

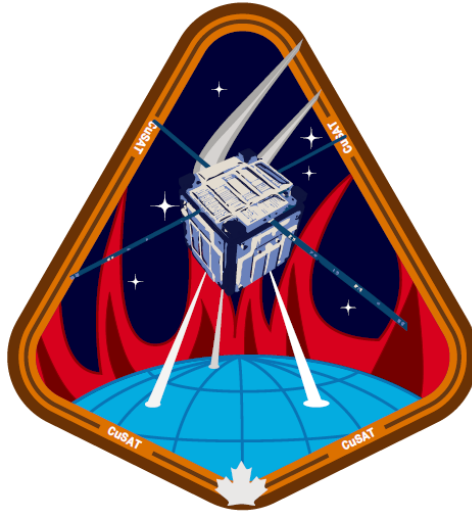


Carleton
UNIVERSITY



Department of
Mechanical & Aerospace
Engineering

Carleton CuSAT Design Project



Command & Data Handling

CuSAT-CDH-RPT-005

Revision 2

Engineering Lead:

Bruce Burlton

Team Members:

Brock Battochio

Isabelle Kosteniuk

Nathan Lim

Zeena Rashid

Document Revision Record			
Revision	<i>Date (yyyy-mm-dd)</i>	Revised By	Description
1	2017-12-07	BB	Initial Document Release
2	2017-12-08	NL	Formatting and editing

ABSTRACT

This report covers the work done in the 2017 Fall term by the CuSAT Command and Data Handling (CDH) subsystem. The purpose of CDH aboard CuSAT is to coordinate the data collection and processing of the spacecraft, as well as execute commands. Previous CDH teams had procured the Innovative Solutions In Space On-Board Computer (iOBC) and coded several subsystem task test programs. As of this term, most data interfaces between other spacecraft hardware and the iOBC have been determined and an Interface Control Document has been drafted. Issues with auxiliary interface connections have been identified and solutions have been examined. A first version of the operating system responsible for task-scheduling and command execution has been developed and undergone initial testing using subsystem shell tasks. iOBC responsibilities including timekeeping, error detection, and non-native matrix operations have been explored. Development of subsystem software tasks has continued, including ADCS, Payload, and GPS. Sample ADCS subroutines have been translated from MATLAB to C with assistance from the ADCS team using the MATLAB Coder tool. After much discussion, an SPI connection has been chosen as the Payload to iOBC interface, and preliminary image processing tests have been developed for Raspberry Pi. The GPS test program developed by previous CDH teams has been expanded, and a software design pattern has been developed for Communications, Thermal, and Power subsystem integration.

TABLE OF CONTENTS

ABSTRACT	ii
Acronyms.....	ix
1. Introduction	1
1.1 CuSAT Mission Objectives	1
1.2 Command & Data Handling (CDH) Purpose in CuSAT	1
1.2.1 Command Requirements	2
1.2.2 Data Handling Requirements	3
1.3 On-Board Computer Specifications	3
1.4 Previous Years' Work.....	5
2. Subsystem Integration.....	5
2.1 Payload	6
2.1.1 Payload Operation	6
2.1.2 Payload Interfaces.....	8
2.1.3 Payload Image Processing.....	9
2.1.4 Current Work	11
2.2 Attitude Determination and Control System (ADCS)	11
2.2.1 Flight Code for ADCS	12
2.2.2 Converting ADCS Algorithms to C Code	13
2.2.3 Validating the Programmed ADCS Code	14
2.2.4 Future Work for ADCS Integration	16
2.3 GPS	16
2.3.1 GPS Test Program.....	18
2.3.2 Future Work for GPS Integration	20
2.4 Communications.....	21
2.5 Power.....	23

2.6	Thermal.....	24
3.	Interfacing & Hardware connections.....	25
3.1	OBC Interface Specifications	25
3.2	Structural Layout of Subsystems on CuSAT.....	27
3.3	Subsystem Data Interfacing	28
3.4	Integration of Physical Connections	29
4.	OBC Programming.....	32
4.1	FreeRTOS.....	32
4.1.1	Task-Scheduling.....	33
4.2	List of Tasks	34
4.2.1	Command Handling Task	34
4.2.2	Attitude Task	36
4.2.3	Housekeeping Task	37
4.2.4	GPS Task	37
4.2.5	Payload Task.....	37
4.2.6	Communication Task.....	37
4.2.7	Error Checking Task.....	37
4.3	Timekeeping.....	38
4.4	Error Detection.....	39
4.5	Coding Standards.....	39
5.	Future Work.....	40
6.	References	41
7.	APPENDICES	42
7.1	Payload	42
7.2	ADCS	42
7.2.1	MATLAB to C Conversion Document	43

7.2.2	TRIAD MATLAB Files.....	47
7.2.3	Relevant C Files Generated by MATLAB Coder.....	48
7.3	GPS	54
7.3.1	Main.c.....	54
7.3.2	Testdata.txt	58
7.3.3	Output	59
7.4	Programming.....	59
7.4.1	JPL Coding Standards	59
7.4.2	Matrix Operations	60
7.5	iOBC files.....	61
7.5.1	obc.c	61
7.5.2	obc.h.....	65
7.5.3	adc.c	66

LIST OF FIGURES

Figure 1-1: CDH architecture.....	2
Figure 1-2: iOBC Block Diagram [1]	4
Figure 2-1: Overview of picture taking mode	6
Figure 2-2: Payload Interfaces	8
Figure 2-3: DMA data path	10
Figure 2-4:ADCS flight code, where blue boxes are inputs, green boxes are outputs, and red boxes are calculations.	12
Figure 2-5: Process for converting MATLAB to C code.....	14
Figure 2-6: Inputs and outputs for converted TRIAD algorithm displayed in command window.	15
Figure 2-7: GPS software task diagram.....	17
Figure 2-8: GPS software task data output.....	17
Figure 2-9: Sample GPS Receiver Position Sentence [6]	18
Figure 2-10: 2016/17 GPS Test Program [7].....	19
Figure 2-11: 2017/18 GPS Test Program	20
Figure 2-12: Communications downlink task block diagram	22
Figure 2-13: Communications downlink task data packet structure	22
Figure 2-14: Communications uplink software task block diagram.....	23
Figure 2-15: Communications uplink software task data structure	23
Figure 2-16: Power data structure	24
Figure 2-17: Thermal reading and writing	24
Figure 2-18: Example thermal data structure	25
Figure 3-1: Top view of OBC, showing locations of interface pins	26
Figure 3-2: CSKB pins, where coloured pins are assigned to an interface or are used for power. White pins are unassigned [1].	26
Figure 3-3: Structural depiction of CuSAT, where arrows point to subsystem boards [8].....	28
Figure 3-4: iOBC mounted with FM daughterboard. CSKB and pins from FM daughterboard are outlined in red [1]	30

Figure 4-1: Time sharing between two tasks with equal priority.....	33
Figure 4-2: FRAM memory usage and Command storage	35
Figure 4-3: Command Handling task block diagram	36

LIST OF TABLES

Table 1-1: iOBC Memory Types.....	5
Table 1-2: iOBC Data Interfaces	5
Table 3-1: iOBC Interface Specifications [1].....	27
Table 3-2: CuSAT data interface assignments	29
Table 4-1: List of tasks	34

ACRONYMS

Acronym	Description
AD&C	Attitude Determination & Control
ADC	Analog-to-Digital Conversion
ADCS	Attitude Determination and Control System
CDH	Command & Data Handling
CSKB	CubeSat Kit Bus
ECF	Earth-Centered Fixed (reference frame)
ECI	Earth-Centered Inertial (reference frame)
GPIO	General Purpose Input/Output
HAL	Hardware Abstraction Layer
I ² C	Inter-Integrated Circuit protocol (interface)
ICD	Interface Control Document
iOBC	Innovative Solutions in Space On-Board Computer
ISIS	Innovative Solutions in Space
LSP	LEO Satellite Position
LSV	LEO Satellite Velocity
PCB	Printed Circuit Board
PWM	Pulse-Width Modulation
RTC	Real-Time Clock
RTT	Real-Time Timer
SDK	Software Development Kit
SIT	Systems, Integration & Testing
SPI	Serial Peripheral Interface
TBC	To Be Confirmed
UART	Universal Asynchronous Receiver-Transmitter

1. INTRODUCTION

This report covers the work done by the CuSAT Command and Data Handling team in fall 2017. This section contains an overview of the mission requirements and the role that CDH plays in achieving those goals, as well as the architecture of the subsystem and its components. The integration of other subsystems, including ADCS, payload, and GPS with their respective data interfacing, software design, and testing is discussed in section 2. Section 3 covers the hardware connections used by the iOBC to connect to subsystem peripherals, as well as the issues presented by the iOBC pin structure and solutions moving forward. Section 4 contains a discussion of the operating system currently being developed to handle task scheduling and communication, and other relevant programming requirements such as coding standards, timekeeping, and error detection. Future work for the Command and Data Handling team is covered throughout the report and is summarized in section 5.

1.1 CuSAT Mission Objectives

The mission objective of the Carleton CuSAT Design Project is to demonstrate the scientific goal of forest fire detection using a satellite. To do this, the satellite shall collect thermal images of Canada and determine the location of hot spots. The images shall be processed and stored aboard the spacecraft before being transmitted to a ground station for further analysis. All other subsystems exist to support this primary goal.

This report will cover the role of the CDH subsystem in the CuSAT project, the current state of the subsystem, as well as results of work done this term and plans for continuing work.

1.2 Command & Data Handling (CDH) Purpose in CuSAT

The Command and Data Handling subsystem is responsible for coordinating the activity of each of the other subsystems, integrating hardware peripherals, and controlling the software operation of the spacecraft. All information gathered by subsystem hardware passes through the on-board computer for processing and storage, and all data transmitted to/from the ground station is passed through the computer as well. The main responsibilities of the CDH system can be divided into three categories: Command, Data Handling, and Storage. The requirements and objectives reported in

sections 1.2.1 and 1.2.2 were derived from section 6 of the CuSAT requirements matrix, which can be found on the CuSAT SVN server.

Figure 1-1: CDH architecture illustrates the high-level architecture of the hardware and software components of the CDH system. The main CDH subsystem component is the iOBC, which provides processing power, data storage, and data interfaces for communicating with other subsystem peripherals. The HAL shipped with the iOBC provides drivers and libraries for programming the mission software, which is currently in development by the CDH team.

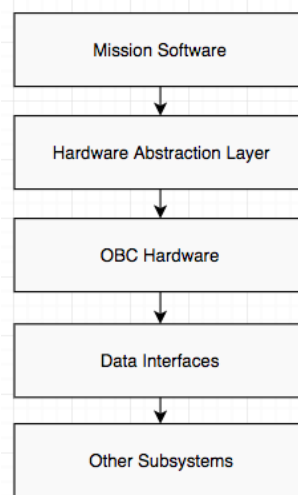


Figure 1-1: CDH architecture

1.2.1 Command Requirements

CDH is responsible for executing commands received from the ground station. These commands include taking pictures, collecting housekeeping data, and switching spacecraft modes. The commands can be either time tagged or immediate. The structure of the commands has yet to be finalized, but it will contain a command code, a time of execution, and a command number. A more detailed look at how commands work can be seen in section 4.2.1. Any commands critical to spacecraft operation shall be performed in a multi-step process to avoid any fatal faults. The operating system being developed by the CDH 2017/18 team will be responsible for initiating command execution and handling task scheduling.

1.2.2 Data Handling Requirements

CDH is required to process all data from all subsystems, as no other subsystem will have a dedicated computer. Subsystem data processing includes ADCS calculations, commands from ground, and all sensor data including thermal images from payload. ADCS sensor data shall be processed by the iOBC and necessary commands will be sent to the actuators. Payload camera images will be compared to a threshold to create a bitmap of hot spots. To transmit data to the ground station for further analysis, telemetry and payload data will be formatted into data packets. Error detection mechanisms shall be employed throughout the life cycle of each data type to ensure accuracy.

CDH is responsible for storing three categories of data: commands, telemetry, and payload data. Each of these categories have lifetime requirements and size limitations. Commands received from the ground station shall be stored aboard the satellite for 7 days. Telemetry shall be collected from each of the subsystems, excluding structures. Payload data will likely be stored primarily as a list of pixels above a threshold, with supplementary complete images if capabilities exist. Memory capabilities of the iOBC are discussed in section 1.3.

1.3 On-Board Computer Specifications

The iOBC procured by previous years CDH team was selected due to its large number of data interfaces, redundant memory, relatively fast, low power processor, and flight heritage. Figure 1-2 illustrates the block diagram of the iOBC, including memory, interfaces, power, timing, and CPU. The iOBC motherboard weighs 94 g including flight daughterboard, uses a 32-bit 400 MHz ARM9 processor, and runs using a 3.3 V power supply [1]. Additionally, the iOBC uses two redundant real-time clocks for timekeeping, and supports a low power mode. As mentioned previously, the iOBC also shipped with HAL libraries to communicate with hardware components.

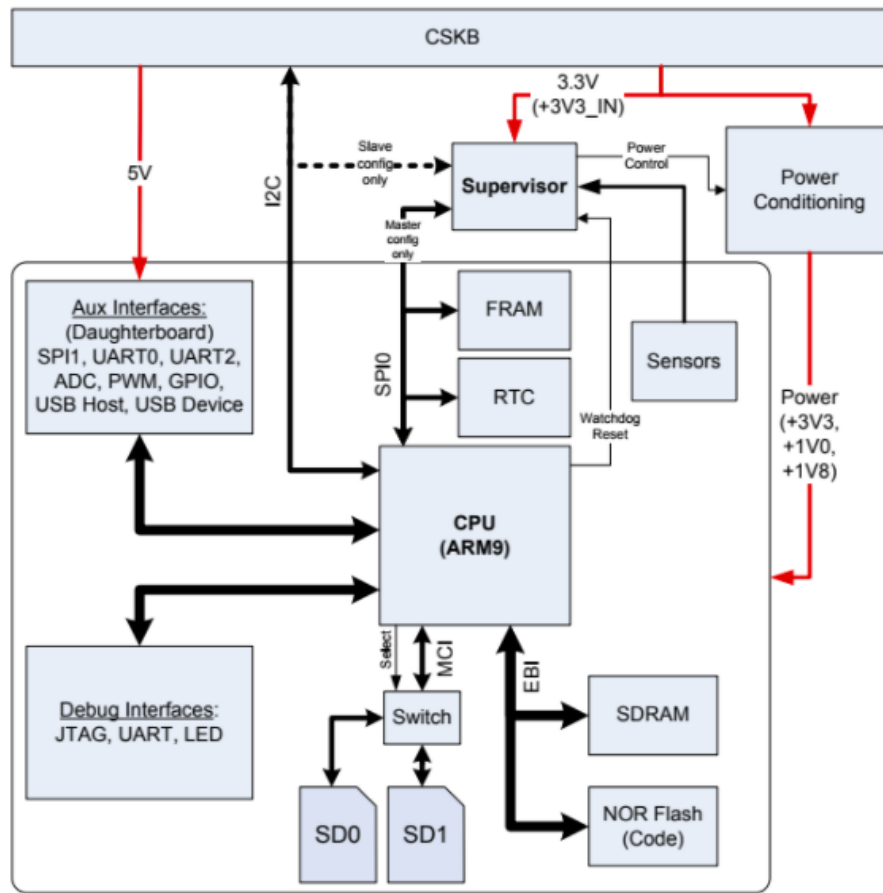


Figure 1-2: iOBC Block Diagram [1]

As shown in Figure 1-2, the iOBC has multiple interface connectors, including the CSKB/PC104 and the auxiliary interfaces block. Many of the connections that will be used by other subsystem peripherals are only accessible through the auxiliary interfaces, including SPI, PWM, and ADC. Section 3 discusses the issues this presents and possible solutions.

Table 1-1 lists each of the memory types contained on the iOBC, their size, and intended use. Table 1-2 lists each of the data interfaces supported by the iOBC and the number of channels. Section 2 discusses the connections used by each subsystem peripheral in further detail and section 3 discusses how these connections will be implemented and what challenges are faced.

