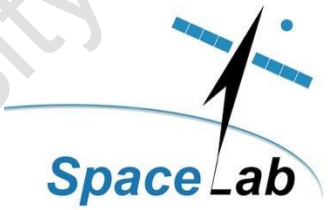
In-Flight Control Simulation of a Proposed, Future Microsatellite for the African Resource Management Constellation

Brendon Maongera



Department of Electrical Engineering
University of Cape Town

Dissertation submitted in partial fulfilment of the academic requirements for the degree of a Master of Philosophy in Space Studies.
05 March 2024

SL24-01M

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this master's dissertation from the work(s) of other people, has been attributed and has been cited and referenced.
3. This master's dissertation is my own work.
4. I have not allowed, and will not allow anyone to copy my work with the intention of passing it off as their own work or part thereof.
5. I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This dissertation has been submitted to the Turnitin module and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Name: **Brendon Maongera** **Student No: MNGBRE003**

Signature: Signed by candidate **Date:** 05 March 2024

Acknowledgements

I would like to thank my supervisors Professor Rene Laufer and Professor Jens Eickhoff for their support and providing the satellite testbench for my research.

I would like to thank Airbus Defence and Space and the University of Stuttgart for sponsoring the satellite testbench and providing the Onboard software for the “Flying Laptop”.

I would like to thank Bharadwaj Chintalapati and Kai Leidig for helping me become accustomed to the satellite testbench and answering questions I had about the testbench during my degree. They helped me solve many issues I was experiencing with the testbench.

I would like to thank my mom and sister for their continued support during this degree. Their words of wisdom kept me going during the difficult periods of the degree.

Abstract

The African Resource Management Constellation (ARMC) is a group of satellites that provide vegetation monitoring over the African continent. It is operated by the following four countries: Kenya, South Africa, Nigeria and Algeria. The constellation allows the four partner countries to learn to control and build their own satellite systems.

The University of Cape Town has been sponsored by a European consortium of academic and industry partners and has received a satellite testbench. The satellite testbench is a fully functional digital twin of the "Flying Laptop" satellite. The simulation testbench is commanded via the commercial mission control software and includes a detailed simulation of the satellite and all subsystems.

The University of Cape Town testbench can be the realistic nucleus of an ARMC mission. Starting from this setup a satellite model with improved remote sensing technologies can be defined for South Africa as well as for the ARMC with vegetation monitoring.

The constellation should operate at altitudes where there is high atmospheric drag, which will reduce the lifetime use of the satellites. An electric propulsion system can be used to restore the satellite to the desired altitude when commanded.

The current study aimed to perform a flight simulation of one of the constellation satellites that demonstrated vegetation monitoring over the African continent and modelled the Gecko Imager for the payload and an electric propulsion system on the testbench.

Each model was simulated in the Simulation Third Generation (SimTG), the flight and mission control software were enhanced and a simulation of the model with the satellite was performed. The research only focused on simulating a South African developed camera product for the payload and an electric propulsion system. The propulsion system was not designed but rather extracted from a previous student's paper. The software was enhanced for both models.

The simulated Gecko and electric propulsion system models were developed on the SimTG. Each model went through unit level testing to prove overall functionality. Each model was integrated with a satellite subsystem and that integration was tested. Other subsystem models were edited to accommodate the new models and the flight software was enhanced for the new models. The mission control system was updated to create telecommands and telemetry packets for the models.

The simulation of the models and the integration of the models to the satellite subsystems was successful. The Gecko Imager was able to capture images and the propulsion system was not able to improve the orbit of a satellite. The realistic flight simulation of the Gecko Imager was successful. Images of South Africa and Kenya were captured during the simulation. The orbit raise manoeuvre was not successful due to the thrust acceleration not overcoming atmospheric drag.

The simulation of the Gecko camera and the electric propulsion system into the SimTG was successful. All objectives were completed, and the enhancement of the flight software was successful and the creation of packets for commanding and telemetry on the mission control system was successful.

Table of Contents

List of Figures and Tables

List of Illustrations

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
AIAA	American Institute of Aeronautics and Astronautics
AIS	Automatic Identification System
APID	Application ID
ARM	African Resource Management
ARMC	African Resource Management Constellation
CADU	Channel Access Data Units
CCS5	Central Checkout System 5
CCSDS	Consultative Committee for Space Data Systems
CLTU	Command Link Transmission Units
CMDC	Command Count
CMDID	Command ID
CPDI	Combined Data and Power Management Infrastructure
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CYGNSS	Cyclone Global Navigation Satellite System
DC	Direct Current
DD	Design Document
DMC	Disaster Management Constellation
DSUB	D-Sub Micro-D
ECSS	European Cooperation for Space Standardization
EFM	Electrical Functional Model
EO	Earth Observation
EP	Electrical Propulsion
EPS	Electric Propulsion System
FEEP	Field Emission Electric Propulsion
FLP	Flying Laptop Platform
FLP1	Flying Laptop Platform Generation 1
FLP2	Flying Laptop Platform Generation 2
FOV	Field of View
FVB	Functional Verification Bench
GIT	Gridded Ion Thruster
GSD	Ground Spectral Density
HEMP	Highly Efficient Multistage Plasma
HEMPT	Highly Efficient Multistage Plasma Thruster
HET	Hall-effect thruster
ICD	Internal Control Document
IF	Interface
IO	Input-Output
IOB	Input-Output Board
IOBI	Input-Output Board Interface
IOBRMAPP	Input-Output Board Remote Memory Access Protocol
IoT	Information of Technology
IRS	Institute of Space Systems
LEO	Low Earth Orbit
LSB	Least Significant Bit
LVDS	Low-voltage Differential Signalling

MDVE	Model-based Development and Verification Environment
MGM	Magnetometer
MGT	Magnetorquer
MGTB	Magnetorquer Base
MIB	Mission Information Board
MICS	Multi-spectral Imaging Camera System
MSb	Most Significant Bit
MSB	Most Significant Bit
NanoCU	Nano Control Unit
NASA	National Aeronautics and Space Administration
NEUT	Neutraliser
OBC	Onboard Computer
OBSW	Onboard Software
OSRIS	Optical Infrared Link System
PCDU	Power Control and Distribution Board
PGA	Programmable Gain Amplifier
PPT	Pulse Plasma Thruster
PPU	Power Processing Unit
PSS	Power Supply Subsystem
PUS	Packet Utilization Standard
RF	Radio Frequency
RMAP	Remote Memory Access Protocol
RTEMS	Real-Time Executive for Multiprocessor Systems
SANSA	South African National Space Agency
SCOS	Satellite Control and Operation System
SimMF	Simulation Modelling Framework
SimOPS	Simulation Operation System
SimTG	Simulator Third Generation
SPI	Serial Peripheral Interface
STB	System Testbed
SVF	Software Verification Facility
TC	Telecommand
TM	Telemetry
TMTC	Telemetry and Telecommand
TWT	Traveling Wave Tubes
UCT	University of Cape Town
UHF	Ultra-High Frequency
USA	United States of America

1. Introduction

1.1 Background to the study

The African Resource Management Constellation (ARMC) is a proposed satellite constellation system for Africa. Its purpose is to provide vegetation monitoring for the African continent. The four partner countries that contribute to the project are: Kenya, Nigeria, Algeria and South Africa. [1]

An issue with most African satellite systems is that they are not built or operated by African countries. This affects local academia and the current and future industry. With the emergence of the ARMC, the four partner countries can therefore learn to control and build their own satellite systems with the South African National Space Agency (SANSA) as the lead for the ARMC project. This system would benefit South Africa because their focus is to improve remote sensing technology for their own vegetation monitoring needs.

The University of Cape Town (UCT) has been sponsored by a European consortium of academic and industry partners and has received a satellite testbench, a fully functional digital twin of the "Flying Laptop" satellite, operated by University of Stuttgart since July 2010. This simulation testbench is commanded via a commercial mission control software and includes a detailed simulation of the satellite and all subsystems including the flight computer down to its microprocessor. The simulated satellite runs the same real flight software as the original hardware "Flying Laptop" in orbit. [2]

In addition, the sponsored testbench includes the simulation model set for orbit dynamics, a dummy Earth observation payload and an electric propulsion system. The simulation infrastructure is from Airbus and is being used in their latest "Flexible LEO Platform" (FLP2) which is the industrialized version of the academic "Flying Laptop" [3][2]

Similar testbenches in the European industry are used as early models for a complete spacecraft and the testbenches are later subsequently enhanced by replacing simulated components by real hardware until the full satellite is assembled. The UCT Satellite Testbench is able to run the real flight software on the simulated computer controlling the simulated satellite. From this perspective the UCT testbench can be the realistic nucleus of a later mission. Starting from this setup a satellite model with improved remote sensing technologies can be defined for South Africa as well as for the ARMC i.e., the vegetation monitoring, to further help South Africa with their remote sensing technology.

The altitudes the constellation will operate in, they will experience high atmospheric drag which will contribute to the constellations orbital decay. A propulsion system can be used to restore the satellite to the desired altitude when commanded. An electric propulsion system for the constellation would be advantageous as it will increase the lifetime of the satellites with minimal fuel required unlike when you use chemical propulsion.

