master acts as a controller and provides a relaying service to the states. Much like a store salesman acts like an access point to the warehouse inventory. The state slaves are the states of the CDFSM. A command is given to the state master, which then forwards this command to the current state. The state makes the decision to execute the command or ignore it. In addition, the current state will tell the state master what the next state is.

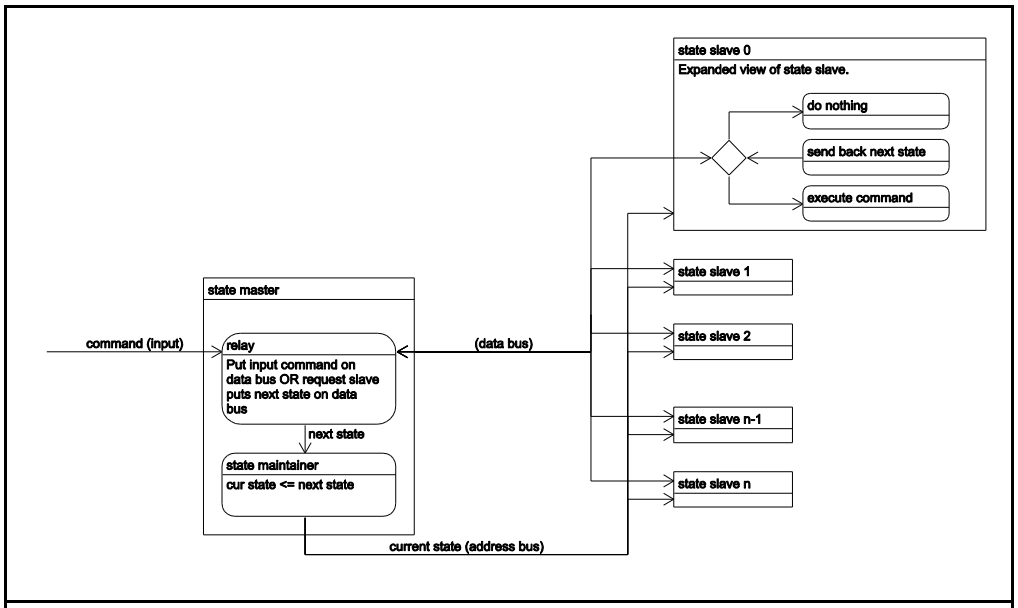


Figure 12.2: State machine data flow diagram.

11.2. Commands

Commands are implemented using a data structure, command_t, which has a function pointer to a method, called execute. The execute variable is set to point to a specific function depending on the purpose of the command.

```
struct command_t
{
  void *data;
  void (*execute)( command_t * );
};
```

This is what a command is. For example, lets say there is a command which collects HK data from the EPS. The code might look something like this.

```
static void execute_eps_hk_command( command_t *command );
{
  eps_t *eps;

  eps = (eps_t *) command->data;
  eps->collect_hk( eps );
}

void initialize_eps_hk_command( command_t *command, eps_t *eps_driver )
{
  command->data = (void *) eps_driver;
  command->execute = &execute_eps_hk_command;
}
```

In the application, a command_t structure would be allocated in memory, it would be initialized with the example initialization function above, then a pointer to it would be given to the CDFSM.

11.3. FreeRTOS Tasks

There is a FreeRTOS task for each major operating purpose of the satellite. This includes:

- Collecting HK data
- Processing commands from ground station
- Sending telemetry to ground station

The stack size of each task has not been determined. Likely, an exact calculation will not be done. The art of getting the correct stack size comes down to experience and guessing. As these tasks are expanded on, tested, debugged, etc, the stack size will change.

11.3.1. Collect Housekeeping

A unique command is allocated for each system that requires housekeeping data collection. These commands perform the following actions:

- Collect housekeeping from the system
- Package the data in a telemetry packet
- Save the telemetry packet in the SD card's file system

The commands are given to the CDFSM for execution. After each command (one for each system) is passed off to the CDFSM the task will delay for one minute.

11.3.2. Processing Ground Station Commands

This task acts as a server to the ground station. It opens a CSP socket and waits for incoming telecommands. When a telecommand is received:

- It is decoded
- A command_t structure is allocated
- The command_t structure is initialized based on the decoding of the telecommand
- The command_t structure is passed off to the CDFSM for execution

The task then loops and repeats.

Due to the memory and processing constraints of an embedded environment, this task is not designed to be a multitasking / forking / select server. Instead, the task will buffer incoming data, waiting until the ground station is no longer beaconing the OBC, then it will process the telecommands.