Table 1-1: iOBC Memory Types

Memory Type	Size
FRAM	256 kB
NOR	1 MB
SDRAM	32 MB
SD	2 x 4 GB

Table 1-2: iOBC Data Interfaces

Data Interface	# Channels
I2C	Bus
SPI	Bus
UART	2 (RS232, RS232/422/485)
ADC	8
PWM	6
GPIO	27
USB	2

1.4 Previous Years' Work

Previous CuSAT CDH teams succeeded in sourcing and procuring the iOBC, deconstructing technical information about the computer, and forming preliminary plans for hardware peripheral connections & software development. Test programs were developed and run on the iOBC using demo software shipped with the HAL, such as a GPS integration test, micro-bolometer test, and task scheduler test.

2. SUBSYSTEM INTEGRATION

Communication between the iOBC and the other subsystems is imperative to meet mission objectives of CuSAT. This section of the report outlines the interfacing requirements between the iOBC and each subsystem and describes the work done by the CDH team to fulfill these requirements.

Specifically, this section will discuss payload interfacing, followed by ADCS, GPS, Communications, Power, and Thermal subsystems.

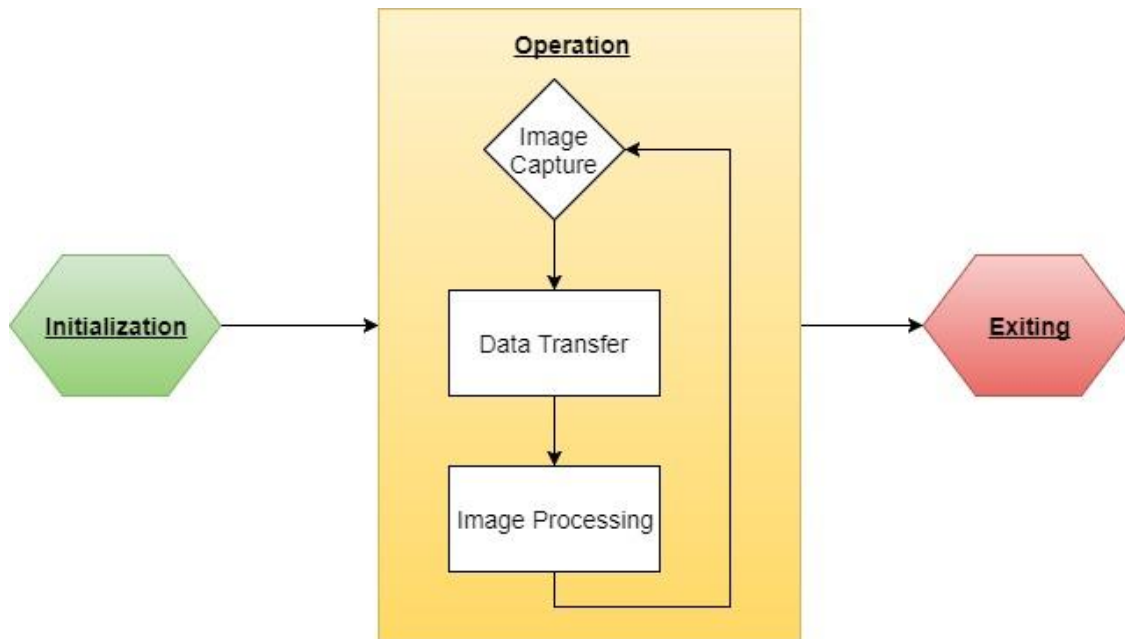
2.1 Payload

One of the major mission requirements of the Command and Data Handling subsystem is processing payload images for the detection of forest fires. The payload section of this report will provide a thorough overview of the picture taking mode of operation, the interfacing of the OBC and payload, and image processing.

2.1.1 Payload Operation

The following section will provide a thorough overview of sequence of events that transpire during picture taking mode of operation, and the associated interactions between the payload and the OBC. As illustrated in the figure below, the picture taking mode of operation can be organized into three main stages, consisting of initialization, operation, and exiting.

Figure 2-1: Overview of picture taking mode



Beginning on the left of Figure 2-1, initialization is the first stage of this picture taking mode. Initialization represents the interactions that occur when the OBC is entering the picture taking mode of operation. The initialization stage involves two main tasks, which are powering the camera on and preparing the camera for operation.

While it is evident that the camera will need to be powered throughout the duration of picture taking mode, it is still yet to be determined if the camera will be powered indefinitely, or whether it will be shut down during the other modes of operation. Nonetheless, there are several design features that will have to be considered for either option. If it is decided that the camera will only be powered on during picture taking mode, then the OBC will need to consider the time delay as the camera is booted. This is to ensure that the camera is prepared for image capturing when the satellite is over a desired location. If the camera were to be always on, the OBC would not need to consider the delay, however, we would need to evaluate the electrical load of the camera.

If the camera will only be on during picture taking mode, the initialization stage would involve the OBC first sending a command to the camera to turn on, and then relay a command notifying the camera to prepare for imaging. Once the camera is on and initialized, the OBC will signal the camera to begin capturing frames.

Seen in the middle section of Figure 2-1, the next stage of the picture taking mode is operation. Operation represents the image capture and data handling processes of picture taking mode. During Operation, the payload and OBC will enter a repeated process where every 6.1 seconds an image will be captured, the raw image data will be transferred to the OBC and processed. The transfer of raw image data and image processing will be discussed in greater depths among later sections. The image capture, data transfer and processing procedure will be iterated until the desired number of pictures have been taken.

The final stage of the picture taking mode, exiting, represents when all images have been taken and processed, and the OBC departing the picture taking mode and transitioning to the next mode of highest priority.

2.1.2 Payload Interfaces

Interfaces serve as the link that binds two or more devices, enabling them to communicate and relay information. Now that a high-level overview of the interactions between the OBC and payload has been established, the ensuing section will specify the interfaces that connect the OBC and payload, as well as their intended functions. Throughout the 2017 fall semester, many modifications were made to optimize the efficiency of the interface design, Figure 2 found below provides a simplified diagram of the interfaces that will be implemented:

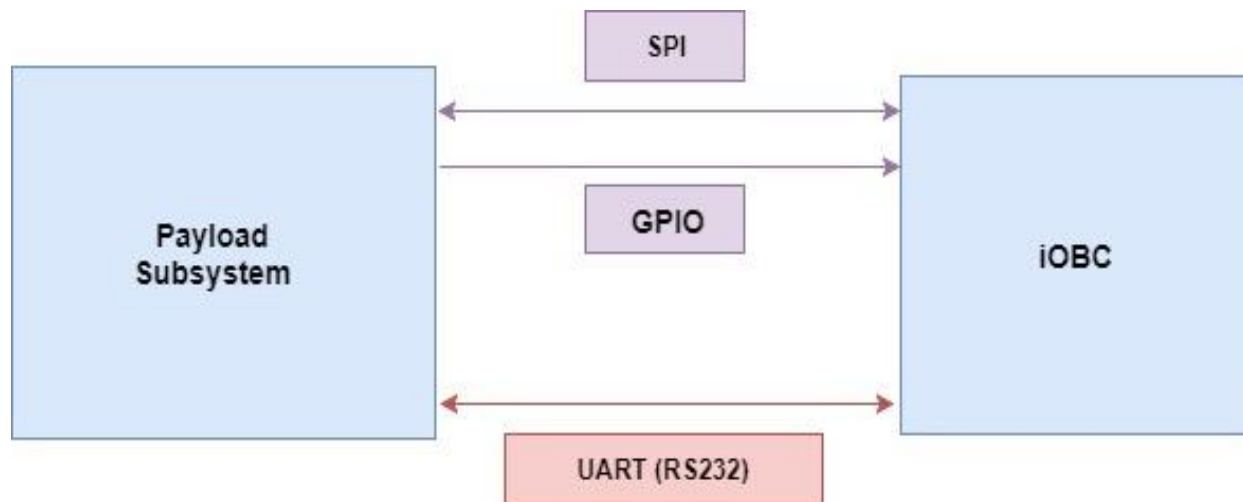


Figure 2-2: Payload Interfaces

Located at the top of Figure 2-2, the first interface that will be discussed is Serial Peripheral Interface or SPI. SPI will serve as the interface used to transmit raw image data from the payload to the OBC. SPI is a bi-directional interface that is normally used to connect a single master device to multiple slave devices, and consists of four wires; Serial Clock, Master In Slave Out (MISO), Master Out Slave In (MOSI), and a Slave Select. The slave select line is used to identify which slave the master is communicating with. However, the payload is the only subsystem interfacing via SPI and the only slave device. Interfacing the payload and OBC via SPI will require a total of five pins, four for SPI and a single GPIO pin. SPI is an interface that immediately expects data to be transferred, therefore a GPIO pin will be needed to inform the OBC to activate SPI when a transfer is needed. For example,

when the camera has captured a frame and the data is ready to be transmitted, the GPIO will interrupt the OBC to notify it to initialize SPI and allow the transfer of data.

The second interface that between payload and the OBC is UART (RS232). UART is an interface used for serial communication over a computer or peripheral device, and RS232 is a standard for serial communication. This interface will be used to transmit commands from the OBC and housekeeping data from the payload. This interface will specifically be used when the OBC enters picture taking mode, a command will be sent from the OBC via the UART RS232 line to tell the camera to prepare for imaging, and then to begin capturing frames.

2.1.3 Payload Image Processing

One of the most significant requirements and functions of the OBC is the processing of raw image data to identify thermal anomalies, enabling the CuSAT-1 satellite to detect forest fires. The following section will examine how raw image data will be transferred to the OBC, and images will be processed. The method that will be used to transmit image data from the camera to the OBC is Direct Memory Access or DMA. DMA is a technique that enables a peripheral I/O device to directly communicate data to or from a specified memory location. Unlike programmed or interrupt based I/O techniques that require constant activity from the central processing unit (CPU), DMA is unique because it allows the transfer of data between a peripheral device and memory without involvement of the CPU.

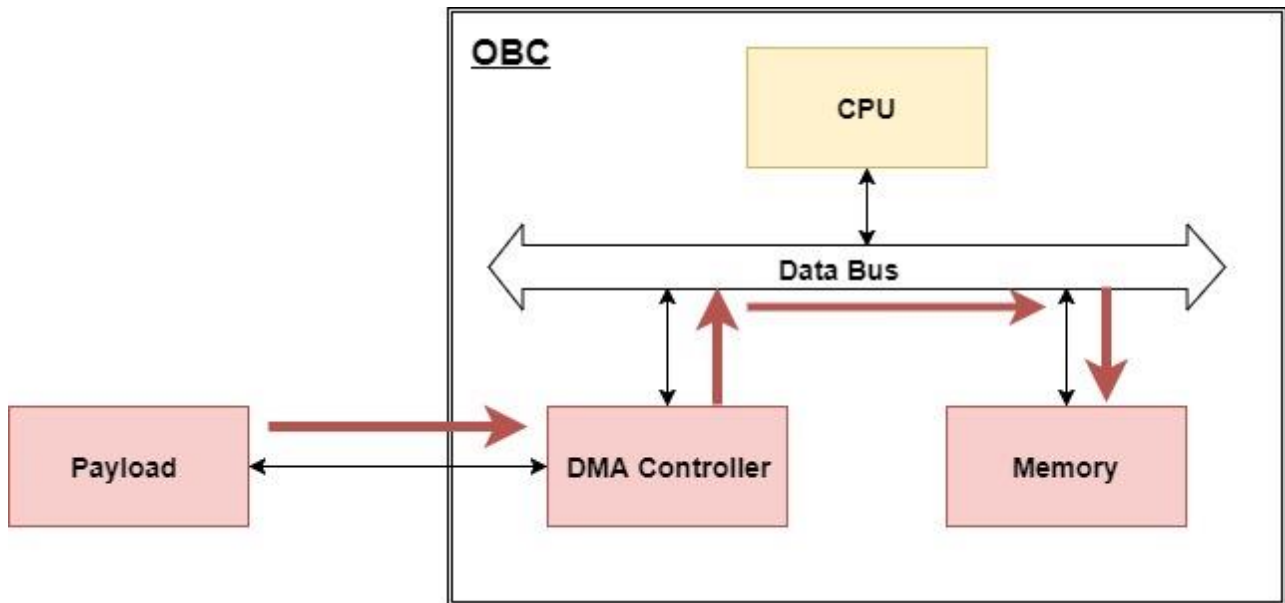


Figure 2-3: DMA data path

The illustration shown above demonstrates the distinct data path of a DMA transfer between payload and the OBC's main memory. As depicted by the red arrow in Figure 2-3, the data transmitted directly from the payload to memory. By circumventing the CPU, DMA allows image data to be transferred much faster, and more importantly, the CPU is released from the bus. Therefore, during the execution of a DMA transfer the CPU can simultaneously perform other processing tasks. DMA transfers are managed by a mechanism called a DMA controller. A DMA controller is the internal device that sets the word size of the data, total number of words to be sent, establishes the pointer to a specified address, and generates the control signals needed to facilitate the transfer of data. In the case of payload image data, once the camera has captured a frame, the GPIO will pulse the OBC notifying it that data is ready to be transferred. The DMA controller will then take control over the bus, set required parameters and commence the transfer. Once the data transfer is complete, the DMA controller will release the bus to the CPU and the CPU will process the raw image data.

The next topic of image processing is pixel thresholding, which is the mechanism that will be used to identify hotspot pixel locations. The payload camera will capture images in greyscale format, with each individual pixel consisting of 16 bits that represent the pixels greyscale intensity, or temperature. The raw image data will be transferred to memory and stored as an array of 16-bit

words. For processing, each element of the array will be scanned and compared to a set threshold value. If the element is above the threshold, the pixel coordinates will be stored with their corresponding GPS, orientation and time information for later downlink. The current design would have the raw image data initially being transferred to main memory, and then hotspot pixel locations will be stored within the 4 GB SD cards within the OBC.

2.1.4 Current Work

In terms of payload, the objective of the command and data handling team throughout the 2017 fall semester was to replicate the pixel thresholding mechanism of the OBC using a microcontroller called a Raspberry Pi. Code was developed in python that takes in a greyscale image, identifies the height and width of the image, and then loops through each pixel comparing the pixels greyscale intensity to a defined threshold value. If a pixel is found to be above the threshold, the program will then output the pixels x and y coordinates. Appendix 7.1 depicts the code used to preform this function on the Raspberry Pi. In terms of future work, our aim is to implement a similar version of this function using a development board to process real images taken from the newly acquired INO test camera.

2.2 Attitude Determination and Control System (ADCS)

The forest fire detection mission of CuSAT relies on the satellite's ability to meet strict pointing requirements during hotspot detection and to determine its attitude for accurate mapping of hotspot locations. The satellite must also be able to point towards the sun to charge its cells such that it can power its electrical components [2]. To meet these pointing requirements, the iOBC must be able to:

- process sensor data, perform all calculations, and output commands per ADCS flight code;
- meet all data interfacing requirements of the ADCS components; and
- capture telemetry data detailing satellite attitude.

While interfacing requirements will be discussed in Section 3 and telemetry data collection will be covered in Section 5, this section will focus on how the iOBC will handle ADCS Flight Code. Specifically, it will provide an overview of the Flight Code, including a brief description of the data

being received by inputs and being sent to outputs. This will be followed by an outline of the process to transform MATLAB code from the ADCS team into usable code for the OBC. This section will then discuss the current work for programming the ADCS Flight Code, and will outline future work to be done such that CuSAT can meet all pointing requirements for a successful mission.

2.2.1 Flight Code for ADCS

The algorithms involved in satellite attitude determination & control (AD&C) are being developed by the ADCS team and are being tested through real-world simulations. The overall scheme for the on-board ADCS is referred to as its flight code. A diagram representing this flight code was devised by the ADCS team, and a modified version, as it pertains to the OBC, is shown in Figure 2-4 below [2]. The three main elements in this diagram are the inputs to the system, which are in blue, the ADCS algorithms in red, and the outputs from these algorithms in green. Each of these three main elements will each be discussed in this section, and will each be examined separately when programming the flight code on the OBC.