1.2 Problems and hypotheses statement

1.2.1 *Problem Statement*

The ARMC needs to build and control their own satellite constellation to be used for African vegetation monitoring. Local academia does not have the facilities to do this. To achieve this goal, satellites will need to be designed, simulated, developed, launched, and controlled by the South African team. The satellite testbench will provide a base to develop and simulate satellites.

1.2.2 *Hypothesis*

Simulating a South African built camera payload on the testbench will show it can aid satellite development for future satellites in the ARMC fleet. Simulating a propulsion system will show how a ARMC satellite will operate in space and can be used for future use.

1.2.3 Research aims

The aim is to perform a flight simulation at UCT of one of the constellation satellites to demonstrate vegetation monitoring over the African continent and model the Gecko Imager for the payload and the electric propulsion system using the FLP2 technology.

1.2.4 Research objectives and questions

- Can the satellite testbench simulate and show that a South African designed camera payload can be simulated and controlled.
- Can a micro electric propulsion system designed and developed by a student be used in orbital manoeuvres to increase the altitude of microsatellites that are in the ARMC constellation.
- Model the South African Gecko camera on the SimTG model.
- Enhance the flight software and mission control system to command the enhanced spacecraft with camera.
- Perform a realistic flight simulation of the virtual satellite monitoring vegetation over South Africa and Kenya.
- Enhance the simulated satellite with a simple electric propulsion Highly efficient multistage plasma thruster (HEMP) thruster system.
- Adapt the flight software and mission control for command/control of the propulsion subsystem.
- Perform a simulation of an orbit raise manoeuvre.

1.3 Scope and Limitations

The research to be performed will focus on using the testbench to execute a realistic flight simulation with the virtual satellite for fictious vegetation monitoring over South Africa. The satellite will be enhanced by a simulation model of a South African developed camera product for the payload. The satellite will be enhanced with the propulsion subsystem to perform simple orbit altitude adaptation manoeuvres. The propulsion system will not be designed but rather extracted from a previous student's paper. Only adaptions to the satellite models to include the two subsystems and the flight software. The mission control software commands are adapted for the new subsystems.

1.4 Research Contributions

A conference paper to be presented at the SpaceOps 2021 conference in Cape Town on the education/knowledge transfer using the simulator testbench and demonstrating the operation of the system over the Cape Town area.

1.5 Plan of development

The list of steps listed below are in order:

- Analyse literature on spacecraft modelling and flight software for the simulator.
- Understand modelling of satellite equipment in the simulator.
- Build-up elementary knowledge about the satellite flight software to implement simple adaptations.
- Understand satellite commanding and telemetry of the simulated spacecraft.
- Request an ICD model of the Gecko camera from SCS Space. Build a SimTG model of the camera and add it to the satellite on the FLP2 simulation.
- Upgrade flight software for the payload to be commanded on/off when the satellite is in nadir pointing operations mode.

- Define corresponding commands in the Mission Control System to command the payload of the virtual satellite.
- Develop a flight scenario for vegetation monitoring over the African continent using the simulator testbench and command the example flight scenario. Identify suitable target visibilities in cooperation with University of Stuttgart
- Enhance the satellite simulation with the electric propulsion model.
- Adapt the flight software for propulsion control.
- Adapt the mission control system with additional commands for propulsion control.
- Perform a flight simulation of a simplified orbit raise manoeuvre (Spiral-up).
- Submission of dissertation.

1.6 Dissertation Outline

The dissertation is organised into 10 chapters. Chapter One is the background and motivation for the study detailing the aims, methodology and scope and limitations of the dissertation. Chapter Two is the development of microsatellites through simulation and hardware throughout history. It also mentions the "Flying Laptop" Generation 2 testbench with a comparison to Generation 1 and details about camera payloads particularly the Gecko Imager. Chapter Two also includes how a satellites orbit is controlled and the different types of low Earth orbit satellite control, especially relating to altitude. The two altitude correction methods are compared to each other.

Chapter Three is the use of propulsion systems and focuses on three electric propulsion systems. Each propulsion system is discussed in terms of how it is developed and how it generates thrust from the different configurations. Micropropulsion systems are then discussed for microsatellites.

Chapter Four is how the Gecko camera is simulated on the Simulator Third Generation (SimTG). There is a breakdown of the sub-models that make up the Gecko Imager. Each sub-model's operation and functional development is shown and the interconnections between all the models. The Gecko connections to the satellite are demonstrated as well. Chapter Five is how the Micro-Highly efficient multistage plasma thruster (µHempt) propulsion system is simulated on the SimTG. There is a breakdown of its functionality and operation and the connections to the satellite subsystems.

Chapter Six is the additions made to the satellite subsystems to accommodate the new connected simulated models from the simulated models' structure to the internal code of each model. Chapters Seven and Eight discuss the tests that are performed to confirm the functionality of the models using different test cases. The test cases start from unit level testing to the mission control software testing.

Chapter Nine is the realistic flight simulation of the Gecko camera capturing images over the African continent and the orbit raise manoeuvre using the electric propulsion system in orbit. Chapter Ten is the conclusions of the dissertation and the recommendations to further improve upon the research conducted.

2. Micro-Satellites

A small satellite is any satellite with a mass less than 500 kilograms (kg). There are different types of small satellites which are classified by their mass as shown in Table 2.1. Small satellites have been used since the 1960s and their typical uses were for space environment data, to flight test various technologies and to provide communication[4]. Throughout the 1970s small satellites were being replaced by big satellites (mass > 500 kg), therefore, they started being utilised by amateur radio[4]. In the 1990s, small satellites made a comeback and have had a huge impact as more small satellites are replacing the big satellites because they provide the same applications but on a much smaller and lower development cost in the Low Earth Orbit (LEO) regions.

Table 2.1 Small Satellite Mass Classification

Satellite Class	Mass (kg)
Mini satellite	100 - 500
Microsatellite	10 - 100
Nanosatellite	1 - 10
Picosatellite	0.01 - 1
Femtosatellite	0.001 – 0.01

Small satellites can be used in constellations to provide global coverage for either IoT, internet services, Earth observation and disaster management services with high temporal resolution. Today, small satellites are predominantly used for Earth observation and science missions. The orbits for small satellites in the LEO region are in the following order of usage: sun-synchronous, non-polar inclined, polar, equatorial and others [5].

Sun-synchronous orbits are the most popular because the satellite is constantly facing the sunlight. This is particularly good for satellites that have Earth observation applications in the visible range [5]. Most small satellites today are launched as secondary payloads with nanosatellites being the highest launched small satellite class.

Microsatellites are satellites with a mass in the range 10 to 100 kg. Universities build microsatellites for technology demonstrations and capacity building. The popular application for this class of satellites are Earth observation payloads and science missions operating in LEO.

In 2016, 19 microsatellites were launched and since that year, the use of small satellites has double almost every year [6][5]. Eight of them were for the Cyclone Global Navigation Satellite System (CYGNSS) FM1-8 constellation. They were built by the NASA Earth System Science pathfinder program with the goal to improve predictions for weather [5]. Another microsatellite that was launched was the NuSat 1-2. These are commercial Earth observation satellites developed by Satellogic. Each satellite has a mass of 37 kg with 40 cm x 43 cm x 75 cm dimensions. [5]

Up until the end of 2022, 15 African countries have launched a total of 50 satellites. South Africa has launched the most with 12. [7] The following countries have at least launched a satellite: Algeria, Kenya, Zimbabwe, Rwanda, Sudan etc with Zimbabwe being the latest in November 2022.

In 2022, South Africa launched CubeSat constellation developed by Cape Peninsula University of Technology. This constellation is the first operational system and was not meant to be a technology demonstrator. [8] EOSat-1 is a South African designed satellite that will be the first satellite to be part of ARMC satellite constellation. [9]

2.1 The Future Low-Cost Platform

The future low-cost platform (FLP) is a microsatellite platform which was developed at the Institute of Space Systems (IRS) of the University of Stuttgart, Germany in partnership with Airbus Defence and Space. The platform targets small satellites in the range 50 to 150 kg with volumes of up to $1m^3$ and satellites that are seen as secondary payloads. [3]

The platform provides three-axis stabilization and is good for mission scenarios that provide Earth observation with nadir pointing or target pointing. The first satellite that was built using the platform was the "Flying Laptop". The "Flying Laptop" was built and developed by PhD and master's students from the University of Stuttgart, and it successfully launched in 2017. [3]

The "Flying Laptop" is a 120 kg Earth observation satellite with dimensions of 60 cm x 70 cm x 85 cm. The main purpose of the flying laptop was to space qualify electronic components particularly the Combined Data and Power Management Infrastructure (CPDI) for the Onboard computer (OBC) and Power Control and Distribution Unit (PCDU) integration. [3]

Figure 2.1 shows the qualified model of the "Flying Laptop" satellite and Table 2.2 describes the satellites' other characteristics. The "Flying Laptop" was developed with FLP Generation 1. The first generation cannot model a payload or a propulsion system. Therefore, the "Flying Laptop" does not have a propulsion system. But the second generation of the platform (FLP2) will be able to model and simulate a propulsion system and a payload system. It also provides constellation capabilities for either an Earth observation or communication constellation missions.