11.3.3. Transmitting Data to Ground Station

The task waits for a beacon from ground station. When this beacon is received, the task attempts to transmit as much telemetry as possible before the ground station stops beaconing, and is therefore out of range.

11.4. Integration

Figure 12.3 is a simplified class UML of the software. All commands are derived from the command_t structure and all states are derived from the state_t structure. The driver_toolkit_t structure acts a holder for all drivers. The purpose of this structure is to reduce the size of the uncompiled code, ie, how much the programmers have to type.

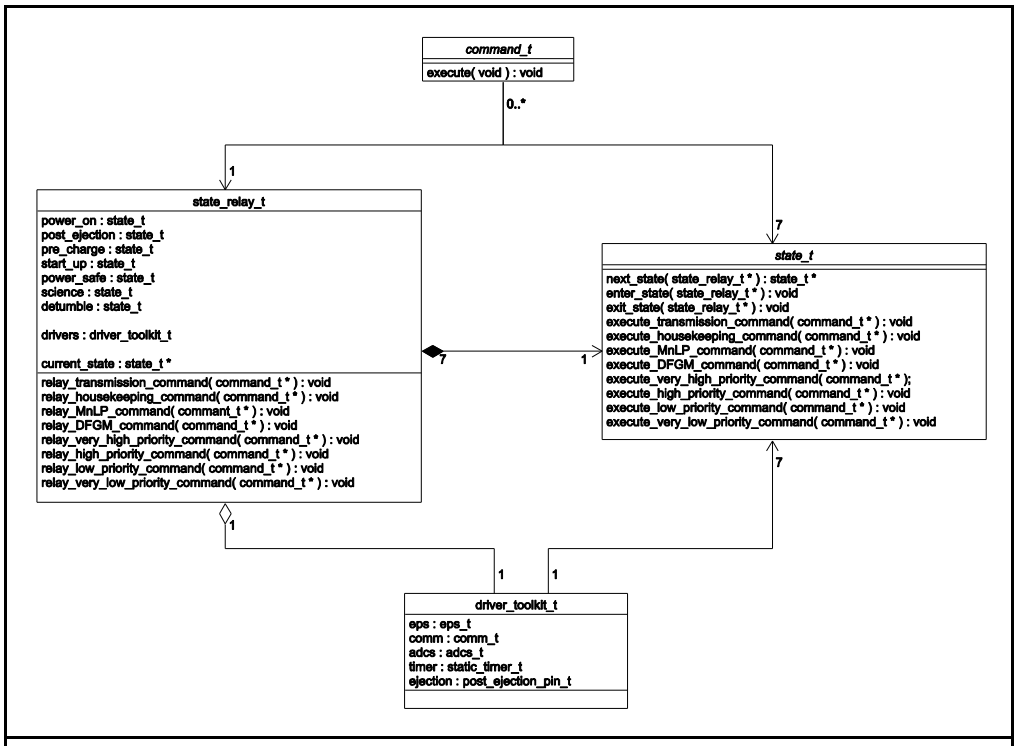


Figure 12.3: Simplified class UML.

12. Test Plan

Software:

Aggressive unit tests to test each function were designed and written. Integration tests to make sure modules work together correctly are also used. This includes tests which run for a long duration to ensure that as entropy builds the software does not crash.

The majority of these tests are done in a simple test suite, it is available along with test code as an AppNote on the course webpage. It is very similar in functionality to JUnit testing for Java. Tests are written individually and then listed off as part of a test suite. Each test is responsible for setting up the needed scenario and is independent of other tests (e.g.: test 1 turns on an led, and then checks if the led is on, it passes. Test 2 then checks if the led is on, but did not initially turn it on, and so it will fail). Each test must have at least one “Assert” line, which does a boolean check and gives a possible error message. All asserts in a test must be true for a test to be considered passed.

Hardware:

The LPC boards only purpose is to be used for testing, and so it does not need to be tested itself. An SD card reader is connected to the board but again that is only to be used for testing the software so minimal testing will be done on the hardware simply to ensure proper functionality.

13. Results of Experiments and Characterization

The OBC can autonomously walk through its states, downlink telemetry when beaconed by a ground station, collect and save housekeeping data, and buffer incoming telecommands and execute them. Incoming telecommands are correctly interpreted and executed.

14. Safety

Since we are only developing software and testing it on manufactured boards the safety concerns are very minimal. Some regular hazards that come with with using external boards are a (very unlikely) small shock or health risks from working at a computer too long.

A more concerning safety issue involves the satellite in orbit. As software developers, we are not sure what the risk is of the satellite colliding with space debris and other satellites. We imagine, however, the safety issues to other peoples property and our own does exist. A collision would likely mean the death of our satellite and what it collided with. Fortunately, the debris will burn up in the atmosphere and pose no safety risk to the life and property on planet Earth.