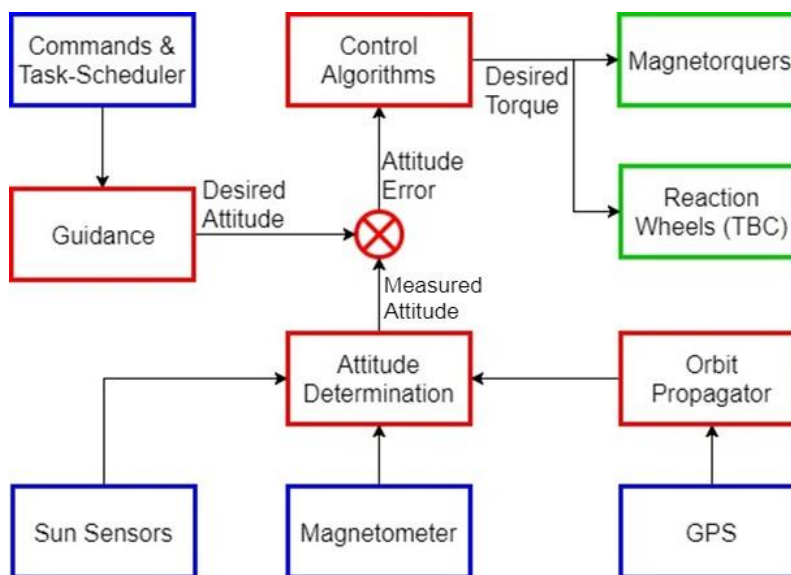


Figure 2-4:ADCS flight code, where blue boxes are inputs, green boxes are outputs, and red boxes are calculations.

The inputs into the ADCS flight code include GPS information, which will be discussed in Section 2.3, task-scheduler commands, described in Section 4, and data from sun sensors and a magnetometer. The four sun sensors will be manufactured in-house, and their outputs will be relayed

to the iOBC via an analog-to-digital converter through the I²C interface [2]. The iOBC must represent the data as a vector providing sun sensor data in the three coordinate axes. The magnetometer, the LIS3MDL, is a three-axis magnetic field sensor with a built-in analog-to-digital converter. Its 16-bit output for the magnetic field measured in each of the three axes will be sent via I²C to the iOBC [3]. The iOBC must also convert all sensor data from sensors' body frames to the satellite's body frame.

To re-orient the satellite to meet pointing requirements, the ADCS flight code will send outputs from its control laws as desired torques to three magnetorquers and three reaction wheels (TBC). Each magnetorquer will be driven by one of three DRV8838 H-bridge motor drivers, which in turn will be controlled by the iOBC via pulse-width modulation (PWM). The reaction wheels are yet to be purchased, but will likely each have a driver controlled via I²C by the OBC.

The third main element in the flight code diagram includes algorithms for ADCS such as the TRIAD algorithm for attitude determination, B-Dot algorithm for control, and the orbit propagator, among others. These algorithms are coded in MATLAB and are currently being modified by the ADCS team based on their simulation results.

2.2.2 Converting ADCS Algorithms to C Code

To implement the ADCS algorithms on the OBC, they must be converted from MATLAB to C code. The steps taken for this process are detailed in Figure 2-5 below.

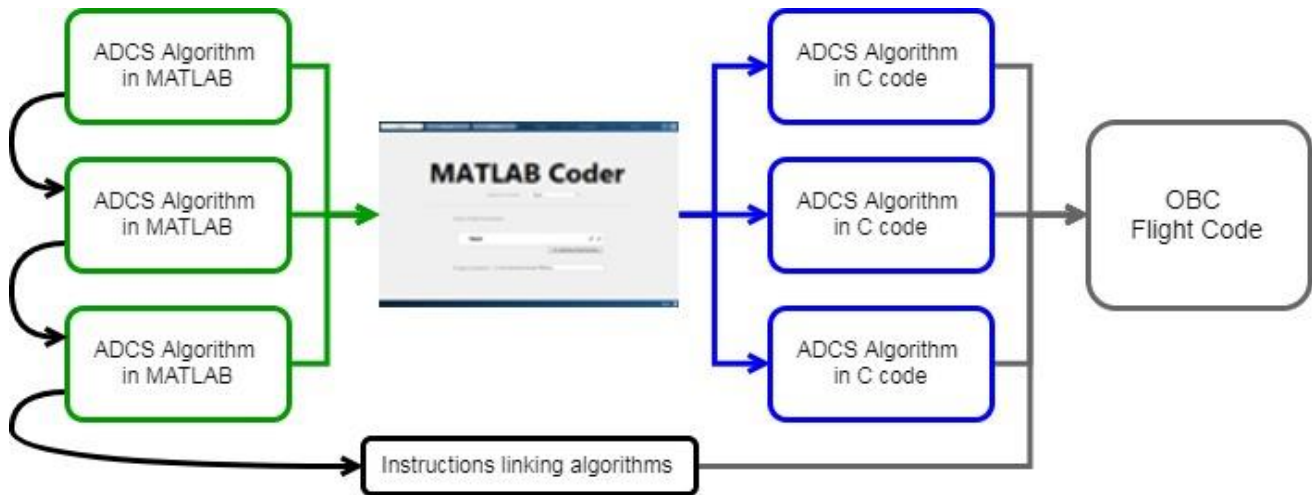


Figure 2-5: Process for converting MATLAB to C code

In the diagram above, the MATLAB code (green) and instructions linking algorithms (black) will be provided by the ADCS team. The ADCS algorithms include MATLAB functions for attitude determination, the orbit propagator, control laws, etc. These functions will be converted to C code that has been optimized for the device to which it will be written; for the OBC, this will be an ARM9 processor [4]. To ensure conversion has been done correctly, a MATLAB to C Document is being written by the CDH team, and can be found in Appendix 7.2.1.

Once converted, each algorithm will be tested on a development board, as discussed in Section 2.2.3 below. After validating that the converted codes demonstrate identical behaviour to the original MATLAB algorithms, they will be programmed onto the iOBC using the instructions linking them to one another. Completion of this process will satisfy the requirement of programming all ADCS calculations on the OBC.

2.2.3 Validating the Programmed ADCS Code

The TRIAD algorithm for attitude determination is being used to test the process described in Section 2.2.2 above. This algorithm receives a magnetic field vector and a sun vector in body-reference frame from the sensors, as well as a magnetic field vector and sun vector in the Earth-centered inertial (ECI) frame, generated by the orbit propagator. The TRIAD algorithm outputs a rotation matrix which estimates the satellite's attitude based on the given set of vectors [5].

This algorithm was written by a member of the ADCS team in MATLAB, found in Appendix 7.2.2, and in this test, the MATLAB Coder settings were configured to generate C code for the host computer. Appendix 7.2.3 of this report provides the relevant source and header C files. The MATLAB Coder also generated an example main file within which testing could be performed. This file was edited in Visual Studio to initialize the converted TRIAD algorithm's four input arguments with four vectors that have a known output rotation matrix when fed into the TRIAD algorithm. The file was also edited to print the inputs and outputs of the converted TRIAD algorithm into a command window. A screenshot of the result is displayed in Figure 2-6 below.

```
For inputs:
sb = 0.781400 0.375100 0.498700
mb = 0.616300 0.707500 -0.345900
si = 0.267300 0.534500 0.801800
mi = -0.312400 0.937000 0.156200

Rotation Matrix:
0.566187 -0.788075 0.241598
0.780296 0.417971 -0.465232
0.265662 0.451927 0.851582
```

Figure 2-6: Inputs and outputs for converted TRIAD algorithm displayed in command window.

This result indicated the converted TRIAD algorithm functioned as intended. The next validation step in this process is verifying that this code can be uploaded onto a development board and produce similar results. Currently, code is being generated using the MATLAB Coder for an ARM Cortex-M7 device. This code will be uploaded onto the STM32 Nucleo-144 development board and a hyper-terminal such as TeraTerm Pro will be used to observe if its results can match those of the code executed on the host computer. This development board was selected for testing because it has many of the same characteristics as the iOBC including a similar microprocessor and many of the same data interfaces. The following section details work to be done with this development board to program the ADCS flight code as well as additional steps to be taken for complete ADCS integration with the OBC.

2.2.4 Future Work for ADCS Integration

As discussed above, the converted TRIAD algorithm will be implemented on the development board to verify its functionality on a device like the iOBC. Efficiency and memory usage will also be examined to optimize the code being generated by the MATLAB Coder. Once adequate testing with the TRIAD algorithm has been done, the remaining ADCS algorithms will be converted and uploaded onto the development board. In addition to programming the ADCS algorithms, the sensor readings and commands to actuators must be programmed and tested until a suitable flight code can be formulated, which can then be implemented on the iOBC.

In addition to work to be done on the flight code, determining how to collect satellite attitude information for housekeeping data and for identifying hotspot locations must be addressed. Finally, the data interfacing requirements of the ADCS components must be met to ensure that the flight code may be realized on the physical satellite. Interfacing requirements will be discussed in Section 3.

2.3 GPS

The GPS subsystem is composed of the SkyFox GPS receiver and antenna. The function of this subsystem is to provide the spacecraft with position, velocity, and time information.

The GPS receiver shall be connected to the iOBC through a UART connection, which has been successfully tested. Data has been collected from the GPS receiver using a test antenna and personal computer, and a connection has been successfully established between the receiver and the iOBC by the previous years CDH team.

Figure 2-7 illustrates a high-level design of the GPS software task. This section will explain each block in the diagram and discuss methods of achieving the required functionality, as well as the current state of the test program. Figure 2-8 shows the data output produced by the task, including position, velocity, and time data, as well as a checksum value for error detection. Initially, the task is put into running mode by the OS. The first operation performed by the GPS task is reading from the UART bus, as shown in Figure 2-7.

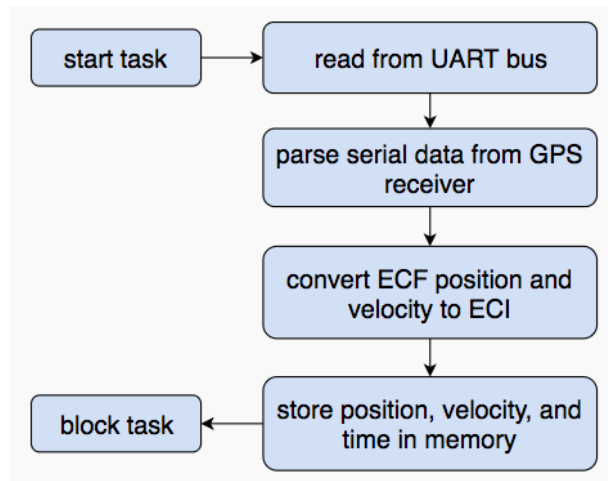
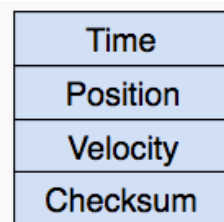
**Figure 2-7: GPS software task diagram****Figure 2-8: GPS software task data output**

Figure 2-9 contains sample data provided by SkyFox labs. The receiver outputs a high volume of data, including time in various formats, latitude longitude, error detection information, most of which is not relevant to CuSAT's mission. As described by the block diagram in Figure 2-7, the GPS software task is responsible for parsing the receiver output and extracting time, position vectors, and velocity vectors as efficiently as possible.

LSP – LEO Satellite Position – The piNAV SENTENCE

\$PSLSP,193772.0585851,780,3963889.204,1001383.917,4879428.991,5,4.5*72

LSP	LEO satellite position
193772.0585851	GPS time [s]
780	GPS week
3963889.204	X position referenced to WGS-84 [m]
1001383.917	Y position referenced to WGS-84 [m]
4879428.991	Z position referenced to WGS-84 [m]
5	Number of satellites used for PVT
4.5	Position Dilution of Precision (PDOP)
*72	The checksum data, always begin with *

Figure 2-9: Sample GPS Receiver Position Sentence [6]

It is important to note that the position and velocity output by the receiver are in WGS-84, an ECF reference frame. ADCS calculations will be performed using position and velocity in an ECI reference frame, therefore the GPS software task will also be required to convert data between the two systems. The subroutine used to perform this transformation shall be developed in MATLAB by the ADCS team and provided to the CDH team for implementation in C.

2.3.1 GPS Test Program

The 2016/2017 CDH team developed an preliminary GPS test program adapted from demo programs shipped with the iOBC. Figure 2-10 shows a block diagram of the program test program that has been successfully implemented. This section will discuss the work done this year, namely that functionality has been added to more accurately parse the GPS sentences and convert ECF to ECI, and recommendations have been made for improving data parsing efficiency.

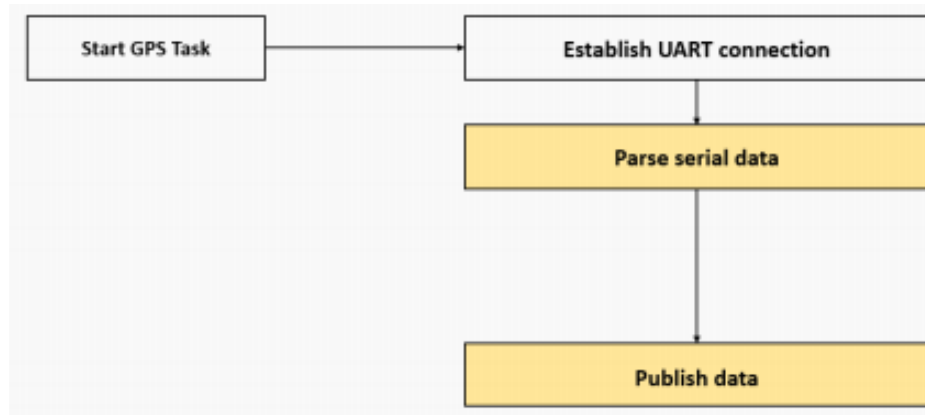


Figure 2-10: 2016/17 GPS Test Program [7]

Figure 2-11 shows the current state of the GPS test program, deconstructed into three main sections: data parsing, data transformation, and data publishing. Several changes were made to the data parsing block, including adapting the code for testing on a personal computer and storing LSP and LSV sentences in tokenized arrays. The data transformation section was developed and can convert UTC time to a value in seconds for use in the calculation of the rotation angle subroutine to be provided by ADCS. As shown in Figure 2-11, the program extracts position/velocity information from the LSP and LSV arrays and converts the values from strings to double floating point. The rotation matrix is calculated and used to transform position/velocity to ECI. To improve efficiency of this calculation, the matrix is deconstructed and scalar operations are performed. After performing the data transformation, the data is published to the development machine's console in a format similar to Figure 2-8.

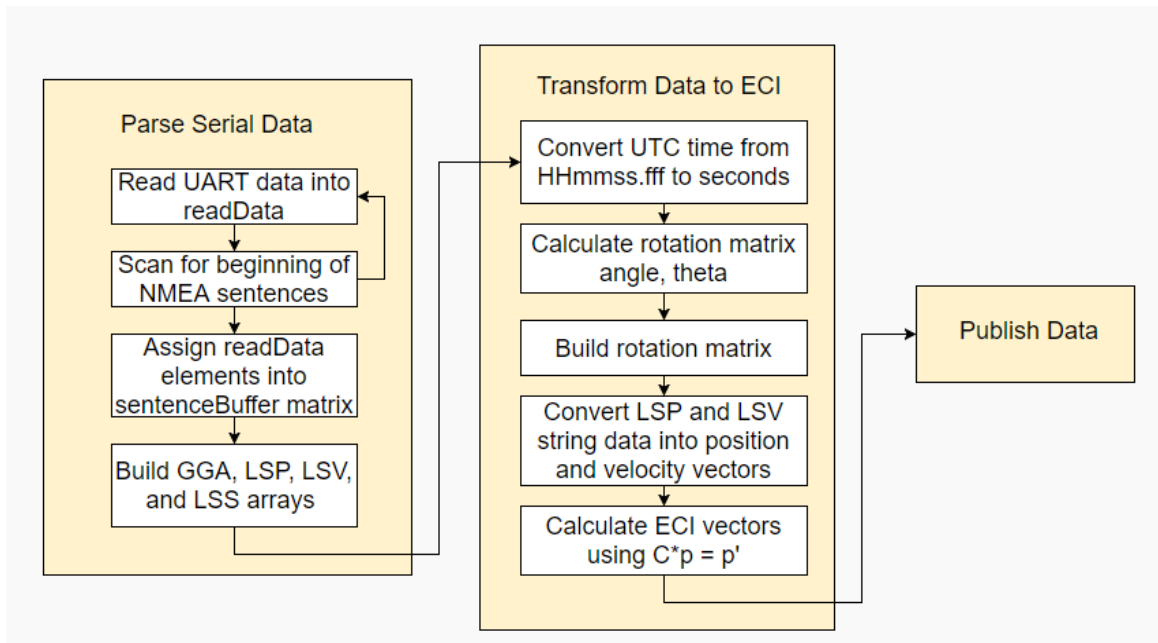


Figure 2-11: 2017/18 GPS Test Program

The data output from the test program has not yet been verified, as the ECF to ECI conversion subroutine has yet to be implemented. As well, the sample data used in testing will be verified to ensure location data is in fact being output in WGS-84. As a sanity check the location data, collected at Carleton University, will be compared to local position information. Appendix 7.3 contains the complete test program as well as sample input and output.