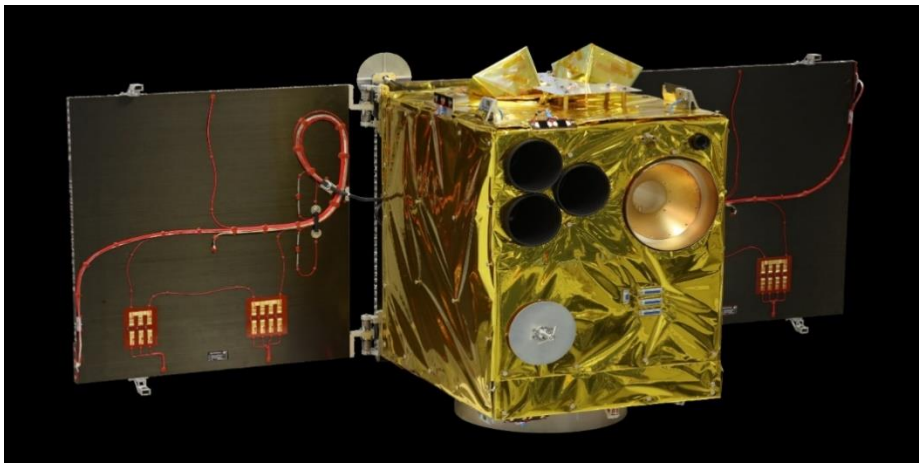


Figure 2.1: The "Flying Laptop" [10]

Table 2.2: "Flying Laptop" Characteristics

Launch Type	Piggy-back (Secondary payload)
Desired Orbit	Circular, polar
Orbit altitude	500 – 650 km
Attitude Control	3-axis stabilized
Communications	S-band (Primary), UHF (Secondary)
Solar Panels	3 (2 deployable)

The main payloads on the satellite are the Multi-Spectral Imaging Camera System (MICS), the Automatic Identification System (AIS) and the Optical Infrared Link System (OSRIS) communication system. The MICS is a multispectral camera consisting of three spectral bands for Earth observation. The AIS is used to

track the location of ships while the OSRIS is used to demonstrate the downlink speed using the optical terminal.

To demonstrate that the platform is reusable for other missions for small satellites, the "Flying Laptop" utilizes international and industry standards. The platform and the satellite use the Consultative Committee for Space Data Systems (CCSDS) and Packet Utilization Standard (PUS) communication protocols while SpaceWire handles the communication links between devices i.e., the I/O boards and processor boards. [11]

2.2 Simulation and System modelling

System modelling involves the detailed models of satellite systems and implementation of the thermal, structural and magnetic analysis providing realistic environmental effects and operations of the satellite. [12]

Satellite simulation involves the study of the dynamics of a spacecraft interacting with the space environment while performing its operations with the payload. Simulation can be classified into two terms: simulated systems referring to the real-world system and the simulation model which refers to an abstract of the real-world system.

Simulators provide a means for engineers to test and verify onboard software and control algorithms. This approach follows the V-model of development and verification. This approach is cheaper, and the development time is reduced drastically while hardware might not be available to test. For simulators to be useful all fundamental aspects of models need to be modelled with the use of the Interface Control Document (ICD) and the Design Document (DD).[13]

Simulators allow the Onboard software (OBSW) to be verified with multiple simulations with and without hardware. This reduces the development and verification time which reduces the cost and increases the quality of the software. All simulators follow a similar development infrastructure depending on the state of the project. [11]

The first stage is the "Algorithm in the Loop" or Functional Verification Bench (FVB) where mainly control algorithms are tested. This process is usually done in external programs like MATLAB. The second stage is "Software in the Loop" or Software Verification Facility (SVF). The algorithms are written in the targeted software (normally OBSW is C++) and the simulator has explicit models that simulate the whole satellite. Therefore, a simulated OBC model is available that can run the OBSW. All subsystems can be controlled in this stage and the spacecraft simulator is run via a control console. The TM/TC inputs from the console have to be propagated via simulated lines and data protocols. Therefore, the OBSW can be verified because all the subsystems and connections are simulated.

The third stage is the "Controller in the Loop" or System Testbed (STB). The OBSW is loaded onto the real OBC hardware to test software and hardware compatibility. The first tests that are done are the booting of the software on the OBC hardware and the TC/TM handling. The booting of the software is either done using a simulated/hardware IOB model which is connected to the simulator.

The fourth stage is "Hardware in the Loop". Simulated equipment are now replaced by their real hardware parts. The OBSW now controls the real hardware and the only simulated models are the flight dynamics and space environment. The satellite in this mode is called the Electrical Functional Model (EFM). Tests are conducted to check and control all satellite subsystems and their payloads.

2.2.1 *FLP Platform Generation 2*

Airbus Defence and Space developed their own satellite simulator which is now in its second generation. The "Flying Laptop" generation 2 platform (FLP2), an upgrade to the FLP generation 1 (FLP1), focuses on constellation capabilities with a few minor differences to FLP1 as shown in Table 2.3.

Table 2.3 Comparison between FLP generations

	Generation 1	Generation 2
Satellite operation software	SCOS-2000	CCS5
Processor	LEON3 UT609	LEON3 GR712
Cores	I/O Board	Router – I/O Board
Satellite Simulator	Model-based Development and Verification Environment (MDVE)	SimTG

Since the "Flying Laptop" was developed with FLP1, it used a single core processor and a separate payload computer to control all spacecraft functions via the CPDI [11]. With the FLP2, new satellites can be designed with the LEON GR712 dual core processor. One core processor will run the satellite's spacecraft bus subsystems while the other core processor will run the payload system.

The FLP2 uses the SpaceWire routing network. The network provides communication between the I/O boards and the CPU-board using a SpaceWire link. The CPU-boards are connected to two routers while the routers are connected to each other via SpaceWire cross-link [14]. The configuration allows for a hot-redundant and cold-redundant connection between all the subsystems.

To improve the simulator environment, the modern satellite Simulator Third Generation (SimTG) is used. The SimTG is where models are developed in the C++ programming language. A few examples of models developed are the onboard computer, the orbit dynamics, the payload subsystems and the attitude control subsystem. The FLP2 uses the Control Checkout System (CCS5) operating system instead of the Satellite Control and Operation System (SCOS-2000) [11]. The CCS5 is equivalent to a ground station being able to send commands and receive telemetry from the satellite.

The FLP2 also provides an extension for a propulsion system such as cold-gas, chemical and electric. The two electric propulsions considered are Highly Efficient Multistage Plasma Thruster (HEMPT) or Resistojet as these were considered in the design of the FLP2 for electric propulsion systems. [14]

The FLP2 flexibility is the use of C++ for its object-orientated programming for the Onboard Software (OBSW) which is run on the Real-Time Executive for Multiprocessor Systems (RTEMS) operating system and the OBSW includes the Packet Utilization Standard (PUS) stack [14]. The PUS is utilized in European space missions to define standards for telecommand and telemetry packets to adequately control a spacecraft subsystems and payloads from the ground mission control.

The FLP2 also provides multiple integrations of payloads for R&D or commercial targets for applications such as In-orbit-Demonstration/Validation and Earth observation and science missions for small satellite constellations or formations. [14]

2.3 Payload Systems

Satellite payloads are the main objectives of satellite development. Satellite bus systems are built around the payload to providing health keeping and storage of the payload system. Payloads for microsatellites can be categorized into five groups: communications, space science, technology verification, Earth observation (EO) and military applications.

The most commonly used payload system for microsatellites is Earth observation. The EO microsatellites can either be single satellites like the "Flying Laptop" or be in EO satellite constellations like the Disaster management constellation (DMC) [15] or the Planet Labs EO constellations: Doves, SkySats, and RapidEye

[16]. Earth observation missions are classified into the following: atmosphere, land, ocean and snow and ice. In each mission, payloads can either be active or passive systems and are further classified into optical and radar systems.

2.3.1 *Optical EO Payloads*

Optical payloads are sensors that use the visual and infrared bands of the electromagnetic spectrum to examine the surface of the Earth. They are predominantly passive systems which requires the light to reflect of objects on the ground to the satellite detectors. Typical applications include vegetation and disaster monitoring.

Optical payloads can either be multispectral or hyperspectral sensors. Multispectral sensors measure reflected wavelength in specific bands. They usually detect between 3 to 10 bands in each pixel of an image they produce. Hyperspectral cameras measure wavelengths in a range of continuous spectral bands. They detect hundreds/thousands of narrow bands between 400 and 2500 nm. [17]

The bands that both multispectral and hyperspectral can measure are in the visible, near infrared and short-wave infrared. The disadvantage of optical payloads is they are weather dependent due to cloud cover during operation and they can only operate during daylight.

Optical payloads were previously designed for large satellites, but a shift has occurred to develop them for small satellites. The most common development is for nanosatellites or CubeSats. A South African optical payload for CubeSats is the Gecko Imager. The Gecko Imager is a small compact CubeSat imager that provides RGB imaging at high frame rates and has a large high-speed mass data storage system [18]. The Gecko Imager was developed by the SCS Space.