Finally, there is the safety issue of launching the satellite. This poses the greatest threat to human life. If the launch pod does not operate as intended,a crash may end the life of people and the project.

15. Environmental Impact

The orbit of the cube satellite will decay approximately two years after launch. This will lead to complete disintegration of the satellite with no impact the environment. The launch itself will have a significant environmental impact described in [4].
The microcontrollers used for mockups will be reused by AlbertaSat for future projects and we assume that they will be recycled appropriately after their use.

16. Sustainability

A summary of the power budget provided by the AlbertaSat team is attached at the very end of the appendix. The satellite contains solar cells to power itself.

17. References

[1] Alminde, L.; Bisgaard, M.; Vinther, D.; Viscor, T.; Ostergard, K., "Educational value and lessons learned from the AAU-CubeSat project," *Recent Advances in Space Technologies, 2003. RAST '03. International Conference on. Proceedings of* , pp.57,62, 20-22 Nov. 2003

[2] “Space Engineering”, 2003. [Online]. Available: http://cwe.ccsds.org/moims/docs/Work%20Completed%20%28Closed%20WGs%29/Packet%20Utilization%20Standard%20Birds%20of%20a%20Feather/Meeting%20Materials/200909%20Background/ECSS-E-70-41A%2830Jan2003%29.pdf

[3] *et al.,* AlbertaSat wiki, 2015. [Online]. Available: http://albertasat.wikispaces.com (Contact group member for authentication)

[4] *et al.,* “EcoSpace”, [Online]. Available: http://web.uvic.ca/~gsteeves/ecospace.pdf

Figures 5.1, 12.1, 12.2 and 12.3 are all original content by Brendan Bruner.

Appendix

A.1. Quick Start Guide

This guide describes the procedures necessary to replicate a demonstration of this project. Assumptions made are:

- Ubuntu or Lubuntu is running operating system, other versions of linux are untested.

A.1.1. Required Hardware

- 1 x LPC1769 development board + USB for flashing code
- 2 x Arduino Mega 2560 + USBs for flashing code
- Several wires for making electrical connections between boards
- 1 x Computer running Ubuntu or Lubuntu and at least three usb ports
- 1 x SD card holder and 2GB SD card
- 3 x 4k ohm resistors

A.1.2. Setup

Install the required software.

- LPCxpresso: download link at **www.lpcware.com/lpcxpresso/downloads/linux**
  - Navigate to the download directory in a terminal
  - Extract the files: `tar -xvzf Download_File.tar.gz`
  - Run the installer: `./Installer_File_Name`
  - Get a license. Follow the tutorial at this link for more help: **http://albertasat.wikispaces.com/Lubuntu+in+VirtualBox**
- C/C++ toolchain
  - `sudo apt-get install make`
  - `sudo apt-get install gcc`
  - `sudo apt-get install g++`
- Arduino sketch book
  - `sudo apt-get install arduino`
- Python and PySerial
  - `sudo apt-get install python`
  - `sudo apt-get install python-serial`

A.1.3 Connect The Hardware

Refer to the data sheet for a schematic of the hardware setup.

- Connect the i2c port one of the lpc board to the i2c port of one arduino. Put a pull up resistor on both the data and clock line.
- Connect the uart port 2 of the lpc board to the uart port 1 of the other arduino.
- Connect the spi port 1 of the lpc board to the SD card holder. The data sheet indicates which pin to use for chip select. Use the last resistor as a pull up resistor on the MOSI line.
- Connect the grounds of the lpc and two arduinos together, but do not connect the power supplies.

A.1.4. Flash The Software

- Flash the arduino connected by uart to the lpc board with the communications board software.
    - Start the arduino IDE, type 'arduino &' in a command line without quotation marks.
    - In the root directory of the software package, the arduino sketch is located at: /prototypes/arduino_comm_board/sketch/sketch.ino
    - Open this file in the arduino IDE.
    - Select the board: Tools -> Board -> Arduino Mega 2560 or Mega ADK.
    - Take note of the serial port: Tools -> Serial Port.
    - Upload the code by hitting the upload button.
- Start the ground station:
    - In the root directory of the software package, navigate to: /prototypes/ground_station
    - Run the python ground station script: python ground_station.py
    - An interactive command prompt will ask for the serial port of the arduino with the communications board software. This is the serial port noted earlier. After entering the correct serial port a message should display saying: port connected.
- Flash the ADCS software:
    - In the arduino IDE open the file (again, assuming the current directory is at the root of the software package): liba/Subsystem/arduino_mocks/adcs_mockup/adcs_mockup.ino
    - Upload this code to the second arduino (the one connected via i2c to the lpc board). WARNING: double check which serial port the code is uploaded to. Do not upload to the same arduino as the previous step.
- Start a debug session on the LPC1769:
    - Open LPCxpresso.
    - Import the following projects (directory paths start at the root of the software package)
        - /lib3rdParty/CMSISv1p30_LPC17xx
        - /lib3rdParty/Drivers_LPC1769
        - /lib3rdParty/FreeRTOS_LPC1769
        - /liba/Core
        - /liba/Subsystem
        - /main/Application_LPC1769
    - Click: Project -> Clean.. then make make sure 'Clean all projects' is selected and 'Build the entire workspace' is selected. Click 'OK'.
    - Right click the Application_LPC1769 project -> Debug -> C/C++ MCU Application.
    - After the code is flashed, click the green resume button in the tool panel. It looks like the play button on a VCR / DVD player.