2.3.2 Future Work for GPS Integration

There are several opportunities for improving the GPS test program before development of the mission software commences. This includes improvements to each of the three blocks discussed in section 2.3.1, as well as issues regarding coding standards and hardware connection.

Additional improvements will be made to the data parsing algorithm. Currently, the program searches the UART bus character by character using nested if-statements to search for a 6-character string that indicates the beginning of the NMEA sentences. Modifying this to search for a shorter, still unique string could improve program efficiency. Storing the entirety of the GPS receiver output in a matrix is another inefficiency in the task, considering only three pieces of information are relevant. Directly extracting position, velocity, and time data from the receiver output would likely result in

much faster execution time and optimized memory use. Section 4.1 discusses coding standards that apply to the GPS software task. Currently, the test program throws several type conversion errors when compiled with the full recommended flag set. These warnings will be resolved before finalizing the test software. Finally, the GPS components are subject to hardware connection issues that are covered in section 3.

2.4 Communications

The communication subsystem consists of transceiver and antenna. The transceiver shall connect to iOBC through I2C to facilitate data transmission to and from ground. The communications software tasks, uplink and downlink, are responsible for receiving commands from the transceiver to be executed on the spacecraft and sending payload/telemetry data to the transceiver, respectively. Before sending data to transceiver for transmission to ground, the downlink task shall create data packets including header information such as packet length. Packets should include data type and time tags for each data word, and error detection fields shall be verified before transmission. Figure 2-12 illustrates the design of the downlink task, and Figure 2-13 shows the data packet structure.

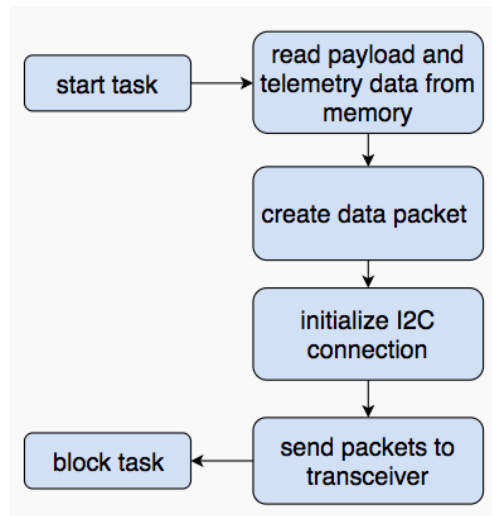


Figure 2-12: Communications downlink task block diagram

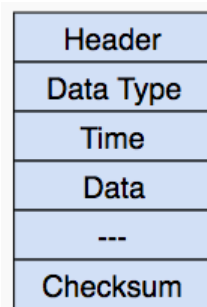


Figure 2-13: Communications downlink task data packet structure

The uplink task is designed as shown in Figure 2-14. Commands received by the transceiver shall be sent through an I2C connection to be parsed by the software task. The commands shall have associated execution times, and will be stored in a command table as described in section 4. Figure 2-15 shows the data structure associated with this task.

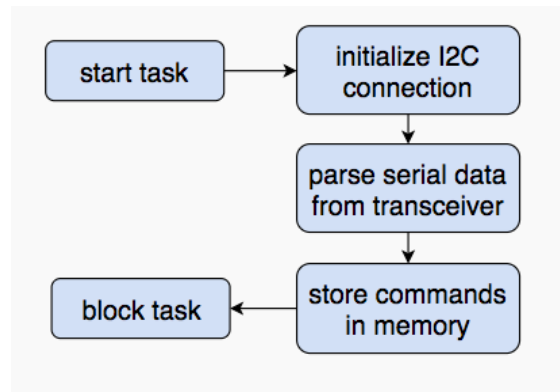


Figure 2-14: Communications uplink software task block diagram

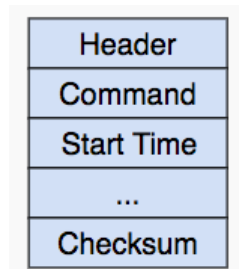


Figure 2-15: Communications uplink software task data structure

2.5 Power

The responsibility of the Power subsystem is to read the power levels of the satellite battery and solar cells. When power levels are low, it will inform the spacecraft that it should be changed to safety mode.

Figure 2-16: Power data structure shows a potential structure to store the power subsystem's data.

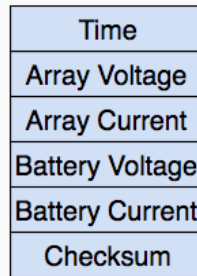


Figure 2-16: Power data structure

2.6 Thermal

The thermal subsystem on the satellite is passive, and therefore needs no computer control. The iOBC needs only to monitor the temperatures throughout the satellite and store them for error checking or housekeeping data purposes.

The thermistors output a voltage indicating the current temperature, and can be converted to a digital value using an ADC. The iOBC has one ADC unit with eight channels and there will be eight thermistors.

The figure below shows how a software task responsible for reading the thermistor values might look.

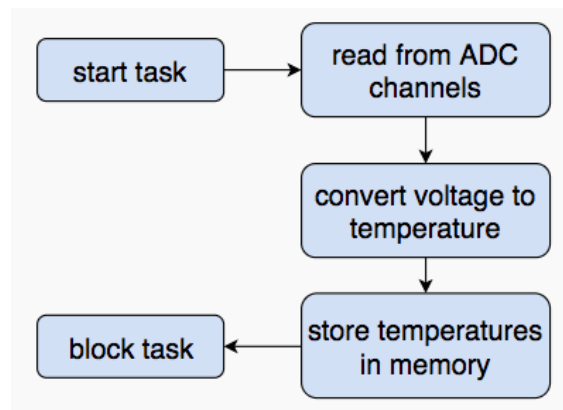


Figure 2-17: Thermal reading and writing

A structure can be built to store all thermistor temperatures, the time they were read, and a checksum.

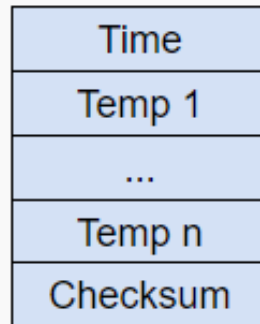


Figure 2-18: Example thermal data structure

3. INTERFACING & HARDWARE CONNECTIONS

The section above describes the requirements to be met by the CDH team from the other subsystems of CuSAT. To fulfil these requirements and facilitate communication between the on-board computer and the components of the other subsystems, data interfacing specifications must align. In Section 1.3, Table 1-2: iOBC Data Interfaces lists the existing data interfaces on the OBC. This section of the report will provide more detailed specifications about these interfaces, including their pin locations on the iOBC itself. This section will then discuss the structural layout of CuSAT, to understand the physical location of the subsystem boards that will require connections to the OBC. This section will then discuss the known data interface requirements from subsystem components, and will conclude with a discussion how to integrate hardware that will fulfil these interfacing requirements while conforming to any physical limitations imposed by the structure of CuSAT.

3.1 OBC Interface Specifications

The pins supplying all data interfaces from the iOBC are located on either the CubeSat Kit Bus (CSKB), which is a PC/104 standard bus, or within a set of J2 pins located just below the CSKB on the iOBC board, as shown in Figure 3-1 below.

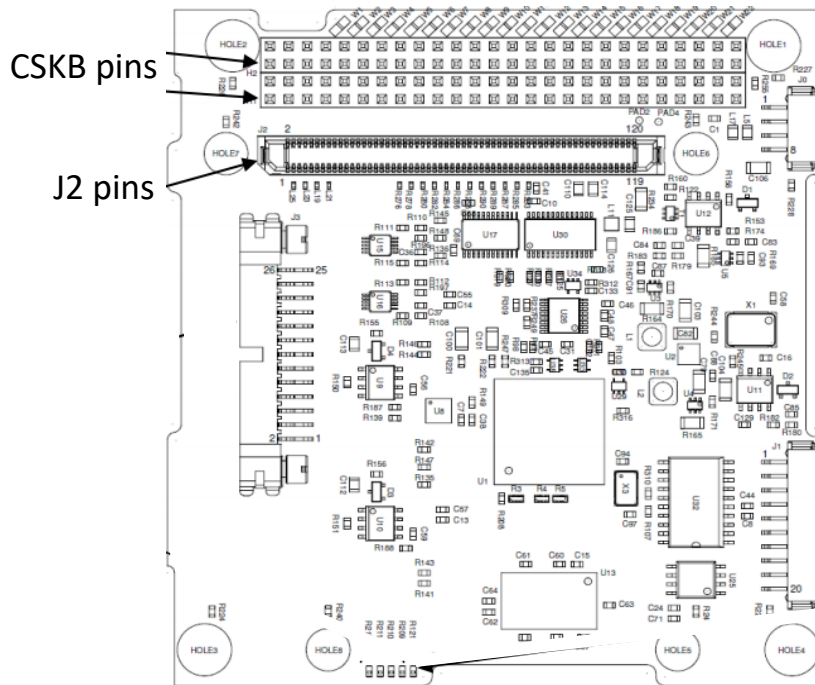


Figure 3-1: Top view of OBC, showing locations of interface pins

In Figure 3-1, there are two distinct rows of J2 pins and 2 sets of pins in the CSKB. The J2 pins are numbered from 1 to 120, starting from the lower row, left-most pin, and continuing such that odd-numbered pins increment from 1 to 119 in the lower row and even-numbered pins increment from 2 to 120 in the upper row. Each of their pin assignments can be found in the iOBC datasheet [1]. The CSKB pins are numbered and categorized as either H1 or H2 pins, where H2 pins are above the H1 pins. Labelled CSKB pins are given in Figure 3-2 below.

[illegible]

Figure 3-2: CSKB pins, where coloured pins are assigned to an interface or are used for power. White pins are unassigned [1].

In addition to the locations of the data interface pins, other specifications that must be adhered to when interfacing with the iOBC include the quantity of each interface available, the available data transfer rates of each interface and if DMA is possible, any limits they have on the data size being transferred, and the voltage requirements for signals being passed along each interface.

These specifications are summarized in Table 3-1 below. In this table, V_{IL} and V_{IH} refer to low and high input to the OBC; V_{OL} and V_{OH} refer to low and high output from the OBC; V_{RL} and V_{RH} refer to low and high receive by the OBC; and V_{TL} and V_{TH} refer to low and high transmit from the OBC. Refer to the table of acronyms at the beginning of this report for definitions of the acronyms for interfaces in the table below.

Table 3-1: iOBC Interface Specifications [1]

Interface	Quantity	Location	Data Transfer	Voltage Requirements
I ² C	1 (to multiple devices)	CSKB	Fast mode ($\leq 400\text{ kbit/s}$) (DMA possible)	$V_{IL} = -0.5$ to 0.9 V $V_{IH} = 2.52$ to 5.5 V $V_{OL} = 0$ to 0.2 V $V_{OH} = 5.5$ V (max)
SPI	1 (to up to 8 devices)	J2 pins	$\leq 10\text{ Mbit/s}$ (DMA possible)	$V_{IL} = -0.3$ to 0.8 V $V_{IH} = 2$ to 3.6 V
UART	2 (1x TTL or RS232; 1x RS485 or RS232)	J2 pins	$\leq 500\text{ kbit/s}$ (DMA possible)	$V_{RL} = -0.5$ to 0.89 V $V_{RH} = 2.52$ to 5.5 V $V_{TL} = 0$ V (min) $V_{TH} = 3.3$ V (max)
ADC	8	J2 pins	$\leq 125\text{ kHz}$ 8- or 10-bit output (DMA possible)	$V_{IH} = 0$ to 2.5 V
PWM	6	J2 pins	$\leq 33\text{ MHz}$	$V_{OL} = 0$ V (min) $V_{OH} = 3.3$ V (max)
GPIO	27	5 on CSKB 22 on J2 pins	Not specified	$V_{IL} = -0.3$ to 0.8 V $V_{IH} = 2$ to 3.6 V

The specifications listed in the table above should be revisited whenever a new part for CuSAT is to be purchased or built to ensure compatibility with the iOBC.

3.2 Structural Layout of Subsystems on CuSAT

In addition to conforming to the data interface specifications of the OBC, hardware connections between the iOBC and other subsystem components must not be impeded by physical limitations imposed by the structural design of the satellite. Many of the subsystems on CuSAT will have their own printed circuit board (PCB) that will house their components and internal circuitry. These boards

will be approximately equal in size, at about 10 cm by 10 cm. Each board will have a PC/104 bus along one of its sides, such that each subsystem's PC/104 bus will align with the iOBC's CSKB. The assumed layout of the boards in CuSAT is depicted as a side-view in Figure 3-3 below.

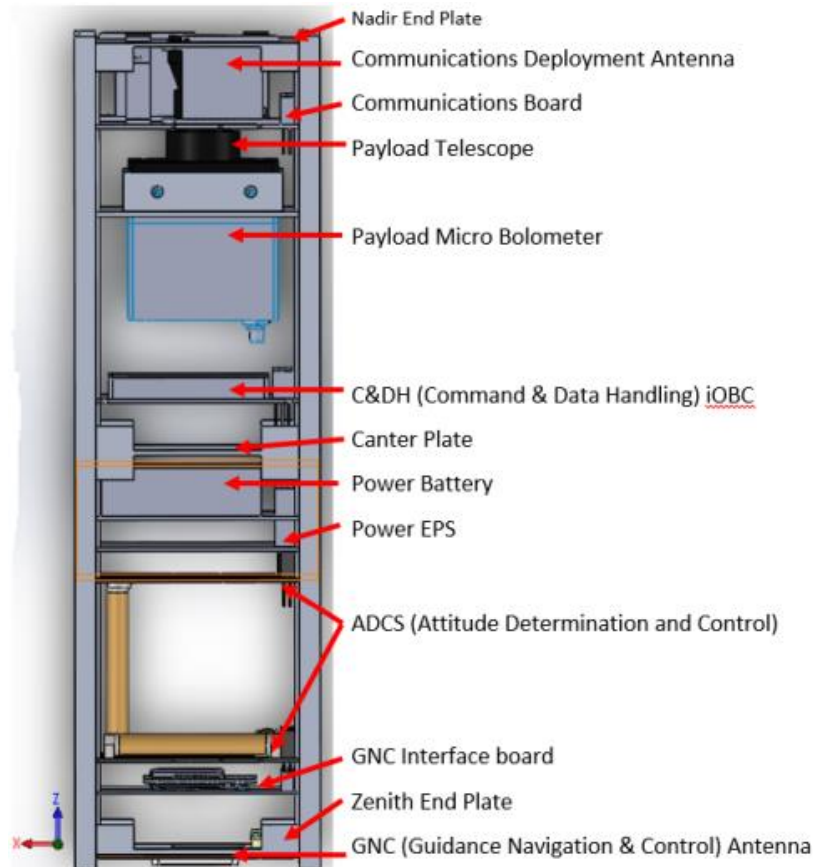


Figure 3-3: Structural depiction of CuSAT, where arrows point to subsystem boards [8]

When including hardware connections to facilitate data interfacing between the iOBC and other subsystem components, this layout must be considered to ensure connections are not impeded by any physical barrier in the satellite's structure.

3.3 Subsystem Data Interfacing

Given the Interface Spreadsheet compiled by the SIT team and previous years' reports detailing requirements from various subsystems, Table 3-2 below summarizes the data interface assignments for the OBC. The components belonging to each subsystem are assumed to reside on the subsystem boards themselves, as depicted in Figure 3-3 above. The exceptions are the reaction wheels and the temperature sensors. It should be noted that these components are also listed with (*), indicating

they have not yet been confirmed. The reaction wheels have yet to be ordered and the thermistors may be modified to be any temperature sensor. In addition, the number of temperature sensors required is also subject to change. The placement of these components is yet to be determined.