There are 3 different versions for the module, they are: version 1.0 Gecko, the Leopard Gecko and the Crested Gecko. The payload module was designed by the Space Advisory Company, and it is compatible with 2U or larger CubeSat satellites [18]. The Gecko captures images in the RGB colour format at five frames per second. Table 2.4 shows the basic features for the Gecko.

Table 2.4: Gecko Imager Features

Spatial Resolution @ 500 km	39 m GSD
Swath @ 500 km	80 km
Image sensor	2.2 Megapixel RGB matrix
Data interfaces	LVDS, SPI and I2C
Dimensions	< 1U
Mass	< 480g
Operating Temperature	$+10^{o}$C to $+30^{o}$C
Integrated Mass storage	128 Gb

2.4 Satellite Orbit Control

Satellite orbit control is the maintenance of a satellite's altitude, eccentricity, inclination etc. Orbit maintenance is performed using thrusters on-board a satellite to mitigate the effects of the Earth's gravity, drag and space weather because these factors overtime effect the satellites orbit. The most common use for satellite orbit control is in geostationary satellites and satellites that operate in the Medium Earth Orbit. Orbital control of geostationary satellites is separated into North-South station keeping and East-West station keeping.[19]

Orbital control is also necessary in Low Earth Orbit. LEO satellites experience atmospheric drag and other perturbations which affect the satellites altitude. LEO orbital control has four control techniques to mitigate the atmospheric drag and other perturbations namely: Altitude correction, frozen orbit control, Sun-synchronous Orbit Control and Repeat Ground-track Orbit Control. The simplest is altitude correction.[19]

2.4.1 *Low Earth orbit control*

i. Altitude Correction

Altitude correction is a basic orbital manoeuvre which keeps the altitude of a satellite fixed. It does not control the particular position of the satellite on the desired orbit. There are two methods that can be used for altitude correction. They are the Hohmann transfer [19] and the spiral orbit transfer. Figure 2.2 shows the Hohmann orbit transfer.

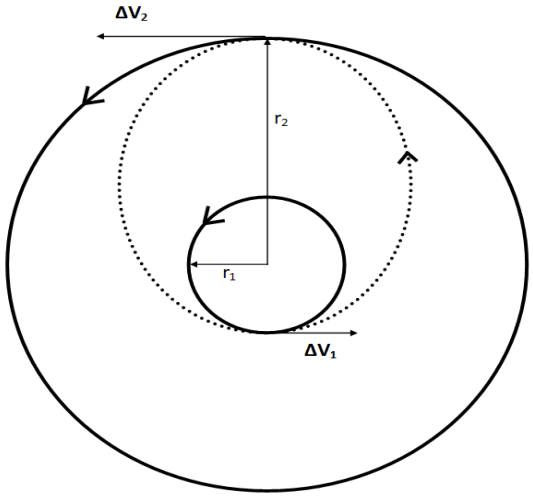


Figure 2.2 Hohmann Orbit Transfer and Spiral Orbit Transfer

The Hohmann transfer uses two impulsive manoeuvres for altitude correction. This involves the use of high thrust low specific impulse thrusters such as chemical thrusters. The thrusters are turned on at the apogee and perigee of the elliptical transfer orbit. Usually, only one half of the transfer orbit is used. This is the most fuel optimal impulsive transfer between two co-planar circular orbits for certain ratios of initial and final radius. In some cases, bielliptic transfer is better.

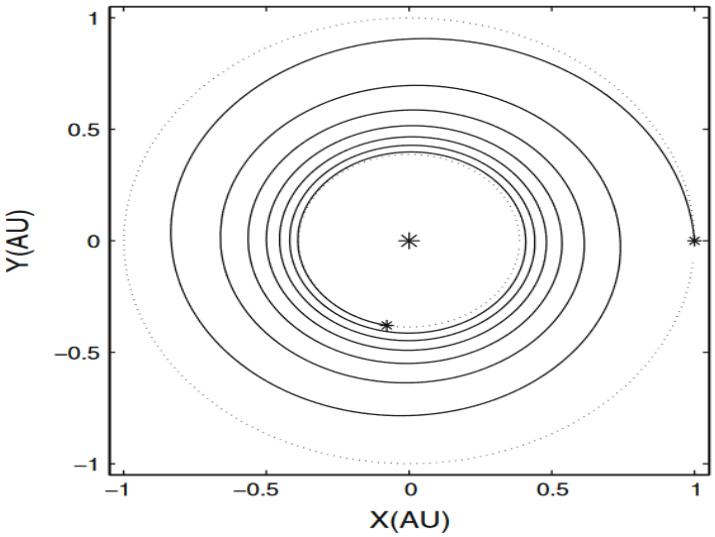


Figure 2.3: Low Thrust Earth-Mercury Orbit Transfer [20]

Figure 2.3 shows an example of a low thrust Earth-Mercury orbit transfer. The satellite would transfer from the Earth to Mercury. The spiral orbit transfer uses a continuous low thrust manoeuvre. This orbit is designed for power limited thrusters such as ion propulsion systems due to their high specific impulse with low thrust. The thrusters continue their operation until the desired orbit is reached. To increase the altitude of the satellite, the thrusters force must be tangent to the orbit and act in the same direction as the motion of the satellite. The opposite must be done to reduce the satellites' altitude.

3. Electric Propulsion Systems

A satellite propulsion system is a device that is used to provide thrust to move a satellite. There is a broad classification of propulsion systems for satellites. Spacecraft propulsion systems can be classified into three groups: chemical, electric, and propellant-less. Chemical propulsion uses the reaction of chemical propellants, liquid or solid, at high pressures which produces a large quantity of energy [21]. The most significant use for chemical propulsion is for launching objects into space because of the large amount of thrust with a small specific impulse.

Electric propulsion is mainly used for attitude, orbit control or orbit raising as it does not provide enough thrust as chemical propulsion to get off the ground. This is due to the low thrust levels generated with high specific impulse. EP can be divided into three categories namely electrothermal, electrostatic and electromagnetic.

Electrothermal propulsion is the use of electricity to heat propellant. The hot gas is then channelled through a nozzle which allows the conversion of thermal to kinetic energy [22][21]. The resistojet is the simplest thruster type for electrothermal. Common propellants used are hydrogen, nitrogen and ammonia. [21]

Electrostatics is positively charged particles, ions or colloids, are accelerated directly in an electric field [21] . The propellant used is either argon or xenon, but predominantly xenon is used. Commonly used thrusters are the gridded ion thruster (GIT), high efficiency multistage plasma thruster (HEMP-T) and field emission electric propulsion (FEEP). [21][23]

Electromagnetic propulsion is the interaction between the electric and magnetic field which induces a Lorentz force in a plasma [21]. The Hall-effect thruster (HET) and the Pulse plasma thruster (PPT) are predominantly used.

The typical ion thrusters described below are normally used for large satellite (i.e., satellites greater than 500 kg) but research for micro-propulsion is being developed for small satellites to control their orbit raising and for use in attitude control systems.

3.1 Ion Thrusters

The well-established thrusters that are utilized on satellites are the Grid ion thruster and the Hall-effect thruster and the more recently used the highly efficient multistage plasma thruster. A detailed description of each is described below.

3.1.1 *Hall-Effect Thruster*

The Hall-effect thruster was researched by the USA and the former Soviet Union. The USA placed more emphasis of its research on the magnetoplasmadynamic thruster while the Soviet Union built thrusters based on the Hall-effect. [21]

The schematic view of a basic HET is shown in Figure 3.1. The thruster is composed of a chamber, an anode used as a propellant feed, an external magnetic field generator and a cathode neutraliser. The external magnetic field generator induces a radial magnetic field, while the positive potential of the anode and the negative potential of the electrons, from the cathode neutralizer, generate the axial electric field. The interaction between the magnetic and electric fields generates a Hall current. The Hall current interacting with the magnetic field is responsible for the axial acceleration of the propellant ions.