A.1.5. Interact with the OBC

All code is running at this point. To interact with the OBC, switch to the ground station command prompt started earlier and type 'beacon --start' without quotes. The ground station is now communicating with the OBC and downlinking telemetry. Navigate to the directory: /prototypes/ground_station/telemetry to see the files. Send a telecommand to the ground station with 'tc -c 0 -p 3'. The OBC should echo the command to the LPCxpresso console. Terminate the beacon to let the satellite resume its normal operation 'beacon --stop'. Remember, type 'help' to see the help.

A.2. Future Work

Comment [1]: Once Brendan is done writing this it needs to be proof read.

- UTC time:
  - Writing an API to talk to the GPS module. This API would synchronize the OBCs time with the actual time collected from the GPS
  - A real time clock on the OBC to maintain the current time when the GPS is unavailable.
- Telemetry:
  - All telemetry collected should have a time stamp on it. This means each telemetry packet indicates the UTC time of its collection.
  - Telemetry should be packetized according to QB50s specification
  - Collecting all required telemetry instead of a subset of it
  - The ability to delete telemetry that ground station has confirmed successful receipt of
  - Able to prioritize the downlinking of telemetry through incoming telecommands
- File system:
  - Restructuring the way telemetry is saved in the file system to meet the requirements of the telemetry file system structure
  - Reliable fault tolerance of critical read and writes
  - Ability to reformat SD card on the fly
  - Data compression and error detection / correction
- Telecommands:
  - Implementing all required telecommands
  - Logging execution status of telecommands in non volatile memory
  - Generic decoder module for telecommands
  - Handling telecommands which contain extended data
  - Buffering incoming telecommands
- Subsystems:
  - Writing the API to use the EPS software that runs on an arduino
  - Add watchdog timers to EPS mock up which match the functionality of the watchdog timers on the nanopower EPS board
  - Finishing the API which enables communication with the communication software on an arduino
  - Transition mock ups from an arduino to an lpc1769

- Hardware Drivers (spi, i2c, and uart):
    - Remove dynamic memory allocation in i2c driver
    - Implement an API which is malleable to changing requirements (such as a change in target hardware)
- Payloads:
    - Write API to talk to payloads and collect scientific data
    - Determine priority of payloads and their operating duty cycles (when in science state)
    - Configure the MnLP windows mock up and use this to simulate MnLP
    - Get verilog source code for DFGM and implement it on an FPGA, spoof the scientific inputs, and use this for the DFGM mock up
- CSP:
    - Implement CSP over I2C
- States:
    - 30 minute blackout window the first time the post ejection state is entered must be done. Blackout window does not happen again unless the 30 minute duration did not expire the first time. Then, the remaining time is used for the blackout duration.
- Unit Testing:
    - Comprehensive tests for the file system API
    - Hardware driver API
    - Subsystem API for communication to arduinos
    - Unit testing the current telecommands implemented
    - Unit testing all future work
- Software / Hardware / Data Error Recovery
    - Detect and turn off faulty hardware
    - Recover from fatal software crashes
    - Watch dog timers to detect deadlock
    - Detection and handling of corrupted object code in flash
    - Detection and correction of corrupt data in non volatile memory
    - Detection and handling of I2C deadlock