Table 3-2: CuSAT data interface assignments

Subsystem	Component	Interface	Pin Location	Pins
ADCS	Magnetometer	I ² C	CSKB	H1.41, H1.43
	Sun Sensors	I ² C	CSKB	H1.41, H1.43
	Magnetorquers	3xPWM	J2 pins	J2.75, 76, 80
		3xGPIO	J2 pins	J2.71, 72, 68
	Reaction Wheels*	3xI ² C	CSKB	H1.41, H1.43
GPS	GPS Receiver	UART	J2 pins	J2.87, 88
Communications	Transceiver	I ² C	CSKB	H1.21, H1.23
Payload	Camera	SPI	J2 pins	J2.19, 24, 27, 28
		GPIO	J2 pins	J2.31
	Interface Board	UART (RS232)	J2 pins	J2.91, 92
Power	EPS Board	I ² C	CSKB	H1.21, H1.23
	Power to iOBC	3.3V, GND	CSKB	H2.28, H2.30
Thermal	Thermistors*	ADC	J2 pins	J2.3,4,7,8,11,12,15,16

While peripheral interfacing through CSKB pins can be done easily, an issue with physical interfacing will arise with the J2 pins. Additional hardware is required to access the interfaces mapped to these pins. This will be addressed in the following section.

3.4 Integration of Physical Connections

In Figure 3-1, the layout of the iOBC shows the CSKB runs along one side of the board. Other subsystem boards will each be equipped with a PC/104 bus with required pins that line up to those labelled in the CSKB, per Figure 3-3. This will permit wiring to run along the z-axis (per the coordinate axis in Figure 3-3) between the PC/104 buses and the CSKB, where interfacing is necessary. For

subsystems requiring access to the J2 pins, the physical connections to these pins have not yet been devised, though several options have been presented.

The iOBC is equipped with a daughterboard that connects to the J2 pins and provides accessible ports for wiring to those interfaces. Currently, the in-house iOBC has an engineering-model (EM) daughterboard that can be used for testing, but as suggested by ISIS, the flight-model (FM) daughterboard should be used in flight. Figure 3-4 below shows the FM daughterboard mounted onto the iOBC, with its various ports labelled along its edges.



Figure 3-4: iOBC mounted with FM daughterboard. CSKB and pins from FM daughterboard are outlined in red [1]

The benefit of using the daughterboard is that it permits access to the J2 pins along the edges of the board. It also has a 1.03-cm gap just passed its J7 port to permit the inclusion of a wire harness that can connect to any of its pins from between itself and the iOBC. The issue with using this daughterboard is that its layout demands the use of a wire harness to travel along the length of the satellite to each of the subsystems with interfacing requirements to J2 pins. As the satellite is currently designed, each subsystem board takes up almost the entire xy-plane of the satellite at its height (per the coordinate system in Figure 3-3), allowing no additional room for a wire harness. To remedy this issue, possible solutions have been devised. They are listed below.

-
1. Realign the subsystem boards such that, per the coordinate system in the bottom left corner of Figure 2, the board faces would align with the xz-plane or the yz-plane. While this would provide sufficient space to run a wire harness for both the PC/104 standard bus as well as for the FM daughterboard pins, it may present a challenge when positioning the magnetorquers. Further, some boards may need to be stacked atop others; should wiring need to go between them, excessive bending and kinking in the wires may also occur.
 2. Run very small wire bundles along the sides of the satellite to their designated subsystem. This would allow the current structure to remain intact but the small bundles may be weak, and with the limited spacing they will likely be insufficiently insulated. If they are running behind the solar cells, the heat absorbed by the cells may damage the wires.
 3. Use Flat Flexible Cables as wire harnesses to all subsystems. These insulated cables required very little space and can adhere to the body of the satellite [9]. With this option, the current stacking landscape of the satellite will be preserved, and the insulation on the cables will protect the wires. However, these cables can be costly, and since they cannot be shifted sideways, design of their placements and connections must be fully tested so that upon implementation, their positioning will not need any modification. In addition, possible sources of friction and issues that may lead to insulation abrading should be investigated.
 4. Use jumper cables to assign interfaces from the FM daughterboard ports to the blank pins on the CSKB. This would require a realignment of interface boards as they are stacked in the satellite such that the iOBC is closer to the top of satellite. This mechanism would entirely remove the need for a wire harness as all interfaces would run through the PC/104 buses. To pursue this option, it would be necessary to ensure the blank pins do not have assignments on other subsystem PC/104 buses, and the electrical connections will not cause any short-circuiting or damage any hardware on the iOBC motherboard or FM daughterboard.

These solutions present some new ideas for realignment in the satellite as well as for procurement options for a wire harness. However, investigation from structural, budget and electrical standpoints should be conducted to determine a suitable physical model for interfacing. In addition,

any additional interfacing requirements for components that have yet to be added to the satellite must be considered to ensure complete satellite functionality.

4. OBC PROGRAMMING

Innovative Solutions In Space provides a Software Development Kit (SDK) with the iOBC which includes GCC ARM compiler, Atmel SAM-BA for code flashing, Eclipse IDE, FreeRTOS, and an ISIS hardware abstraction layer. For the development of the Satellite Design Project program, the provided toolchain is used.

The hardware abstraction layer interfaces between low level, hardware specific, code and high-level functions. It contains drivers for I/O buses, FRAM communication, timing, watchdog, and the supervisor interface. Using this layer, the programmers can rely upon tested protocols and functions built by the developers instead of creating new, and potentially error prone, drivers from scratch.

4.1 FreeRTOS

The tool used to schedule tasks, manage memory, and deal with inter process communication is FreeRTOS, a real-time operating system. It is a small kernel consisting of a minimum three source files.

FreeRTOS tasks have one of 4 states

Running – The task is currently being executed.

Ready – The task can be executed at any time. It is in the Ready queue.

Blocked – The task is waiting for an event to occur before it can be run.

Suspended – The task will not be run until it is explicitly taken out of the suspended state.

Upon creation, a task is given a priority and placed in the 'Ready' state. Periodically, the OS kernel will pause the currently executing task to search through all tasks in the 'Ready' queue and allow the highest priority task to run. While running, a task can enter the 'Blocked' state if it is asked to wait for an event, such as receive a message from another task or read an I/O port. Especially useful for periodic tasks, such as housekeeping, a task can block itself for a specific amount of time. When the event occurs, the task is again placed in the 'Ready' queue.

4.1.1 Task-Scheduling

The FreeRTOS task scheduler is responsible for deciding which task will be swapped in and when it will be swapped out.

FreeRTOS defines intervals of time called “time slices”. After every time slice interval, an interrupt is raised and the FreeRTOS kernel runs an interrupt service routine to decide which task should be run. There are three possible situations: the running task has a higher priority than all other ready tasks, the running task shares the highest priority level with another ready task, and there is a ready task with a higher priority than the running task.

1. If the running task has the highest priority out of all ready tasks, it will continue running.
2. If it has the same priority as another ready task, it will switch with that task. This causes each task to be entered in turn, but does not guarantee that they will receive equal CPU time.

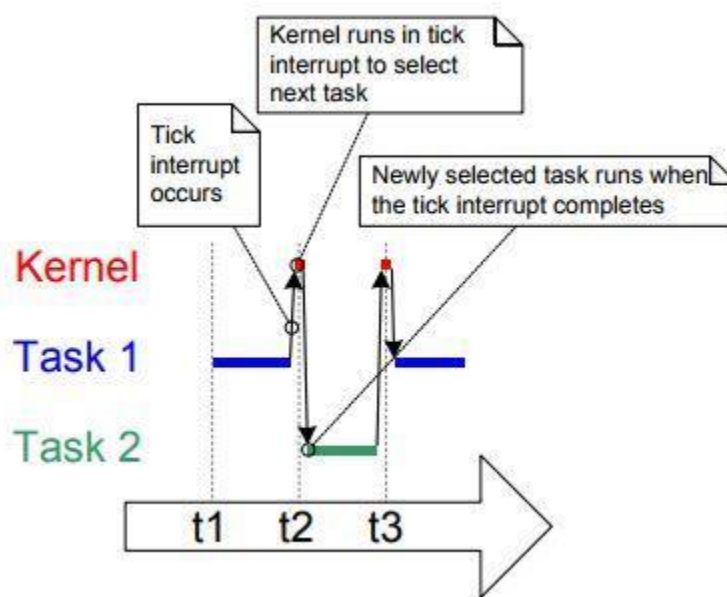


Figure 4-1: Time sharing between two tasks with equal priority

This figure shows an example of two tasks (Task1 and Task2) being scheduled over several time slices. At the start, Task1 is running. One time-slice later, an interrupt is raised and the kernel takes control to choose which task should be run. Since Task2 has the same priority as Task1 and Task1 has already run for one time slice, Task2 is swapped in and Task1 is swapped

out. Task2 runs for one time-slice before the kernel interrupt is processed and swaps Task2 out for Task1.

3. If the running task has a lower priority than a ready task, the running task is swapped out, and the highest priority task is swapped in.

A default task called the “Idle Task” is created by default for a low power idle which is also responsible for task deletion. This task has a priority of zero, meaning it can never starve other tasks and will only run when all higher priority tasks are blocked.

4.2 List of Tasks

The satellite flight code is split into the tasks listed below. Many of the tasks are described in section 2 of this document, but this section covers how they interact at a higher level and how they are scheduled.

Table 4-1: List of tasks

Task	Priority	Data	Responsibilities
Command Handling	High	Task Handles	Execute commands. ie. mode switching
Attitude Determination	Low	Sun vector, Magnetic vector	Calculate magnetorquer output
Housekeeping	Low	Housekeeping Packet	Gather telemetry data, form packet, and store it
GPS	Medium	GPS data	Read and compute GPS data
Payload	Medium	Images	Process raw image
Communication	High	Up/Downlink packets	Receive and transmit data to ground, build downlink package
Error Checking	Medium	Error data	Check that no data has been corrupted

4.2.1 Command Handling Task

This main controller task is given a high priority because it is responsible for initiating commands, and if a command is to be run at an exact time, it should not wait for any other task

before being executed. It is situated in the `obc.c` file found in Appendix 7.5.1. This task uses commands sent from the ground station to determine what must be done and when to execute them. Each command has a command code, an execution time, and a command number to remove repeatedly sent commands. The command code is an integer where each number has a unique command. Commands that are meant to be executed immediately will have an execution time tag of all 0's, and all other commands will have a specific execution time.

The FRAM stores a list of all tasks to be performed by the computer sorted in order of execution time. A pointer in FRAM keeps track of the next command to be executed.

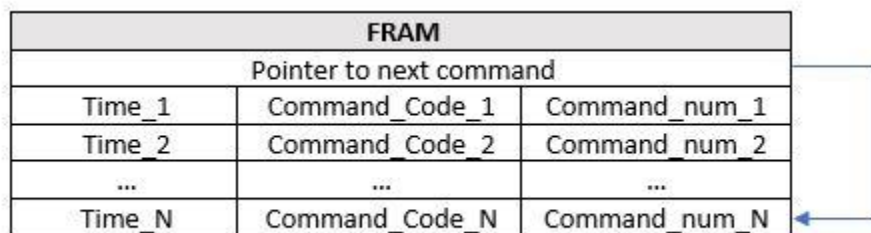


Figure 4-2: FRAM memory usage and Command storage

The Command Handling task reads the pointer to the next command, fetches that command, and checks the execution time. If it is the correct time to execute the command, it performs the necessary actions. If it is not yet the correct time, the task delays itself until the execution time, then checks again.

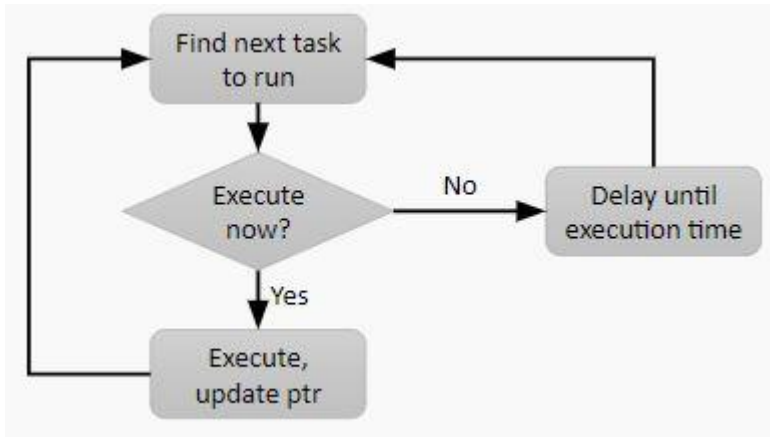


Figure 4-3: Command Handling task block diagram

This process will continue until all of the commands in the FRAM have been executed. In practice this should never occur because each day, the ground station will send seven days worth of commands to the satellite.

A function in the Command Handling task will perform mode switching. Modes will be represented by integers where each bit in the integer corresponds to a task. If the bit is '1', the task should be active, if the bit is '0', the task should be suspended in this mode. An enum will list the bit number for each task in the mode integer representation. As the mode is changed, it will send inter-process communication signals to the tasks to trigger other necessary changes. For example, a message will be sent to the Attitude Determination task instructing it to change its desired pointing vector.

After receiving an uplink transmission from ground station, the Communication task builds a list of all commands and sends it in a message to the Command Handling task. The Command Handling task combines it with the already existing list, removes duplicates, sorts it based on execution time, and places it in the FRAM.

4.2.2 Attitude Task

The Attitude task is responsible for retrieving data from the sun sensors and magnetometers, performing necessary calculations, and giving output signals to the magnetorquers. The skeleton code is found in Appendix 7.5.3. It has several modes that it can function in. Detumbling mode will perform

calculations to stabilize the satellite, sun mode will run the algorithm to align the spacecraft with the sun, nadir mode will point to the earth, and passive mode causes a passive rotation. External tasks can change the attitude mode by passing a message indicating the desired attitude mode change.

4.2.3 Housekeeping Task

This task gathers telemetry data across the satellite, compiles it into a housekeeping packet, and stores it on the SD card. Once completed, it blocks itself for 5 minutes then runs again.

4.2.4 GPS Task

The GPS task communicates with the GPS module to get the current time and position. It performs the necessary transformations to get the data in the desired format.

4.2.5 Payload Task

Once a raw image has been loaded into main memory, the Payload task processes it to find all pixels with values over a threshold. Once it receives a message to stop imaging, the processed images are then stored in the SD card.

4.2.6 Communication Task

This task is responsible for uplink and downlink of data. It collects and builds downlink packets by gathering all of the necessary data and adding a header and checksum. It processes uplink data and passes the new commands to the Command Handling task.

4.2.7 Error Checking Task

The Error Checking task will keep search for and keep track of all errors that have occurred. There will be an error structure for each task which it will populate with any errors it has encountered so far. This task collects those structures and stores them in FRAM as well as checks overall functionality, such as power levels, and memory and task status.

4.3 Timekeeping

All or nearly all of the data handled by the iOBC shall be time-tagged, including telemetry, payload data, and commands. The iOBC therefore is responsible for distributing timing information to tasks and ensuring accurate time values. This section discusses the timekeeping tools provided by the HAL and possible strategies for implementing the required functionality.

The iOBC has 2 redundant real time clocks, an external RTC connected through internal SPI (SPI0), and an internal RTT on the CPU. The external RTC compensates for temperature-induced clock-drift [1]. The RTC is power controlled by supervisor in order to prevent failures caused by power interruptions or power cycles of the iOBC. These connections are shown in figure 2. The HAL has three timing libraries, RTC.h, RTT.h, and Time.h, that will be used in CDH timekeeping.