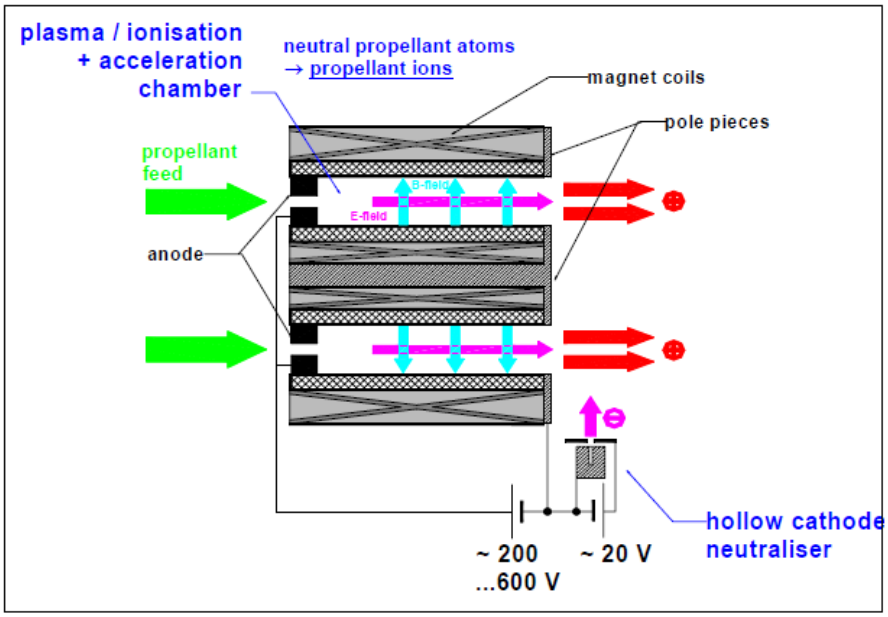


Figure 3.1: Hall Effect Thruster schematic view [23]

When the propellant feed is turned on, the propellant (e.g., Xenon) enters the chamber through the anode, the Xenon atoms are ionised by collisions with the high circulating energetic electrons producing ions. Since there is an electrostatic potential difference, the ions are accelerated out of the chamber while also removing an equal number of electrons to generate a neutralised plasma.

The main advantage of the HET thruster is its low power to thrust ratio which can be adjusted between 15 to 20 W/mN [23], while its main disadvantage is the erosion of the discharge chamber walls because of the ions impacting the walls during acceleration and exiting the thruster [23]. The thruster can not work without a neutralizer and it has a low ion beam efficiency of usually 60 to 70%. [23]

3.1.2 *Gridded Ion Thruster*

The Grid ion thruster has been studied since the 1960s. The thruster is composed of a chamber, an anode used as a propellant feed, a cathode neutralizer, acceleration grid, screen grid and plasma generator to ionize the propellant as shown in Figure 3.2. There are three types of plasma generators for the magnetic coils: direct current (DC) electron discharge, radio frequency (RF) discharge and microwave [24]. The most common is the radio frequency plasma excitation. Figure 3.2 is an example of the radio frequency ion thruster.

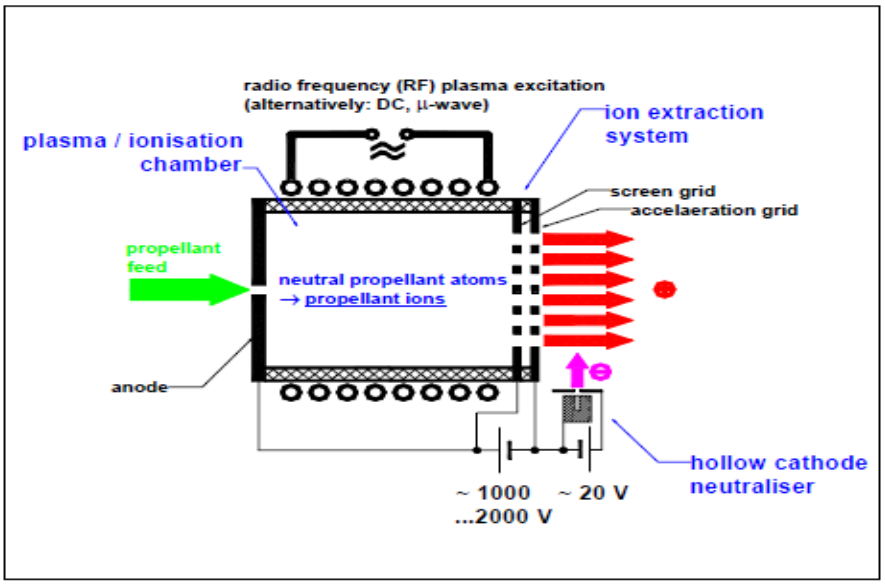


Figure 3.2: Grid Ion Thruster Schematic View [23]

The radio frequency ion thruster works by generating an axial magnetic field which induces a secondary circular electric field [11]. The electric field accelerates the electron and impacts the propellant creating an ion and another electron; therefore, plasma and ionisation occur inside the chamber.

The grid system is used to accelerate the ions out of the chamber to generate thrust. The screen grid is positively charged to attract the ions while the acceleration grid is negatively charged to accelerate the ions for thrust. The neutralizer cathode, which emits electrons at the same rate as ions exit, is used to prevent an increasing negative charge on the thruster. This causes ions to be attracted back to the thruster and decrease thrust. [24]

The main advantages of the GIT are the negatively polarized acceleration grid, its high beam efficiency in the range 80 to 85%, the separation of the ionization and acceleration zones and its high adjustable impulse range of 3000 to more than 5000s [23]. The main disadvantages are the complex turn on and turnoff characteristic, the complex control unit needed for operation and the erosion of the acceleration grid due to impacts and returned charged ions. [23]

3.1.3 *Highly Efficient Multistage Plasma Thruster*

The highly efficient multistage plasma thruster was first introduced in 1998 through a patent by the Thales Electron Devices [23]. They proposed that the thruster follows the traveling wave tubes (TWT) setup. The TWT is used for amplification of the radio frequency downlink on telecommunication satellites. The HEMP-T is composed of multistage plasma chambers, a positively charged high voltage anode used as a propellant feed and a hollow cathode neutraliser as shown in Figure 3.3.

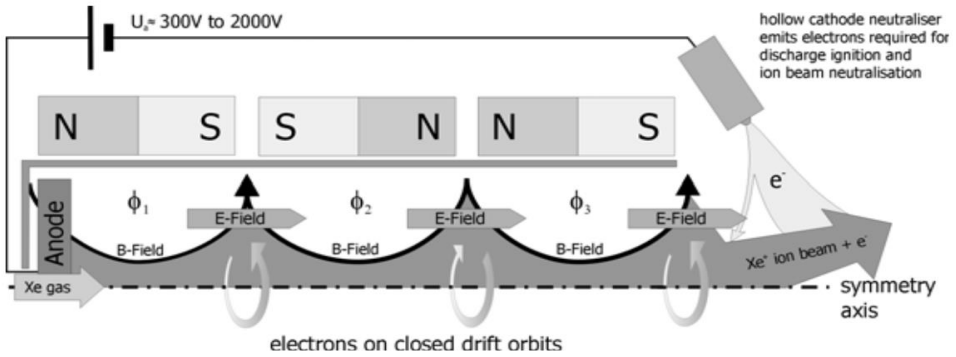


Figure 3.3: Highly Efficient Multistage Plasma Thruster Schematic View [25] [23]

The HEMP-T thruster works by using permanent magnets which generate magnetic cusps. The magnetic cusps confine the plasma to the magnetic field and minimizes the plasma's contact with the wall. The cathode neutraliser emits free electrons which are used to prevent the increasing negative charge as is the case in GIT thrusters. The potential difference created between the anode and the cathode creates an electric field whereby electrons will be pulled to the thruster exit. But the radial magnetic field lines at the cusp center create a barrier, therefore, electron mobility is reduced. [22]

The few electrons that are able to enter the chamber travel upstream through different magnetic cusps because of the elastic collisions providing gained kinetic energy. The electrons are held in closed drift orbits between magnetic cusps as shown in Figure 3.3.

When the propellant enters the chamber, it collides with electrons creating ions and extra electrons. In the cusp regions, the strong radial magnetic fields lines keep the plasma from contacting the walls and ions are accelerated by a strong axial electric field at the magnetic exit cusp[23]. The strong electric field in these regions keeps the ionisation process separate from the acceleration process. Since ions rarely collide with the walls, the ion beam is confined and generates a high thermal efficiency due to both minimal heat dissipation and high acceleration efficiency. [26]

The main advantages of the HEMP-T are the high ion beam power efficiencies, it can operate without a neutraliser, the turn on/off characteristic is very simple which allows for a simple control unit for operation and the erosion of the chambers is very minimal. [23]

3.2 Micropropulsion

Micropropulsion is the development of small propulsion systems for small satellites under 100 kg used in the LEO regions. These satellites can be CubeSats, nanosatellites, microsatellites and mini satellites.[27]

Small satellites launches have increased significantly since 2012 where there were 52 small satellite launches to 1743 in 2021 [6][28]. Development of thrusters, particularly electric thrusters for small satellites has increased since the world's first operation of an ion engine on the "Hodoyoshi 4" small satellite. Since 2013, multiple developments of different types of electric thrusters have operated on satellites such as the BRICSat-P Mission, Aerocube-8 satellites and the Delfi-n3Xt satellite [29] [30]. This is due to the fact small satellites are compact, have very limited power, limited budgets, and minimal design considerations to utilize the normal electric thrusters used of larger satellites.