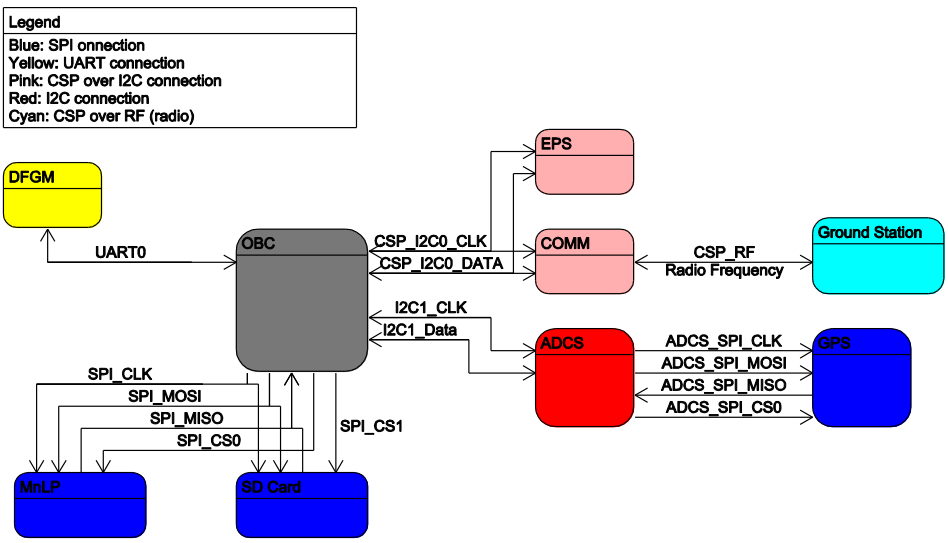


Figure A.2.1: The eventual layout of boards and connections to meet all AlbertaSat goals and specifications.

A.3. Hardware Documentation

Figures from sections 5 (figure 5.1) and 9 (figure 9.1) can used to give a good description of the hardware.

Current data flow (Fig. 5.1) shows that two Arduinos are simulating two of the subsystems (the COMM board and the ADCS) and are connected to the LPC 1769 board. The LPC board also has an SD card connected to it for logging of data. The ground station is simulated with a PC. EPS and GPS mockucps are contained on the LPC and ADCS mockup respectively.

The datasheet diagram shows a more realistic description of how the LPC and the two Arduinos are connected.There are two wires going from LPC UART2 pins (pins 40 and 41) to the COMM Arduino transmit and receive pins (pins 1 and 0 respectively). The connection going from the LPC1769 SDA1 and SCL1 pins (pins 9 and 10 respectively) going into the ADCS Arduino board's own SDA and SCL pins (20 and 21 respectively). The SD card is connected to the LPC in the following way:

- Chip Select to LPC pin 14
- MOSI to LPC pin 11
- SCK to LPC pin 12

- MISO to LPC pin 13

There is also a 4 kOhm pull up resistor between the MISO and VDD on SD card. SD card VDD is connected to the VDD pin 28 on the LPC and the ground on SD card is connected to the pin 1 ground on the LPC.

A.4 Source Code Section

Please see
https://bitbucket.org/bbruner0/albertasat-on-board-computer
for the most up to date source code (as of April 10, 2015). All modules successfully compile and execute.
Source code index table:

Filename	Status	Description
liba\Core\include		
command_allocator.h	C	Contains definitions for functions that manage allocation and freeing space for a command structure; not used in the project
command_buffer.h	C	Contains definition for a command buffer Macro; not used in the project
commands.h	T	Contains definition of the command structure and public methods used to operate on the structure
core_defines.h	T	Defines macros used by the static library
system_state_relay.h	T	Defines the system_state_relay structure and functions that operate on it
system_states.h	T	Defines the common structure used by states and prototypes the methods to initialize this structure depending on the state
task_defines.h	T	Defines constants for FreeRTOS tasks
liba\Core\source\commands		
command_buffer.c	C	Executes all the commands that have been buffered; not used in the project
commands.c	T	Executes a command and contains implementation of user-defined commands
liba\Core\source\relay		
system_state_relay.c	T	Manipulates states of the On Board Computer

liba\Core\source\states		
detumble.c	T	Does next state calculation from the Detumble state and decides what tasks are executing in it
post_ejection.c	T	Does next state calculation from the Post Ejection state and decides what tasks are executing in it
power_on.c	T	Does next state calculation from the Power On state and decides what tasks are executing in it
power_safe.c	T	Does next state calculation from the Power Safe state and decides what tasks are executing in it
pre_charge.c	T	Does next state calculation from the Pre-charge state and decides what tasks are executing in it
science.c	T	Does next state calculation from the Science state and decides what tasks are executing in it
start_up.c	T	Does next state calculation from the Start Up state and decides what tasks are executing in it
system_state_generic.c	T	Implements generic state. There is a function for executing in this state and a function for not executing. To define behaviour, a state has to link to one of the functions. The next from Generic State state is always Generic State
transmit.c	T	Does next state calculation from the Transmit state and decides what tasks are executing in it
liba\Subsystem\arduino_mocks\adcs_mockup		
adcs.h	T	Contains definitions for the structures and functions used by ADCS
adcs_mockup.ino	T	Simulates ADCS behaviour on an Arduino board
gps_module.c	T	Simulates GPS module for the ADCS mockup on the Arduino board
gps_module.h	T	Defines GPS module functions
liba\Subsystem\eps_mockup		
eps.h	T	Contains definitions used by the EPS mockup (provided by another AlbertaSat member)
eps_mockup.ino	C	Arduino mockup of the Electronic Power Supply