Time.h is used to implement a redundant timekeeping system on the iOBC. In the event of an RTC failure, time is kept using the RTT. Time.h also provides a time structure in human readable format including seconds, minutes, etc. Two time set functions are provided:

```
Unsigned long Time_getUptimeSeconds();  
Int Time_getUnixEpoch(unsigned long *epochTime);
```

These functions can be used to modify clock values using either program uptime or time in seconds since Unix Epoch. Analogous time get functions are also provided that use the current RTT value as an offset. This strategy is used to prevent frequent communication with the relatively slow RTC. Because time.h implements a redundant timekeeping system, the functions provided in the library modify both RTC and RTT values. RTC.h and RTT.h can be used to modify the clocks individually.

To further ensure accurate timekeeping, the iOBC's RTC will periodically be synchronized to time output by the GPS receiver. This task shall be implemented with data protection measures in place to prevent errors in command execution or data loss.

4.4 Error Detection

Error detection will be used on the spacecraft to ensure data is accurate at all points in the information life cycle. For instance, the following pieces of data shall be checked periodically for errors: downlinked/uplinked data, intertask communications, and telemetry. Error detection can be computationally heavy, thus a multi-tiered approach should be taken to maximize the probability that error is detected and minimize the computational power used. Mission critical data such as ground commands should use a more robust error check such as CRC, while less critical data such as temperature points could use parity to save CPU time. The HAL provides routines for checksumming that will be tested for efficiency. Metrics to consider in making these design decisions include maximum weight, Fraction of errors, Special patterns, execution time, check sequence size [10].

Next steps in the implementation of error detection mechanisms include testing execution time vs. efficacy of the mechanisms mentioned in this section, categorization of critical vs. non-critical data, and research of error probability for different data interfaces/components. Also, note that no error correction is currently being considered.

4.5 Coding Standards

The set of C language coding standards developed by the Jet Propulsion Laboratory [11] shall be used to ensure high quality development of CuSAT's mission critical flight software. The JPL standards define good coding practices for real time embedded software, and can be divided into four categories: defensive coding, code clarity, language compliance, and predictable execution.

Appendix 7.4.1 contains a list of the JPL standards that will be used in the iOBC flight code. This list is a subset of the rules in the JPL document for current and future development of the flight code. There are two rules which should be highlighted and explained.

Rule 5 states "There shall be no use of dynamic memory allocation after task initialization." In a complex and dynamic system, such a task is not easy. For example, the size of processed images is highly variable and may result in an overflow of the available buffer, or a large amount of wasted space. Nevertheless, regular memory allocations and frees will fragment the memory, as well as

introduce potential memory access failures. Removing this source of error will increase reliability and allow for static diagnostic tools to perform better analysis.

Rule 8 states, “Data objects in shared memory shall have a single owning task. Only the owner of a data object shall modify the object.” Active adherence to this rule allows deterministic evaluation on modules, increases code modularity, and decreases the likelihood of critical section memory access violations. This rule shall be implemented by having a data structure be declared in its corresponding task source file. The data structure can be defined either in the corresponding source or header file. A task may deliver copies of its data structure to other tasks, but those tasks may not propagate the data.

5. FUTURE WORK

This section will discuss the items identified by the CuSAT team as future work moving into the next term. All existing subsystem integration tasks will continue to move forward, including Payload, ADCS, and GPS. Preliminary subsystem integration tasks, Communications, Power, and Thermal will enter the development stage. Future work for ADCS includes continuing to test converted ADCS algorithms and begin implementing test procedures for reading sensor data and sending commands to actuators. Also, it will be determined how telemetry data will be collected from the ADCS peripherals. The GPS software task will continue to be optimized, and integrate routines to be written by the ADCS team in order to validate program output and begin development of mission software. The CDH team will begin development of communications, power, thermal tasks based off the proposed software design covered in this report. Future work for interfacing requirements with subsystems involves testing the presented options for hardware interfacing with J2 pins. Also, it will be ensured that interfacing through CSKB will be reliable while in flight. Building off of the task scheduling system developed this term, inter-process communications lines will be coded, the command handling task will be completed, and shell tasks will be programmed. Additionally, error checking and timekeeping tasks will be developed and a model will be built for memory management.

6. REFERENCES

- [1 H. Péter-Contesse and A. Piplani, "ISIS-OBC Datasheet," Innovative Solutions in Space, Delft, Netherlands, 2014.
- [2 K. Stadnyk, G. Sévigny and S. Ulrich, "Attitude Determination & Control Subsystem Report," Carleton University Department of Mechanical & Aerospace Engineering, Ottawa, 2017.
- [3 STMicroelectronics NV, "LIS3MDL: Digital output magnetic sensor," 2017. [Online]. Available: <http://www.st.com/content/ccc/resource/technical/document/datasheet/54/2a/85/76/e3/97/42/18/DM00075867.pdf/files/DM00075867.pdf/jcr:content/translations/en.DM00075867.pdf>.
- [4 M. Feuerherm and T. Vainio, "Command and Data Handling Subsystem Final Report," Carleton University Department of Mechanical & Aerospace Engineering, Ottawa, 2017.
- [5 C. D. Hall, "Attitude Determination," in *Spacecraft Attitude Dynamics and Control*, Blacksburg, Virginia Tech, 2003, pp. 4.1-4.23.
- [6 SkyFox Labs, "CubeSat GPS Receiver/Next Generation piNAV-NG Product Datasheet," 2016.
- [7 E. & W. H. Smal, "CarletonCuSAT-1 Design Project: Guidance and Navigation Subsystem Final Report," Carleton University, 2017.
- [8 A. Bassi, M. Bettez, J. Nicola and B. Burlton, "Systems Preliminary Design Report," Carleton University, Ottawa, 2017.
- [9 P. Karuza, G. Maul and D. Hinkley, "Flat Flexible Cables (FFC) in Picosatellites," 11 August 2012. [Online]. Available: http://mstl.atl.calpoly.edu/~bklofas/Presentations/SummerWorkshop2012/Karuza_Flat_Flexible_Cables.pdf. [Accessed 2017].
- [1 P. Koopman, "Checksum and CRC Data Integrity Techniques for Aviation," Honeywell Laboratories, 0] 2012.

[1 "JPL Institutional Coding Standard for the C Programming Language," Jet Propulsion Laboratory, 1] California Institute of Technology, 2009.

7. APPENDICES

7.1 Payload

This section of the Appendix provides the code used by the Raspberry Pi to threshold greyscale images.

```
import cv2
import numpy as np

img = cv2.imread('C:\\Users\\bb\\Desktop\\globe2.jpg',cv2.IMREAD_GRAYSCALE)
cv2.imshow('image',img)
cv2.waitKey(0)

#Set Threshold Value
Threshold=150
binary= cv2.threshold(img, Threshold , 255, cv2.THRESH_BINARY);
plt.imshow(binary,'grey')

# Grab Image Dimensions
h = img.shape[0]
w = img.shape[1]

# loop over the image, pixel by pixel
for y in range(0, h):
    for x in range(0, w):

        # threshold the pixel
        print( x,y) if img[y, x] >= Threshold else 0
```

Figure 1: Thresholding code

7.2 ADCS

This section of the Appendix contains the MATLAB to C Conversion Document and the generated files when implementing the MATLAB Coder on the TRIAD algorithm.

7.2.1 MATLAB to C Conversion Document

1. Create a function in MATLAB. Remove any anonymous functions within it (i.e. create separate function files or re-write the entire function every time it is called).

```
function [ C_BI ] = TRIAD (sb,mb,si,mi)
%TRIAD_test function performs the TRIAD algorithm for attitude
%determination given sensor and propagator data

%Inputs:
%sb - Sun vector in body fixed reference frame
%mb - Magnetic field vector in body fixed reference frame
%si - Sun vector in ECI
%mi - Magnetic field vector in ECI

%Outputs:
%C_BI - The attitude matrix/rotation matrix between ECI and body fixed

%Comments:
% Pat Roussio, Carleton University
% June, 2017
|
%Skew Matrix
%skew = @(A) [0 -A(3) A(2) ; A(3) 0 -A(1) ; -A(2) A(1) 0];

%TRIAD Algorithm
%Triad vector in Body Fixed reference frame
t1_b = sb;
t2_b = (skew(sb)*mb)/(norm(skew(sb)*mb));
t3_b = (skew(t1_b)*t2_b)/(norm(skew(t1_b)*t2_b));

%Triad vector in ECI reference frame
t1_i = si;
t2_i = (skew(si)*mi)/(norm(skew(si)*mi));
t3_i = (skew(t1_i)*t2_i)/(norm(skew(t1_i)*t2_i));

%Defining TRIAD vectors in Body Fixed and ECI reference frames
tb = [t1_b t2_b t3_b];

ti = [t1_i t2_i t3_i];

%Attitude Matrix C_BI = C_bt * C_ti
C_BI = tb * ti';

end
```

Figure 1: skew is a separate function file (not shown).

2. Create a script the calls the function to be converted. Ensure all files are in the same folder.

```
% Main test script to call TRIAD function
% Inputs to function:
v1b = [0.2673; 0.5345; 0.8018];
v2b = [-0.3124; 0.9370; 0.1562];
v1i = [0.7814; 0.3751; 0.4987];
v2i = [0.6163; 0.7075; -0.3459];

rot = TRIAD(v1i, v2i, v1b, v2b);
% rotation matrix should be:
% 0.5662 0.7803 0.2657
% -0.7881 0.4180 0.4518
% 0.2415 -0.4652 0.8516
```

Figure 2: testTRIAD.m script file.

3. Go to Apps → MATLAB Coder. Next to the line stating *Generate code for function*: click the Browse button and select function to be converted to C code. Figure 3 shows the window after selecting the function. Do not change Numeric Conversion. Click Next.

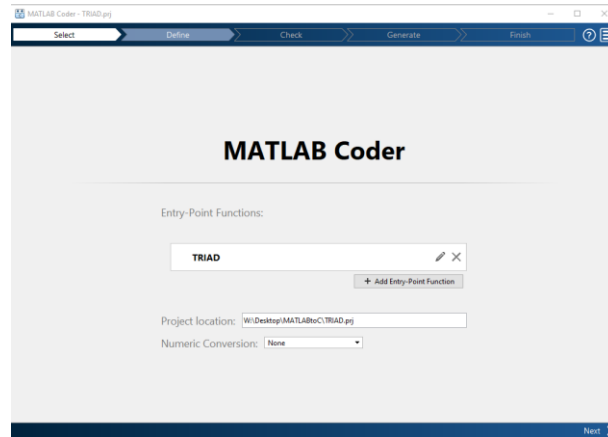


Figure 3: Selected triad.m file for conversion.

- To define input types, select “Let me enter input of global types directly”. See Figure 4.

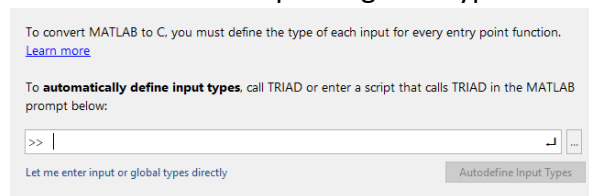


Figure 4: Choose “Let me enter input of global types directly”.

Inputs should appear as a list, per Figure 5. For each input, select “Click to define”. Choose the appropriate type. For example, a 3-element vector should be defined as *double(m x 1)*, where *m=3*. Complete this step for all inputs to the function. Click Next.

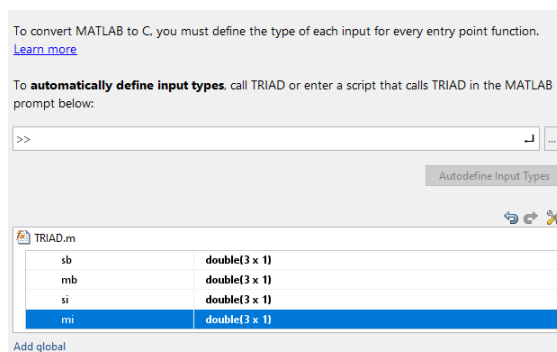


Figure 5: Defining input types.

- To check for run-time issues, select the script that calls the function to be converted. Then choose *Check for Issues*. A new folder, *codegen* will open in the folder containing your MATLAB

function. If your MATLAB function and test script can be suitably converted to C code, no issues should be detected, per Figure 6. Click Next.

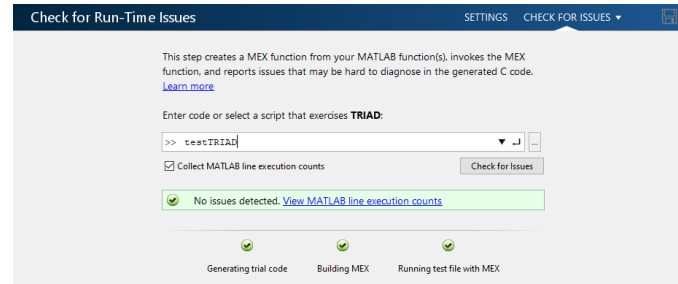


Figure 6: Check for Run-Time Issues.

6. To generate C code, change the settings such that they conform to those of the device to which the code is to be written. For the iOBC, Figure 7 shows proper configuration. The device is an ARM9 processor.

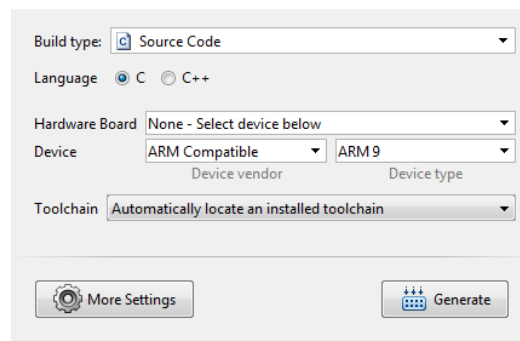


Figure 7: Settings to generate C code for iOBC.

If the option for the Hardware Board does not allow for the specific Device selection, select the down arrow for this Setting and choose “Get Support Package for ARM Cortex-M Processors...”. This will open the MATLAB Support Package Installer, depicted in Figure 8 below. Note that in this Figure, the support package selected is for the ARM Cortex-A processors. Ensure that the package to be installed is for ARM Cortex-M processors.

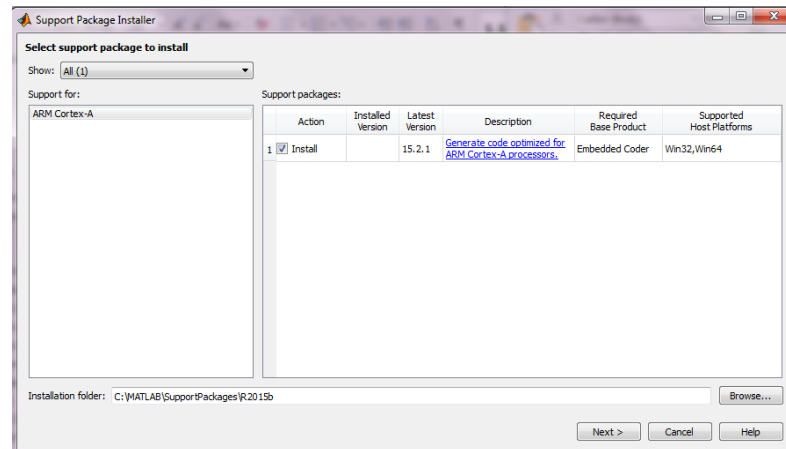


Figure 8: Support Package Installer

Alternatively, the package can be downloaded and installed via the MathWorks website, by going to <https://www.mathworks.com/hardware-support/arm-cortex-m.html> and selecting “Get Support Package”, as seen in Figure 9.

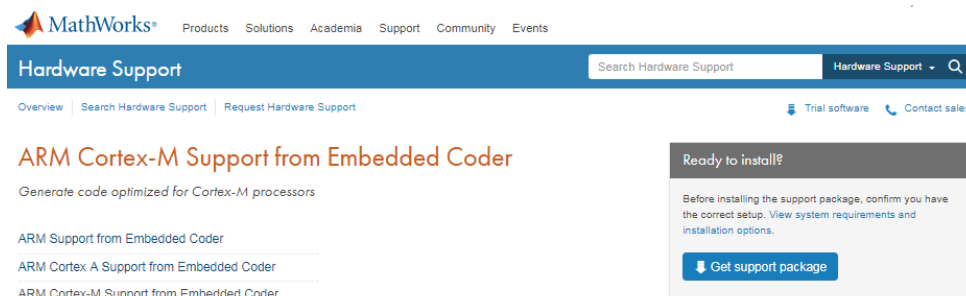


Figure 9: Online access to hardware support package.