Micro propulsion systems have also been used in mega constellations. Mega constellations are very large constellations, with hundreds or thousands of individual satellites. The first mega constellations were started in 2019 with operations in LEO such as Starlink by SpaceX and Airbus Oneweb by Eutelsat. Starlink satellites use krypton as fuel for its Hall-effect thrusters [31] while Oneweb utilizes the BHT-350 Hall-effect thruster[32]. Micropropulsion systems will allow small satellites to perform orbit raising, orbit maintenance, attitude control and end of mission disposal manoeuvres without impacting greatly on the design and cost. Since CubeSats are regularly built, there have been three very low power thrusters that have been developed: Micro Pulsed Plasma Thruster (µPPT), electrospray and micro ion thruster. [28]

3.2.1 *Micro Ion Thruster*

The micro ion thruster works the same way as ion thrusters used for large or medium satellites, except their operation is primarily for small satellites. The Micro Gridded Ion Thrusters mission applications are exploration, orbit/inclination change, attitude control, station keeping, spacecraft detumble and de-orbit [28]. Airbus has developed their own micropropulsion system based on the HEMP-T concept design[11]. In [11], the design and simulation of the micro HEMP-T propulsion system has been accomplished.

4. Gecko Imager Simulation Model

The Software Verification Facility (SVF) is located at the University of Cape Town. The facility has the structure as shown in Figure 4.1. The SVF uses the SimTG for simulating and testing models.

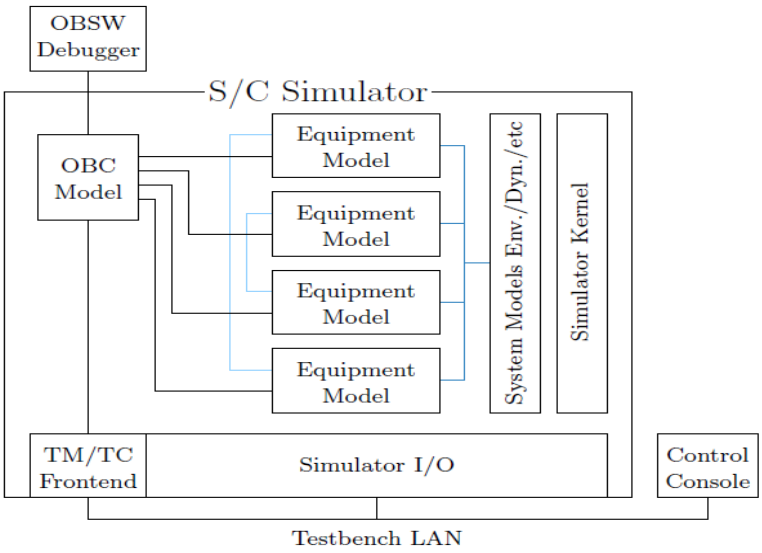


Figure 4.1: Software Verification Facility [11]

The SimTG environment is used to model the Gecko Imager. The SimTG modelling process is separated into two parts: The Simulation Modelling Framework (SimMF) and the Simulation Operation System (SimOPS). In the SimMF, the model will be created and written in C++ while in the SimOPS, the model's functionality is tested as well as the integration with other satellite subsystems. These verification tests are written in Java.

SimTG models can be developed in two modes: synchronous and asynchronous mode[33]. In synchronous mode, the input and output values of models are updated continuously according to the step time of the model. In asynchronous mode, the input and output values are updated based on event-based scheduling. This implies values are updated only when there is information needs to be sent or read. The changed values are only updated in the next step time. The Gecko will be developed according to the asynchronous mode because commands can be received at any time during the operation of the satellite.

The basic model structure simulated on the SimTG is shown in Figure 4.2. The left diagram indicates how the model is created in the SimTG and the functions that are called. The *step()* function is executed periodically during the simulation process until a stop simulation has been initiated. The step time of the function is defined during the verification tests and this is model dependent. The right diagram of Figure 4.2 demonstrates the functions found inside the *step()* function. [33]

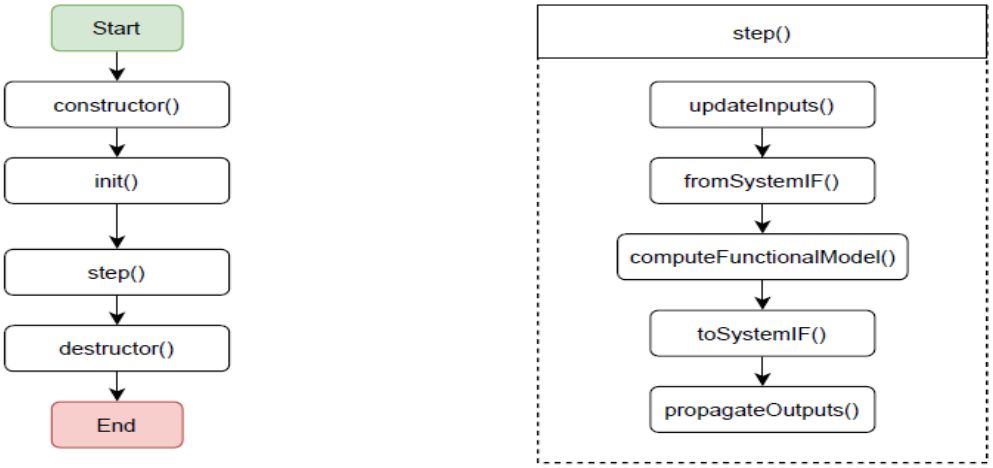


Figure 4.2: SimTG Model Structure[33]

The functions represent the operation of the simulated model. The *updateInputs()* places the data onto the input interfaces. The *fromSystemIF()* reads and stores the data on the input interfaces. The *computeFunctionalModel()* contains the algorithm for the functionality of the model based on the data from the input interfaces. The *toSystemIF()* sends the output of the algorithm to the output interfaces. The *propagateOutputs()* send the output values on the output interfaces. This process is suitable for a model in synchronous mode but there are slight differences for asynchronous mode. To model the Gecko, Figure 4.3. demonstrates the functions found inside the *step()* function.

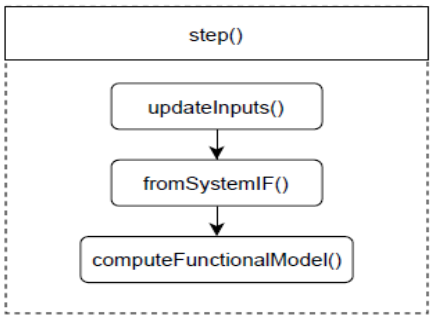


Figure 4.3: Gecko Model Structure

Since the Gecko Imager is developed in asynchronous mode, the output interfaces can only be updated depending on the algorithm developed in the *computeFunctionalModel()*. Since this is unknown, the toSystemIF() and *propagateOutputs()* are not included.

Gecko Imager has different versions and configurations. The version chosen for simulation in the SimTG is the Leopard Gecko Imager[34]. The Leopard Gecko is less than 390 g with dimensions of 100.5 x 96 x 52.1 mm. The control interface for Telecommand, Tracking and Communication is I2C interface and the data interface for the image data is the Low Voltage Differential Signal (LVDS). The I2C interface transfers all the commands and telemetry data between the satellite and the camera. The LVDS interface transfers all the image data from the camera to the satellite. The physical connector that is used will be a micro DSUB 9 pin interface[34] and the power interface for the Gecko is 5V and ground. [35]

4.1 Gecko Imager model

The Gecko Imager model is separated into two sub-models: the NanoCU and the image sensor. The NanoCU is further modelled into the NanoCUBoard and the flash. Therefore, the NanoCUBoard, the flash and the image sensor combine to represent the Gecko. All the connections between the model are illustrated in

Figure 4.4. The arrows represent the direction of information between all the models. The red arrows are powerlines, the green arrows are serial lines and the blue lines are data connections such as int, Boolean, double etc. For the Gecko, all blue arrows data connections are double.

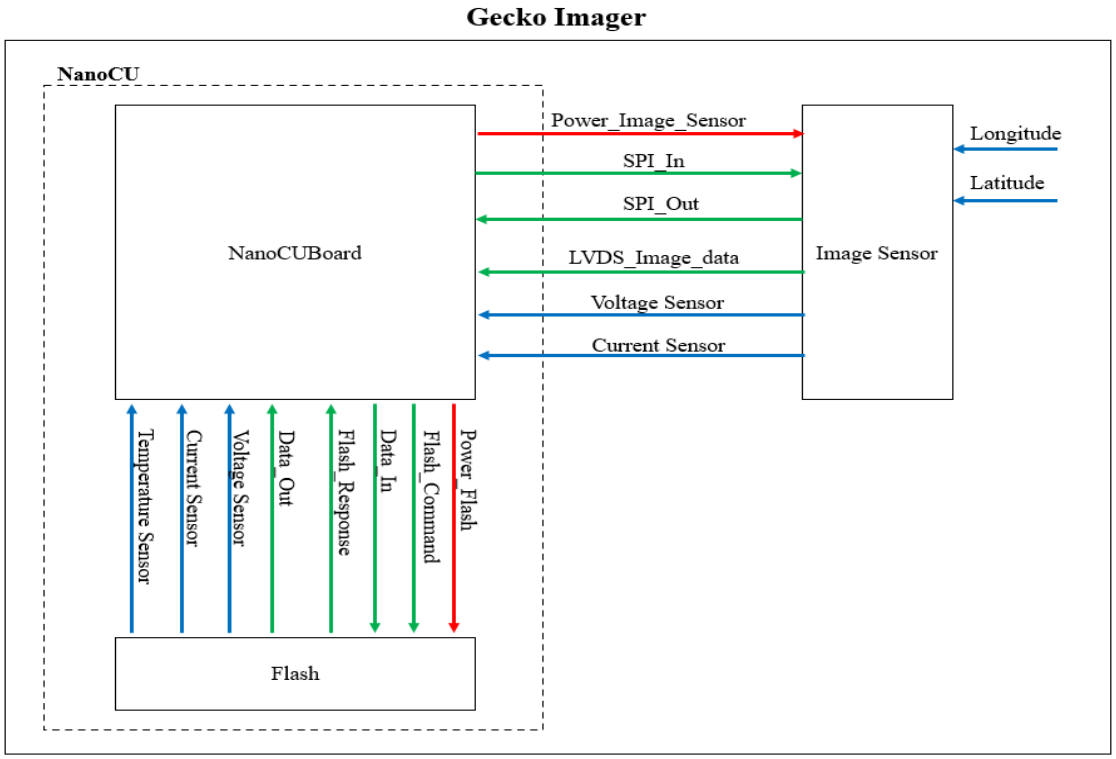


Figure 4.4: Gecko Imager interconnections[35]