nanopower.c	C	Spoofs data output for the EPS mockup (provided by another AlbertaSat member)
nanopower.h	C	Contains definition of the spoof data output function for the EPS mockup (provided by another AlbertaSat member)
liba\Subsystem\include		
adcs.h	T	Contains definitions for the structures and functions used by ADCS
comm.h	T	Contains definitions for the structures and functions used by COMM board
driver_toolkit.h	T	Contains all drivers used by command, states, etc.
ejection_pin.h	T	Defines the API for querying the post ejection pin
eps.h	T	Contains definitions for the structures and functions used by EPS
i2c.h	T	Contains definitions for the I2C function used by the On Board Computer
mission_logger.h	T	Defines interaction of the On Board Computer with subsystems over the I2C protocol
mock_comms_struct.h	T	Defines interfacing with a third-party COMM objects
nanopower.h	T	Contains definition of the spoof data output function for the EPS mockup (provided by another AlbertaSat member)
static_timer.h	T	Defines behaviour of the static timer within the On Board Computer
telecommands_packet.h	T	Defines the packet format for a telecommand from ground station
telemetry_packet.h	T	Defines the packet format for a telemetry packet
uart.h	T	Contains prototypes for functions to set up UART2 communication on the LPC 1769 board
liba\Subsystem\source\adcs		
adcs_arduino_remote.c	T	Executes communication between the On Board Computer and the ADCS Arduino mockup
adcs_generic.c	T	Converts ADCS mode to string

adcs_lpc_local.c	T	An early version of an ADCS mockup which was simulated on the On Board Computer; not used in the final version
liba\Subsystem\source\comm		
comm_arduino_remote.c	T	Executes communication between the On Board Computer and the COMM Arduino mockup
comm_generic.c	T	Converts COMM mode to string
comm_lpc_local.c	T	An early version of a COMM mockup which was simulated on the On Board Computer; not used in the final version
liba\Subsystem\source\ejection_pin		
ejection_pin_lpc_pin.c	C	LPC GPIO ejection pin; not used in the final project
ejection_pin_mock.c	T	GPIO spoof of ejection pin
liba\Subsystem\source\eps		
eps_generic.c	T	Converts EPS mode to string
eps_lpc_local.c	T	An EPS mockup which was simulated on the On Board Computer
nanopower.c	T	Spoofs data output for the EPS mockup (provided by another AlbertaSat member)
liba\Subsystem\source\i2c		
i2c.c	T	Sets up I2C communication between an LPC 1769 board and a slave device
liba\Subsystem\source\mission_logger		
mission_logger.c	T	Takes data from subsystems and writes them to an SD card
mission_logger_fatfs_sd.c	T	Sets up file system on an SD card and provides interaction with the file system
liba\Subsystem\source\static_timer		
static_timer_ram_soft.c	T	Implements static timer for the On Board Computer

liba\Subsystem\source\toolkit		
driver_toolkit.c	T	Initializes driver toolkit for the LPC board for interaction with subsystems
liba\Subsystem\source\uart		
lpc_uart.c	T	Provides communication over UART2 to LPC 1769
main\Application_LPC1769		
main.c	T	The main function that controls communication between subsystems and the SD card, manages state transition, and sends telecommands and receives telemetry data

A.5 Supplementary Information on the Multi-Needle Langmuir Probe Software Specifications

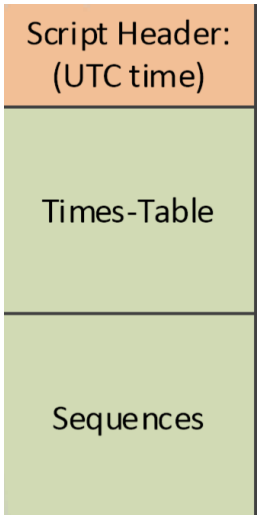
(All data and figures are from: T.A. Bekkeng, J. I. Moen, E. Trondsen, “Project: QB50”, 2014. [Online]. Available: https://bytebucket.org/bbruner0/albertasat-on-board-computer/wiki/1.%20Resources/1.1.%20DataSheets/MnLP/mNLP_icd_issue_4.pdf?rev=c72e1c6102968c9b61a053f42d96ed611640e1aa
(Will need authentication to access).