- Once the settings have been configured as shown in Figure 7, Select Generate. Once completed, the generated code will be saved in a folder called *codegen*, and a Code Generation Report will provide additional details for any errors or warnings that may have appeared during code generation. The final screen should show the file path to the created C files, as in Figure 10.

Generated Output




C Code	 C:\Users\Zeena\Documents\BEng\MAAE 4907\devboard\matlab\matlab\codegen\lib\TRIAD
Example main Files	 ...Zeena\Documents\BEng\MAAE 4907\devboard\matlab\matlab\codegen\lib\TRIAD\examples
Reports	 Code Generation Report

Figure 10: File path to generated code and link to Code Generation Report.

7.2.2 TRIAD MATLAB Files

TRIAD algorithm function:

```
function [ C_BI ] = TRIAD (sb,mb,si,mi)
%TRIAD_test function performs the TRIAD algorithm for attitude
%determination given sensor and propagator data

%Inputs:
%sb - Sun vector in body fixed reference frame
%mb - Magnetic field vector in body fixed reference frame
%si - Sun vector in ECI
%mi - Magnetic field vector in ECI

%Outputs:
%C_BI - The attitude matrix/rotation matrix between ECI and body fixed

%Comments:
% Pat Rouso, Carleton University
% June, 2017

%TRIAD Algorithm
%Triad vector in Body Fixed reference frame
t1_b = sb;
t2_b = (skew(sb)*mb)/(norm(skew(sb)*mb));
t3_b = (skew(t1_b)*t2_b)/(norm(skew(t1_b)*t2_b));

%Triad vector in ECI reference frame
t1_i = si;
t2_i = (skew(si)*mi)/(norm(skew(si)*mi));
t3_i = (skew(t1_i)*t2_i)/(norm(skew(t1_i)*t2_i));

%Defining TRIAD vectors in Body Fixed and ECI reference frames
tb = [t1_b t2_b t3_b];
ti = [t1_i t2_i t3_i];

%Attitude Matrix C_BI = C_bt * C_ti
C_BI = tb * ti' ;

end
```

Skew Function (called in TRIAD)

```
function [skew_mtrx] = skew (A)
%Function returns the skew matrix of a 3x1 matrix
%Inputs:
%A - 3x1 matrix

%Outputs:
%skew_mtrx - the skew matrix of A

skew_mtrx = [0 -A(3) A(2) ; A(3) 0 -A(1) ; -A(2) A(1) 0];
end
```

7.2.3 Relevant C Files Generated by MATLAB Coder

The programs below include the TRIAD and main source codes. Additional source and header files that were generated by the MATLAB Coder can be found on the SVN.

TRIAD Code:

```
/* File: TRIAD.c
 *
 * MATLAB Coder version      : 3.3
 * C/C++ source code generated on : 26-Oct-2017 11:08:52
 */

/* Include Files */
#include "rt_nonfinite.h"
#include "TRIAD.h"

/* Function Definitions */

/*
 * TRIAD_test function performs the TRIAD algorithm for attitude
 * determination given sensor and propagator data
 * Arguments      : const float sb[3]
 *                  const float mb[3]
 *                  const float si[3]
 *                  const float mi[3]
 *                  float C_BI[9]
 * Return Type    : void
 */
void TRIAD(float sb[3], float mb[3], float si[3], float
           mi[3], float C_BI[9])
{
    float b_sb[9];
    float b_si[9];
    int i0;
    float s_cross_m_b[3];
```

```
float s_cross_m_i[3];
float x;
int i1;
float t1_cross_t2_b[3];
float b_x;
float t1_cross_t2_i[3];
int i2;

/* Inputs: */
/* sb - Sun vector in body fixed reference frame */
/* mb - Magnetic field vector in body fixed reference frame */
/* si - Sun vector in ECI */
/* mi - Magnetic field vector in ECI */
/* Outputs: */
/* C_BI - The attitude matrix/rotation matrix between ECI and body fixed */
/* Comments: */
/* Pat Rouso, Carleton University */
/* June, 2017 */
/* Skew Matrix */
/* skew = @(A) [0 -A(3) A(2) ; A(3) 0 -A(1) ; -A(2) A(1) 0]; */
/* Defining variable for cross product between s and m: */
/* Function returns the skew matrix of a 3x1 matrix */
/* Inputs: */
/* A - 3x1 matrix */
/* Outputs: */
/* skew_mtrx - the skew matrix of A */

b_sb[0] = 0.0F;
b_sb[3] = -sb[2];
b_sb[6] = sb[1];
b_sb[1] = sb[2];
b_sb[4] = 0.0F;
b_sb[7] = -sb[0];
b_sb[2] = -sb[1];
b_sb[5] = sb[0];
b_sb[8] = 0.0F;

/* Function returns the skew matrix of a 3x1 matrix */
/* Inputs: */
/* A - 3x1 matrix */
/* Outputs: */
/* skew_mtrx - the skew matrix of A */
b_si[0] = 0.0F;
b_si[3] = -si[2];
b_si[6] = si[1];
b_si[1] = si[2];
b_si[4] = 0.0F;
b_si[7] = -si[0];
b_si[2] = -si[1];
b_si[5] = si[0];
b_si[8] = 0.0F;
for (i0 = 0; i0 < 3; i0++) {
    s_cross_m_b[i0] = 0.0F;
```

```
s_cross_m_i[i0] = 0.0F;
for (i1 = 0; i1 < 3; i1++) {
    s_cross_m_b[i0] += b_sb[i0 + 3 * i1] * mb[i1];
    s_cross_m_i[i0] += b_si[i0 + 3 * i1] * mi[i1];
}
}

/* TRIAD Algorithm */
/* Triad vector in Body Fixed reference frame */
x = (float)sqrt((s_cross_m_b[0] * s_cross_m_b[0] + s_cross_m_b[1] *
    s_cross_m_b[1]) + s_cross_m_b[2] * s_cross_m_b[2]);
for (i0 = 0; i0 < 3; i0++) {
    s_cross_m_b[i0] /= x;
}

/* Defining variable for cross product between t1b and t2b: */
/* Function returns the skew matrix of a 3x1 matrix */
/* Inputs: */
/* A - 3x1 matrix */
/* Outputs: */
/* skew_mtrx - the skew matrix of A */
b_sb[0] = 0.0F;
b_sb[3] = -sb[2];
b_sb[6] = sb[1];
b_sb[1] = sb[2];
b_sb[4] = 0.0F;
b_sb[7] = -sb[0];
b_sb[2] = -sb[1];
b_sb[5] = sb[0];
b_sb[8] = 0.0F;
for (i0 = 0; i0 < 3; i0++) {
    t1_cross_t2_b[i0] = 0.0F;
    for (i1 = 0; i1 < 3; i1++) {
        t1_cross_t2_b[i0] += b_sb[i0 + 3 * i1] * s_cross_m_b[i1];
    }
}

x = (float)sqrt((t1_cross_t2_b[0] * t1_cross_t2_b[0] + t1_cross_t2_b[1] *
    t1_cross_t2_b[1]) + t1_cross_t2_b[2] * t1_cross_t2_b[2]);

/* Triad vector in ECI reference frame */
b_x = (float)sqrt((s_cross_m_i[0] * s_cross_m_i[0] + s_cross_m_i[1] *
    s_cross_m_i[1]) + s_cross_m_i[2] * s_cross_m_i[2]);
for (i0 = 0; i0 < 3; i0++) {
    s_cross_m_i[i0] /= b_x;
}

/* Defining variable for cross product between t1i and t2i: */
/* Function returns the skew matrix of a 3x1 matrix */
/* Inputs: */
/* A - 3x1 matrix */
/* Outputs: */
/* skew_mtrx - the skew matrix of A */
```

```

b_sb[0] = 0.0F;
b_sb[3] = -si[2];
b_sb[6] = si[1];
b_sb[1] = si[2];
b_sb[4] = 0.0F;
b_sb[7] = -si[0];
b_sb[2] = -si[1];
b_sb[5] = si[0];
b_sb[8] = 0.0F;
for (i0 = 0; i0 < 3; i0++) {
    t1_cross_t2_i[i0] = 0.0F;
    for (i1 = 0; i1 < 3; i1++) {
        t1_cross_t2_i[i0] += b_sb[i0 + 3 * i1] * s_cross_m_i[i1];
    }
}

b_x = (float)sqrt((t1_cross_t2_i[0] * t1_cross_t2_i[0] + t1_cross_t2_i[1] *
    t1_cross_t2_i[1]) + t1_cross_t2_i[2] * t1_cross_t2_i[2]);

/* Defining TRIAD vectors in Body Fixed and ECI reference frames */
/* Attitude Matrix C_BI = C_bt * C_ti */
for (i0 = 0; i0 < 3; i0++) {
    b_sb[i0] = sb[i0];
    b_sb[3 + i0] = s_cross_m_b[i0];
    b_sb[6 + i0] = t1_cross_t2_b[i0] / x;
    b_si[3 * i0] = si[i0];
    b_si[1 + 3 * i0] = s_cross_m_i[i0];
    b_si[2 + 3 * i0] = t1_cross_t2_i[i0] / b_x;
}

for (i0 = 0; i0 < 3; i0++) {
    for (i1 = 0; i1 < 3; i1++) {
        C_BI[i0 + 3 * i1] = 0.0F;
        for (i2 = 0; i2 < 3; i2++) {
            C_BI[i0 + 3 * i1] += b_sb[i0 + 3 * i2] * b_si[i2 + 3 * i1];
            //printf("%d: %f ", i0+3*i1, C_BI[i0 + 3 * i1]);
        }
    }
}
}

/*
 * File trailer for TRIAD.c
 *
 * [EOF]
 */

```

MAIN PROGRAM:

```

/ * File: main.c
*
* MATLAB Coder version      : 3.3

```

```
* C/C++ source code generated on   : 26-Oct-2017 11:08:52
*/

                                                                    */
/*****
/* Include Files */
#include "rt_nonfinite.h"
#include "TRIAD.h"
#include "main.h"
#include "TRIAD_terminate.h"
#include "TRIAD_initialize.h"
#include <stdio.h>
#include <windows.h>

/* Function Declarations */
static void argInit_3x1_real32_T(float result[3]);
static float argInit_real32_T(void);
static void main_TRIAD(void);

/* Function Definitions */

/*
 * Arguments      : float result[3]
 * Return Type    : void
 */
static void argInit_3x1_real32_T(float result[3])
{
    int idx0;

    /* Loop over the array to initialize each element. */
    for (idx0 = 0; idx0 < 3; idx0++) {
        /* Set the value of the array element.
           Change this value to the value that the application requires. */
        result[idx0] = argInit_real32_T();
    }
}

/*
 * Arguments      : void
 * Return Type    : float
 */
static float argInit_real32_T(void)
{
    return 0.0F;
}

/*
 * Arguments      : void
 * Return Type    : void
 */
static void main_TRIAD(void)
{
    /* Initialize function 'TRIAD' input arguments. */
    /* Initialize function input argument 'sb'. */
```

```
/* Initialize function input argument 'mb'. */
/* Initialize function input argument 'si'. */
/* Initialize function input argument 'mi'. */
float fv0[3] = {0.7814, 0.3751, 0.4987};
float fv1[3] = {0.6163, 0.7075, -0.3459};
float fv2[3] = {0.2673, 0.5345, 0.8018 };
float fv3[3] = {-0.3124, 0.9370, 0.1562};
float C_BI[9];

/* Call the entry-point 'TRIAD'. */

TRIAD(fv0, fv1, fv2, fv3, C_BI);
printf("For inputs: \nmb = ");
int i;
for (i = 0; i < 3; i++) {
    printf("%f ", fv0[i]);
}
printf("\nmb = ");
for (i = 0; i < 3; i++) {
    printf("%f ", fv1[i]);
}
printf("\nsi = ");
for (i = 0; i < 3; i++) {
    printf("%f ", fv2[i]);
}
printf("\nmi = ");
for (i = 0; i < 3; i++) {
    printf("%f ", fv3[i]);
}
printf("\n\nRotation Matrix: \n");
for (i = 0; i < 9; i++) {
    if (i == 3 || i==6) {
        printf("\n");
    }
    printf("%f ", C_BI[i]);
}
}

/*
 * Arguments      : int argc
 *                  const char * const argv[]
 * Return Type    : int
 */
int main(int argc, const char * const argv[])
{
    (void)argc;
    (void)argv;

    /* Initialize the application.
     * You do not need to do this more than one time. */
    TRIAD_initialize();

    /* Invoke the entry-point functions.
```

```
    You can call entry-point functions multiple times. */
main_TRIAD();

/* Terminate the application.
   You do not need to do this more than one time. */
TRIAD_terminate();

Sleep(100000);
return 0;
}

/*
 * File trailer for main.c
 *
 * [EOF]
 */
```

7.3 GPS

7.3.1 Main.c

```
//
//  main.c
//  gps
//
//  Created by Isabelle Kosteniuk on 2017-11-05.
//  Copyright © 2017 Isabelle Kosteniuk. All rights reserved.
//

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>

int createSentenceArray(char *sentenceBuffer, char *sentenceArray[], int
sentenceSize);
int createVector(int startindex, char *sentenceArray[], double *vector);
int convertToECI(double theta, double *ECEF, double *ECI);
double convertToSeconds(double time);
```

```
int main(){

    // declare all arrays here to avoid malloc use in functions

    //data read variables

    FILE *fp;
    uint32_t c;
    uint8_t i = 0;
    uint8_t j = 0;
    uint8_t sentenceSize = 80;
    uint8_t sentenceNumber = 12;
    char sentenceBuffer[sentenceNumber][sentenceSize];

    //data parse variables

    char* lsv_array[9] = {0};
    char* lsp_array[8] = {0};
    char* gga_array[11] = {0};

    //convert to eci variables

    uint16_t index = 0;
    double w_earth = 7.29115e-5;
    double timeUTC;
    double secondsUTC;
    double pos[3];
    double vel[3];
    double posECI[3];
    double velECI[3];

    // read from data bus - in this case, a text file
    // this could be converted to a function, also algo probably needs
improvement

    fp = fopen("/Users/isabellekosteniuk/Documents/SDP
2017/GPS/testdata.txt", "r");

    if(fp==NULL){
        perror("error");
        return(-1);
    }

    while( !feof(fp)){ // need a statically determinable condition here.
wait a certain time?

        c = fgetc(fp);
```

```
    if (c == '$') {
        c = fgetc(fp);
        if (c == 'P') {
            c = fgetc(fp);
            if (c == 'S') {
                c = fgetc(fp);
                if (c == 'L') {
                    c = fgetc(fp);
                    if (c == 'S') {
                        c = fgetc(fp);
                        if (c == 'P') {
                            for (i=0; i < sentenceNumber; i++){
                                for (j=0; j<sentenceSize; j++) {
                                    c = fgetc(fp);
                                    if (c == '\n'){j = sentenceSize;}
                                    sentenceBuffer[i][j] = c;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

fclose(fp);

// Populate Arrays containing GGA, LSP, and LSV Sentences

createSentenceArray(sentenceBuffer[3], gga_array, sentenceSize);
createSentenceArray(sentenceBuffer[0], lsp_array, sentenceSize);
createSentenceArray(sentenceBuffer[1], lsv_array, sentenceSize);

timeUTC = atof(gga_array[1]);
secondsUTC = convertToSeconds(timeUTC);
printf("total seconds: %f", secondsUTC);

// Populate Vectors containing Position and Velocity values

createVector(2, lsp_array, pos);
createVector(3, lsv_array, vel);

double theta = w_earth*secondsUTC;

convertToECI(theta, pos, posECI);
convertToECI(theta, vel, velECI);
```

```
// display some output for test purposes

for(index = 0; index<3;index++){
    printf("\n position: %f", pos[index]);
}

for(index = 0; index<3;index++){
    printf("\n velocity: %f", vel[index]);
}

printf("\n time: %f \n X eci: %f \n Y eci: %f \n Z eci: %f \n",
timeUTC, posECI[0], posECI[1], posECI[2]);

    exit(0);
}

int createSentenceArray(char *sentenceBuffer, char *sentenceArray[], int
sentenceSize){

    uint8_t index = 0;
    const char* delim = ",*";

    sentenceArray[index] = strtok(sentenceBuffer, delim);

    while (sentenceArray[index] != NULL || index < sentenceSize) {

        index++;
        sentenceArray[index] = strtok(NULL, delim);

    }

    return 0;
}

int createVector(int startindex, char *sentenceArray[], double *vector){

    vector[0] = atof(sentenceArray[startindex]);
    vector[1] = atof(sentenceArray[startindex +1]);
    vector[2] = atof(sentenceArray[startindex +2]);

    return 0;
}

int convertToECI(double theta, double *ECEF, double *ECI){

    ECI[0] = ECEF[0]*cos(theta) + ECEF[1]*sin(theta);
```

```
    ECI[1] = -ECEF[0]*sin(theta) + ECEF[1]*cos(theta);
    ECI[2] = ECEF[2];

    return 0;
}

double convertToSeconds(double time){

    double timeHrs;
    double timeMins;
    double seconds;
    double timeSecs;

    timeHrs = floor(time/10000);
    timeMins = floor((time - 10000*timeHrs)/100);
    timeSecs = time - 10000*timeHrs - 100*timeMins;
    seconds = 3600*timeHrs + 60*timeMins + timeSecs;

    return seconds;
}
```