Figure 4.5 illustrates the NanoCUBoard connections to the other satellite subsystems. The NanoCUBoard handles all communication to the satellite system, the commanding of the image sensor and the transmitting of image data and telemetry data to the satellite. A breakdown of each model's development is described in the following sections.

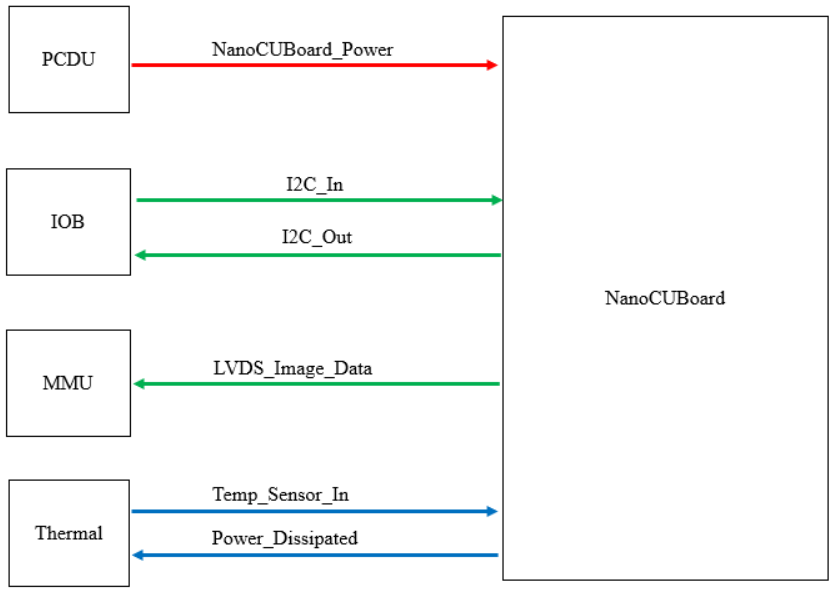


Figure 4.5: Satellite Connections to the NanoCUBoard

4.2 The NanoCU Board

The NanoCUBoard handles communication between the satellite and Gecko, as well as commanding the flash and image sensor. It executes all modes of operation for the Gecko and continuously updates the telemetry data from all connected models inside the Gecko. The functionalities of the board are separated into two main functions, the *fromSystemIF()* and the *computeFunctionalModel()*.

4.2.1 *fromSystemIF()*

The following functions are called: *getPower(), checkPower(), receiveRspFromSensor(), flashResponse(), updateTelemetryData() and readCmds()*. The *getPower()* function reads the supplied input voltage. The flowchart for the *checkPower()* function is shown in Figure 4.6.

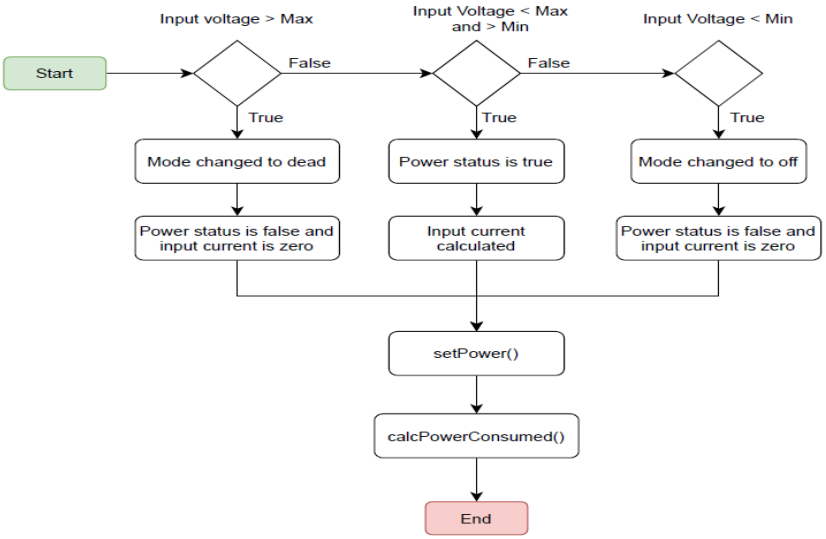


Figure 4.6: Flowchart for checkPower()

The function performs two operations. The first operation is to check if the input voltage is between the Gecko operating conditions, then the bool *powerStatus* is set accordingly. The second operation is to set the mode of the board according to the input voltage, thereafter, the input current is calculated as shown in Table 4.1. [35]

Table 4.1: Check power conditions

Input Voltage (V)	powerStatus	mode	Input Current (A)
5	true	off	Power consumed / Input voltage
>5.25	false	dead	0.0
<4.85	false	off	0.0

After calculating the input current, the functions *setPower()* and *calcPowerConsumed()* are called. The *setPower()* function outputs the board current to the PCDU as well as sets the required output voltage to the image sensor and the flash depending on the mode of the board as shown in Table 4.2.[35]

Table 4.2: Output Voltage to Image Sensor and Flash

Mode	Output Voltage (V)
Off/Dead	0
On/Erase/Read Image/Pre-imaging/Imaging/Image to Flash	3.3

The *calcPowerConsumed()* function calculates the power consumed depending on the board mode as shown in Table 4.3.[35]

Table 4.3: Power consumed for board mode

Mode	Power Consumed (W)
Off/dead	0.0
On/erase image/read image	1.4
Imaging/pre-imaging/image to flash	2.7

The *receiveRspFromSensor()* and *flashResponse()* functions are used to read the responses from the image sensor and the flash when they are done performing the commands sent from the board. The *updateTelemetryData()* function gathers all the telemetry data from all the models connected to the board. The list of telemetry data stored is the input voltage and input current, the flash voltage, current and temperature and the image sensor voltage and current. The telemetry data can be commanded at any time.

The *readCmds()* function receives and stores the commands sent on the I2C serial line into a command variable. Commands sent to the board are five bytes in length. The first byte is the register address with the most significant bit (MSb) either set or clear. The remaining bits are for the register address. If the MSb is set, then the remaining bytes are for the write data. If the MSb is clear, then the remaining bytes are for the readout data[35]. The basic Gecko command structure is shown in Table 4.4. A list of Gecko registers is shown in Table 4.5. All registers are 32 bits. To write to any register, the register data must first be read out, then change the required individual bits then send the write data to the register.

Table 4.4: Gecko Command Structure [35]

Command	Address byte	Data byte 1 - 4
Read	MSb is zero	Readout Data
Write	MSb is one	Write Data

Table 4.5: List of registers for the Gecko [35]

Register Address	Functionalities
0x0A	• Set bit 0 to turn on sensor (i.e. Imaging mode) • Set bit 29 to switch Gecko on. • Clear bit 29 to switch Gecko off.
0x02	• Set bit 1 to start reading data out process. • Set bit 2 to start capture image process. • Set bit 3 to start erase process. • Clear bit 4 to capture image data. • Bit 16 indicates flash has been initialised. • Set bit 12 – 14 to configure the LVDS frequency. • Set bit 15 to configure readout over dual channels. • Bit 24 is set when readout of data is done. • Bit 25 is set when capture image is done. • Bit 26 is set when image erased is done.
0x05	• Used to set the image ID to erase, readout or take image.
0x0D	• Set sensor PGA, ADC and sensor offset. Each setting takes a byte with sensor offset being the least significant byte.

0x0E	• Set the image sensor exposure time.
0x06	• Set the image sensor frame amount.
0x0B	• Set the image sensor frame rate.
0x0C	• Bit 17 is set when sensor is on. • Bit 18 is set when sensor is trained correctly.
0x29	• Readout all the telemetry data.

There are five options for setting the LVDS frequency in bits 12 – 14 in register 0x02. The data rates are shown in Table 4.6 [35]. If dual channel option is set, then the set clock frequency is split evenly between the two channels. The image sensor setting registers 0x0E, 0x06 and 0x0B each use the least significant byte to set the values.