A.5.1 Script Handling

The syntax of the command to upload script to the payload is the following:

OBC_SCRIPT_UPLOAD(slot, script)

Here, ‘slot’ is the script buffer number and ‘script’ is the script command to be written to the buffer. If a script is contained at the location, it is overwritten.
The script structure is illustrated below



Excerpt from Table 13-1 from m-NLP Interface Control Document
On-Board M-NLP Script Buffer Usage

The following figure illustrates how the script operation should be run on CubeSat.

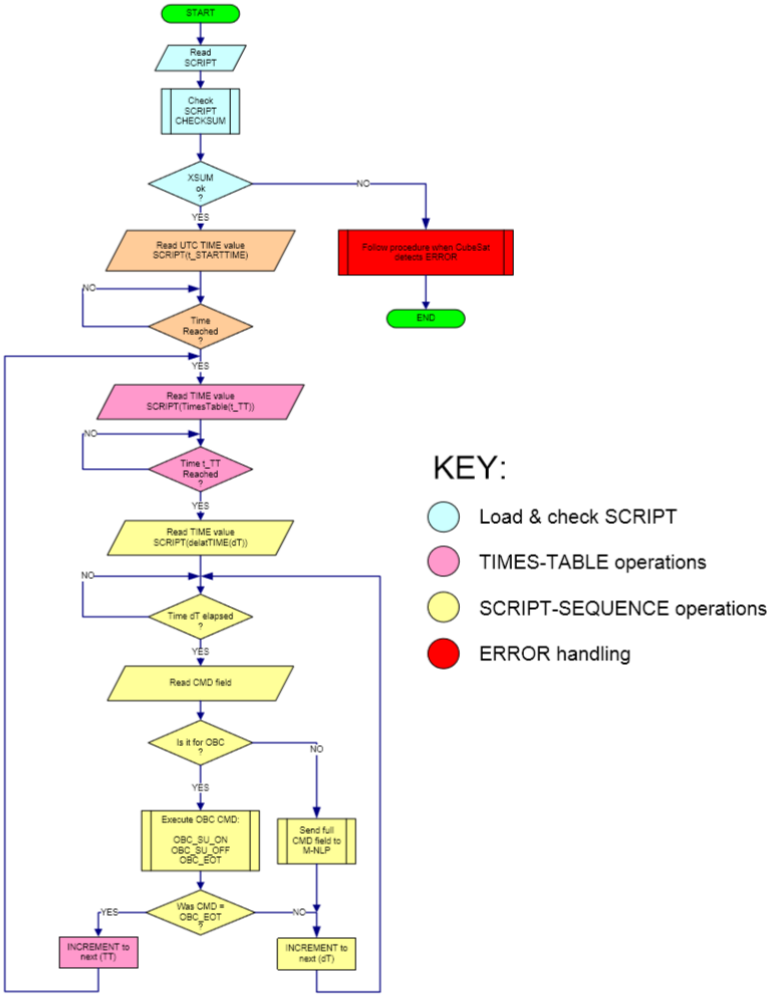


Figure 13-6 from m-NLP Interface Control Document
Script Operation Flow Chart

A.5.2 State Management and Error Handling

State transition diagram that the OBC manages for M-NLP

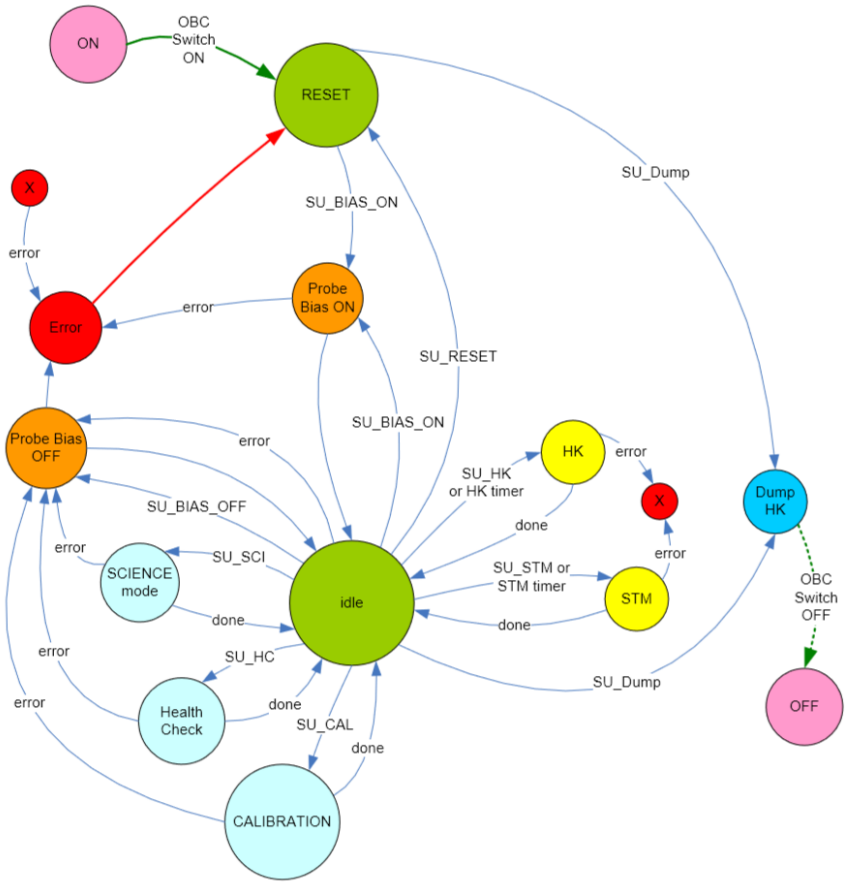


Figure 13-2 from m-NLP Interface Control Document
M-NLP State Transition Diagram

Error handling is done in one of two ways

In case when the M-NLP detected an error, the M-NLP aborts the executing operation, sets an ERROR_FLAG in the HK_STATUS_REG and sends an SU_ERR packet to the OBC. After that, the M-NLP goes to RESET state, where by definition all parameters are reset to the RESET values. From this state, the M-NLP is unable to support communication, which will cause a timeout detection in CubeSat.

If the M-NLP did not detect an error and the OBC has not received a data packet of any kind within the last 400 seconds, the following procedure is followed. First, the M-NLP is turned off. Next, an OBC_SU_ERR packet (defined in Table 13-4 of the m-NLP Interface Control Document) is generated by the OBC. It contains the scripts currently stored on the payload and will later be sent to the ground station for analysis. The science header described above (Table 13-2) is also added as the packet is considered a part of science data. After waiting 60 sec, M-NLP is turned back on and the script is run from the next times-table time-field value after the current UTC time.