7.3.2 Testdata.txt

```
$GPGSV,4,1,15,12,54,56,45,21,40,284,45,27,5,13,00,18,20,346,45*49
$GPGSV,4,2,15,15,8,31,45,30,8,233,00,02,6,139,00,25,72,186,45*7E
$GPGSV,4,3,15,05,29,122,45,,,,,09,12,5,45,,,,*7A
$GPGSV,4,4,15,,,,,,,,,,,,,*7D
$PSLSP,3065.1001232,801,-6596649.995,79478.615,-1744310.844,7,2.6*7E
$PSLSV,3065.1001232,801,1451.437,-4428.612,-5626.881,7,2.6*49
$PSLSS,3.30,1.20,3.29,45,9,22*7B
$GPGGA,005055.100,1453.9667,S,17918.5828,E,1,7,1.4,447101.6,M,0,M,,, *48
$GPGSA,M,3,12,21,18,15,25,05,09,,,,,2.6,1.4,2.1*3A
$GPGLL,1453.9667,S,17918.5828,E,005055.100,A,*0D
$GPRMC,005055.100,A,1453.9667,S,17918.5828,E,14202.14,142.86,281214,,, *3F
$GPVTG,142.86,T,,M,14202.14,N,26302.43,K,A,*1A
$GPGSV,4,1,15,12,54,56,45,21,40,284,45,27,5,13,00,18,20,346,45*49
$GPGSV,4,2,15,15,8,31,45,30,8,233,00,02,6,139,00,25,72,186,45*7E
$GPGSV,4,3,15,05,29,122,45,,,,,09,12,5,45,,,,*7A
$GPGSV,4,4,15,,,,,,,,,,,,,*7D
$GPGSV,4,2,15,15,8,31,45,30,8,233,00,02,6,139,00,25,72,186,45*7E
```

7.3.3 Output

```
position: -6596649.995000
position: 79478.615000
position: -1744310.844000
velocity: 1451.437000
velocity: -4428.612000
velocity: -5626.881000
time: 5055.100000
X eci: -6124995.316235
Y eci: 2450824.429223
Z eci: -1744310.844000
Program ended with exit code: 0
```

7.4 Programming

7.4.1 JPL Coding Standards

Rule #	Details
1	All C code shall conform to ISO/IEC 9899-1999(E) standard with no reliance on undefined or unspecified behaviour.
2	All code shall be compiled with warnings enabled at highest warning level available with no errors or warnings resulting.
3	All loops shall have a statically determinable upper-bound on the maximum number of loop iterations.
4	There shall be no recursive function calls.
5	There shall be no use of dynamic memory allocation after task initialization.
7	Task synchronization shall not be performed through the use of task delays.
8	Data objects in shared memory shall have a single owning task. Only the owner of a data object shall modify the object.
11	The goto statement shall not be used.
12	In an enumerator list, the '=' construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
13	Data objects shall be declared at the smallest possible level of scope. No declaration in an inner scope shall hide a declaration in an outer scope.
14	The return value of non-void functions shall be checked or used by each calling function.

15	The validity of function parameters shall be checked at the start of each public function. The validity of function parameters to other functions shall be checked by either the function called or by the calling function.
16	Assertions shall be used to perform basic sanity checks throughout the code.
20	Use of C preprocessor shall be limited to file inclusion and simple macros.
21	Macros shall not be #define'd within a function or block.
22	#undef shall not be used
23	All #else, #elif, and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

7.4.2 Matrix Operations

Several mission software tasks will require matrix operations. The standard C math library does not contain routines for performing these operations, so a third-party library was sourced. The GNU Scientific Library (GSL) supports CBLAS, a C language interface for BLAS, a set of low level linear algebra routines. Tests were written to determine the performance of routines supported by the GSL. Figure 8 below shows the average test results for various functions. The CBLAS routines outperformed the BLAS and gsl_matrix functions, but require a more advanced implementation. Any matrix operations required in the iOBC software shall make use of the GSL CBLAS routines.

test	function tested	library dependencies	clock ticks	speed (s)
allocation	gsl_matri_alloc	gsl matrix	33.3	0.000033
initialization	gsl_matrix_set_all	gsl matrix	2.3	0.000002
access element	gsl_matrix_get	gsl matrix	6.0	0.000006
set element	gsl_matrix_set	gsl matrix	1.3	0.000001
element multiplication	gsl_matrix_mul_elements	gsl matrix	7.7	0.000008
MxM transpose	gsl_matrix_transpose	gsl matrix	4.0	0.000004
MxN transpose	gsl_matrix_transpose_memcpy	gsl matrix	1.7	0.000002
BLAS matrix initialization	std c, gsl_matrix_view_array	gsl matrix	6.0	0.000006
BLAS DGEMM	gsl_blas_dgemm	gsl matrix, gsl blas	33.0	0.000033
CBLAS matrix initialization	std c	no dependencies	0.7	0.000001
CBLAS DGEMM	cblas_dgemm	gsl cblas	3.0	0.000003

Results of GSL Matrix Operation Tests

7.5 iOBC files

Not all of the files are included in this section of the appendix. It is meant to give an idea of how the overall code is organized and show task control.

7.5.1 obc.c

```
/*
 * obc.c
 *
 * Created on: Oct 6, 2017
 * Author: Nathan Lim
 */

#include <hal/Utility/util.h>

#include <FreeRTOS/FreeRTOS.h>
#include <FreeRTOS/task.h>

#include "housekeep.h"
#include "adc.h"
#include "obc.h"
#include "gps.h"
#include "fram.h"

#define HIGH_PRIORITY          configMAX_PRIORITIES-1
#define MEDIUM_PRIORITY      configMAX_PRIORITIES-2
#define LOW_PRIORITY          configMAX_PRIORITIES-4

/* List of Priorities for all tasks on the system */
#define HOUSEKEEP_PRIORITY      MEDIUM_PRIORITY
#define ERROR_PRIORITY          MEDIUM_PRIORITY          // check and handle
errors
#define ATTITUDE_PRIORITY      LOW_PRIORITY              // do attitude
determination and control. Triad, Detumbling, rebasing
#define GPS_PRIORITY            MEDIUM_PRIORITY
#define PAYLOAD_PRIORITY        MEDIUM_PRIORITY          // process payload data
#define COMMUNICATION_PRIORITY MEDIUM_PRIORITY          // Deal with data uplink and
downlink
#define COMMAND_H_PRIORITY      MEDIUM_PRIORITY          // command timing setup

/* Command Handle task data */
struct obc obc;

/* List of tasks on the system that can be turned on and off. */
enum task {
    // gather and store sensor data
    HOUSEKEEP,          // 0
```

```
// calculate attitude controls
ATTITUDE,          // 1
// gather GPS data
GPS,               // 2
// process payload data
PAYLOAD,          // 3
// transmit communication data
COMMUNICATION,     // 4
// handle communication commands
COMMAND_H_HANDLE, // 5
// check for errors
ERROR,            // 6
NUM_TASKS,

};

/* List of task handles. The entries match up with enum task. */
xTaskHandle task[NUM_TASKSI];

/*
 * Matrix for determining which tasks should be active during each mode.
 * The index of the masks follows enum mode defined in obc.h.
 *
 * ex. If mode_mask[1] refers to SUN_POINTING mode.
 * In order to stop task 0 from being run during sun pointing mode, and run all
other tasks, we set
 *     mode_mask[1] = 0b00000001
 *
 * The numbering of the tasks are shown in enum task, where each entry represents
the bit number.
 */

const unsigned char mode_mask[] = {
                                0b1100011, // Initial Mode
                                0b1110111, // Sun-pointing
                                0b1101111, // Nadir-pointing
                                0b1101010, // Safety
                                };

/*
 * Function to resume and suspend tasks based on the mode
 */
void mode_switch(mode newMode)
{
    //printf("now switching to mode %i\n", obc.mode);
    for (i = 0; i < ARRAY_SIZE(task); ++i) {
        if (task[i] == 0)
            continue;
        else if (mode_mask[newMode] & BIT(i)) {
            vTaskResume(task[i]);
            //printf("resume task %i\n", i + 1);
        } else {
            vTaskSuspend(task[i]);
            //printf("suspend task %i\n", i + 1);
        }
    }
}
```



```
}

// Send out necessary mode switching messages to tasks
switch (newMode) {
    case INITIAL:
        // tell Attitude task to detumble
    case SUN_POINTING:
        // tell Attitude task to do sun pointing
    case NADIR_POINTING:
        // tell Attitude task to do nadir pointing
    case SAFETY:
        // tell Attitude task to do passive rotation
    default:
        // error
}
}

void task_command_handler(void *arg)
{
    unsigned char deviceID[9] = {0};
    unsigned char FRAMread[10] = {0};
    int retVal, i;

    // initialize the FRAM
    retVal = FRAM_start();
    if(retVal != 0) {
        TRACE_WARNING(" Error during FRAM_start: %d \n\r", retVal);
        while(1);
    }

    // read the device ID
    retVal = FRAM_getDeviceID(deviceID);
    if(retVal != 0) {
        TRACE_WARNING(" Error during FRAM_protectBlocks: %d \n\r", retVal);
        while(1);
    }
    // TODO: check the device ID is as expected

    TRACE_DEBUG_WP("Device ID: ");
    for(i=0; i<sizeof(deviceID); i++) {
        TRACE_DEBUG_WP("0x%02X ", deviceID[i]);
    }
    TRACE_DEBUG_WP("\n\r");

    // Main task loop
    for (;;) {
        // read the command stack pointer
        retVal = FRAM_read(FRAMread, FRAM_COMMAND_STACK_POINTER,
        ARRAY_SIZE(FRAMread));
        if(retVal != 0) {
            TRACE_WARNING(" Error during FRAM_read: %d \n\r", retVal);
            while(1);
        }
    }
}
```

```
// debug print out the command stack pointer value
for(i=0; i<ARRAY_SIZE(FRAMread); i++) {
    TRACE_DEBUG_WP("0x%X, ", FRAMread[i]);
}

/* TODO:
    - use stack pointer to get the next task
    - read the time on the task
    - execute the task if time
    - else, delay until task time. continue
*/

// next command code read from the FRAM
command_code nextCommand;

// Execute the command
switch (nextCommand) {
    case CM_INITIAL:
        mode_switch(INITIAL);
        break;
    case CM_SUN_POINTING:
        mode_switch(SUN_POINTING);
        break;
    case CM_NADIR_POINTING:
        mode_switch(NADIR_POINTING);
        break;
    case CM_SAFETY:
        mode_switch(SAFETY);
        break;
    case BEGIN_IMAGING:
    case END_IMAGING:
    case BEGIN_DOWNLINK:
    case END_DOWNLINK:
    default:
        //error
}

//vTaskSuspend(NULL);
}

}

#ifdef _DEBUG
void task_debug(void *arg)
{
    xTaskHandle *hmode_switch = (xTaskHandle *)arg;
    unsigned int num = 0;

    for (;;) {
        printf("Debug Mode\n");
        printf("Enter a number to select the mode of operation\n");
```

```
printf("1. Initial Mode\n");
printf("2. Sun Pointing Mode\n");
printf("3. Nadir Pointing Mode\n");
printf("4. Safety Mode\n>> ");
fflush(stdout);
while(UTIL_DbgGetIntegerMinMax(&num, 1, 4) == 0);
if (num < 1 || num > 4)
    continue;
mode_switch(num - 1);

vTaskDelay(2000);
}
}
#endif

void obc_init(void)
{
    obc.mode = INITIAL;
}

void obc_main(void)
{
    obc_init();
    xTaskHandle task_mode;

    // This is where all of the tasks are created.
    xTaskGenericCreate(task_housekeep, (const signed char*)"housekeep", 4096,
NULL, HOUSEKEEP_PRIORITY, &task[HOUSEKEEP], NULL, NULL);
    xTaskGenericCreate(task_attitude, (const signed char*)"ADC", 4096, NULL,
ATTITUDE_PRIORITY, &task[ATTITUDE], NULL, NULL);
    xTaskGenericCreate(task_gps, (const signed char*)"GPS", 4096, NULL,
GPS_PRIORITY, &task[GPS], NULL, NULL);
    xTaskCreate(mode_switch, (const signed char*)"Mode Switching", 4096, NULL,
LOW_PRIORITY, &task_mode);
    xTaskCreate(task_command_handler, (const signed char*)"Command Handler",
4096, NULL, MEDIUM_PRIORITY, &task[COMMAND_H_HANDLE]);

#ifdef _DEBUG
    xTaskGenericCreate(task_debug, (const signed char*)"Debug", 4096, &task_mode,
configMAX_PRIORITIES-2, NULL, NULL, NULL);
#endif

    // start the tasks
    vTaskStartScheduler();
}
```

7.5.2 obc.h

```
/*
 * obc.h
 *
 * Created on: Oct 6, 2017
 * Author: Nathan Lim
```

```
*/

// choose whether system goes into debug mode or not
#define _DEBUG

/*
 * Bit selection.
 * BIT(0) = 0b0001
 * BIT(1) = 0b0010
 * BIT(2) = 0b0100
 * ... etc
 */
#define BIT(x) (1 << (x))

#define ARRAY_SIZE(x) (sizeof(x) / sizeof(x[0]))

// A base number for the first command code
#define COMMAND_CODE_BASE 0

/* List of all possible Satellite modes */
enum mode {
    INITIAL,
    SUN_POINTING,
    NADIR_POINTING,
    SAFETY,
    NUM_MODES,
};

/* List of all commands */
enum command_code {
    // Change Mode
    CM_INITIAL = COMMAND_CODE_BASE,
    CM_SUN_POINTING,
    CM_NADIR_POINTING,
    CM_SAFETY,
    // Imaging
    BEGIN_IMAGING,
    END_IMAGING,
    // Communications
    BEGIN_DOWNLINK,
    END_DOWNLINK,
};

void obc_main(void);
```

7.5.3 adc.c

```
/*
 * adc.c
```

```
*
* Created on: Oct 6, 2017
* Author: Nathan Lim
*/

#include <freertos/FreeRTOS.h>
#include <freertos/task.h>
#include <freertos/semphr.h>
#include <stdio.h>

#include "adc.h"

// enum to define which
enum adc_mode {
    DETUMBLE,
    SUN,
    NADIR,
    PASSIVE,
};

// mutex to protect the adc data
xSemaphoreHandle adc_mutex;

/* Attitude task data */
static struct adc_data adc_data;

void task_attitude(void *arg)
{
    adc_mode current_adc_mode;
    for (;;) {

        // TODO: check messages to see if we need to change the adc_mode

        // perform work based on the adc mode
        switch (current_adc_mode) {
            case DETUMBLE:
            case SUN:
            case NADIR:
            case PASSIVE:
            default:
                // error
        }
        vTaskDelay(100);
    }
}

// copy the adc data to the provided structure
void adc_get_data(struct adc_data &data)
{
    xSemaphoreTake(&adc_mutex, portMAX_DELAY);
    memcpy(&data, &adc_data, sizeof(struct adc_data));
    xSemaphoreGive(&adc_mutex);
}
```