	Assumptions
Altitude (km)	*350*
Albedo (%)	*-*
Attitude	*Inertial sun stare of the smallest face*
Orbit (minutes)	*90*
Max Harnessable Power per solar cell (W)	*1.2*
Any other assumptions	*30% TJ GaAS cells*

					Average Duty Cycle by Mode (%)		
Loads	Power Consumption (W)			Off mode	Power Safe Mode	Detumbling mode	Active Mission
	Direct from BAT (14.8V)	+5V Regulator	+3V Regulator				
GPS	*0.00*	*0.000*	*1.000*	*0%*	*0%*	*0%*	*2%*
P31us	*0.00*	*0.000*	*0.125*	*100%*	*100%*	*100%*	*100%*
Nanomind	*0.00*	*0.000*	*0.459*	*0%*	*100%*	*100%*	*100%*
Nanohub	*0.00*	*0.000*	*0.066*	*0%*	*0%*	*0%*	*100%*
UHF Rx	*0.00*	*0.000*	*0.231*	*0%*	*100%*	*100%*	*100%*
UHF Tx	*0.00*	*0.000*	*5.000*	*0%*	*2%*	*2%*	*5%*
Active THCS - Heaters	*0.00*	*7.000*	*0.000*	*5%*	*5%*	*0%*	*0%*
Fluxgate Magnetometer	*0.00*	*0.400*	*0.000*	*0%*	*0%*	*0%*	*50%*
MNLP operational	*0.00*	*0.600*	*0.250*	*0%*	*0%*	*0%*	*50%*
MNLP standby	*0.00*	*0.150*	*0.180*	*0%*	*0%*	*0%*	*50%*
SSADCS Y momentum control (daylight)	*0.07*	*0.240*	*0.337*	*0%*	*0%*	*0%*	*50%*
Sum loads (W)				0.475	1.265	1.567	2.576
Efficiency				88%	88%	88%	88%
Orbit Average Power Consumbed (W)				0.539	1.436	1.779	2.924
Orbit Average Power Generated (W)				4.97	4.97	4.97	4.97
Power Margin (%)				89%	71%	64%	41%
Energy Available to Recharge Battery (watt-min.)				398.94	318.26	287.39	184.31
Battery Depth of Discharge After One Eclipse (%)				1.1%	2.8%	3.5%	5.7%
Time Required to Recharge Battery After E.O.E. (min)				3.2	10.7	14.7	37.8