Table 4.6: LVDS Clock Frequency

LVDS Clock Frequency (MHz)	Bit values that need to be set.
60	011
80	100
120	101
160	110
240	111

4.2.2 *computeFunctionalModel()*

The *computeFunctionalModel()* function determines which mode of operation the board is executed based on the bits set in the Gecko registers. The modes of operation for the board are off, on, erase, read image, pre-imaging, imaging, image to flash and dead. The flowchart for the function is shown in Figure 4.7. The board status can be idle, busy, done, wait and sentImageToSatellite. The board status determines which process of operation is the board currently undergoing. New received commands cannot be executed unless the board status is idle.

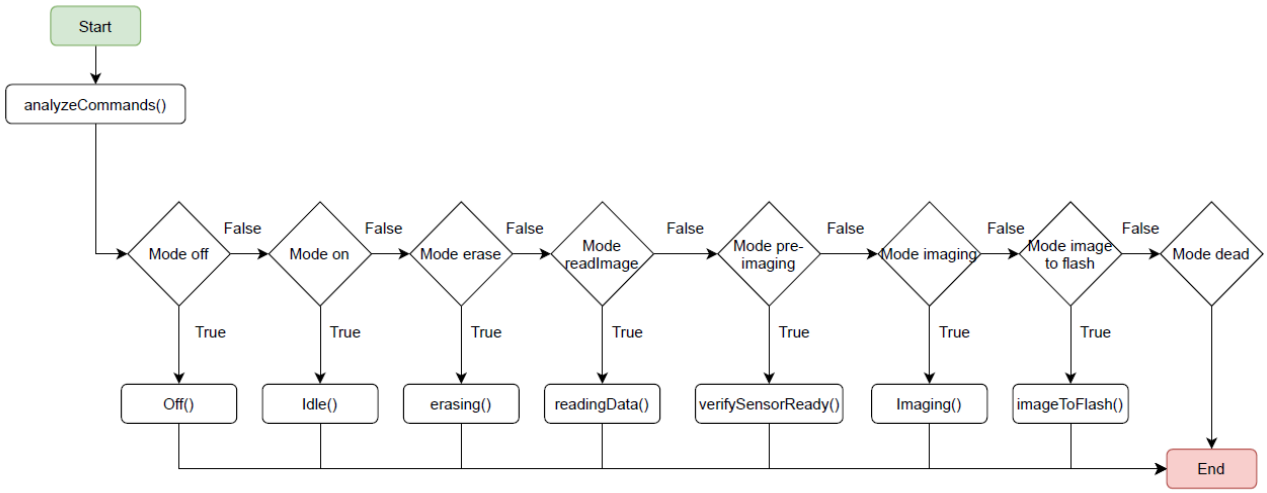


Figure 4.7: Flowchart for computeFunctionalModel()

In the *analyzeCommands()* function, the address byte is checked to see whether we are reading or writing to a register and which register is commanded. If the register is to be written, the write data is placed inside the register and if it's to be read, then the register data is sent to the satellite using the *sendRspToSat()* function.

The registers that follow this approach are register 0x29 and the sensor setting registers(0x0D, 0x0E, 0x06 and 0x0B). If register 0x29 needs to be read from, the *sendTelemetryData()* function is called.

Inside the *sendTelemetryData()* function, all the telemetry data is stored in a specific format inside an 80-byte packet. The first byte of the packet is for the logical address followed by the telemetry data. The logical address is used to tell the Input Output Board (IOB) where the telemetry packet is to be stored. The format of the telemetry data is as follows: a short description, followed by the voltage rail measurement and its associated current measurement and temperature measurement if available. When the packet is done being created, it is sent to the satellite. Inside the *analyzeCommands()* function, the *configureNanoCUBoard()* is called to set up the board settings for readImage mode. The bits are checked inside the function and the *dualOut* Boolean is set to true or false and the *clockFreq* is set. Default clock frequency is 120 MHz and default channel mode is one.

If the sensor setting registers need to be written, the *sendCmdSettingsToSensor()* function is called. The function sends the new register values to the image sensor.

i. Off mode

The *Off()* function is called. Off mode is the default mode for the board. Inside the *Off()* function, the powerStatus and register 0x0A is checked. If *powerStatus* is set and bit 29 is set in register 0x0A, the mode of the board is changed to on, the flash initialisation bit is set and an automatic turn on command is sent to the flash. The flash turn-on command is sent using the *sendFlashCommand()* function. The *sendFlashCommand()* function determines the size of data to be sent depending on the board mode. The default command size sent is one byte unless the board mode is erase or readImage then its five bytes.

ii. On mode

The *Idle()* function is called. Inside the *Idle()* function, certain registers and their bits values are checked. If the registers are set then the board mode can be changed accordingly. The board mode can be changed to erase, read image, pre-imaging and off.

Before any board modes can be started, each mode requires it to be set up before the process can begin. Before changing the board to erase, the following sequence must be followed:

- The image ID must first be set to register 0x05.
- Once set up, bit 3 in register 0x02 can be set to start the erase mode.

To change the board mode to readImage, the following sequence must be followed:

- The flash initialisation has to be checked in register 0x02.
- The image ID must be set in register 0x05.
- The settings for the LVDS such as the clock frequency and the dualOut option must be set.
- Once this is set up, bit 1 in register 0x02 can be set to start the readImage mode.

To change the board mode to imaging, the following sequence must be followed:

- The image sensor settings must be set in registers 0x0D, 0x0E, 0x06 and 0x0B.
- Once this is done, the sensor can be command on by setting bit 0 in register 0x0A. During this process, an automatic turn on sensor command is sent using *sendCmdToSensor()* function and the board status is changed to busy and the board mode is changed to pre-imaging mode to allow the board to set certain bits in register 0x0C based on the response from the image sensor. Once this is done, the board mode is changed to imaging.

iii. Erase mode

When the erase mode is started, the *erasing()* function is called. The flowchart for the erasing() function is shown in Figure 4.8. The *erasing()* function performs two operations. The first operation is to send an erase command to the flash with the image number set in register 0x05 using the *sendFlashCommand()* function. The first byte for the command is the erase command and the remaining bytes are for the image number. The image number is stored according to the most significant byte to the least significant byte. The board status then changes from idle to busy and no new commands can be executed while status is busy and erase mode is activated.

The second operation is to wait for a flash response to confirm the command has been executed. Once received, both the board status and board mode change back to on and idle respectively and bit 26 is set in register 0x02 to indicate image erased is done. To know if the erase mode is done, register 0x02 must be read by the satellite.

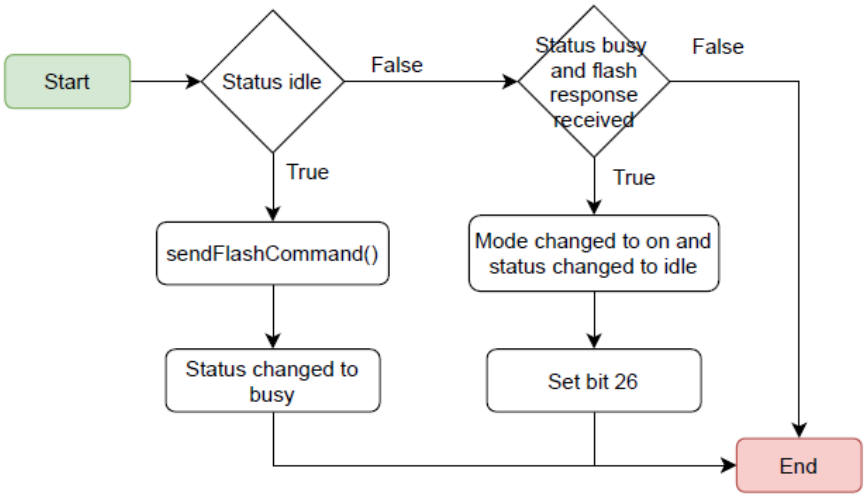


Figure 4.8: Flowchart for erasing()

iv. Read Image mode

When the read image mode is commanded, the *readingData()* function is called. The flowchart for the *readingData()* function is shown in Figure 4.9. The *readingData()* function performs four operations. The first operation is to send a read command to the flash with the image number set in register 0x05 using the *sendFlashCommand()* function and change the board status to busy.

The second operation is to wait for a response from the flash indicating the command has been executed and the image data has been sent to the board. Once the image data has been received, the *getImageFromFlash()* function is called. The function reads the image data received from the flash and stores it in a buffer. When the image data is done being read and stored, the board status is changed from busy to done.

The third operation is to check if the board status is done. If the board status is equal to done, then the image data is sent to the satellite depending on the configuration of *dualOut*, either using the *sendImageDataToSatellite()* or the *sendImageDataToSatelliteDual()* function. These functions send the image data to the satellite using one or two serial lines. Once all the image data has been sent to the satellite, the board status is changed to sentImageToSatellite. This occurs inside both functions.

A fourth operation is to check if the board status has been changed to sentImageToSatellite. If this is true, then the board status and mode are changed to idle and on respectively and bit 24 is set in register 0x02 to

indicate readImage is done. No new commands can be executed while read image mode is activated. To know if the readImage mode is done, register 0x02 must be read by the satellite.

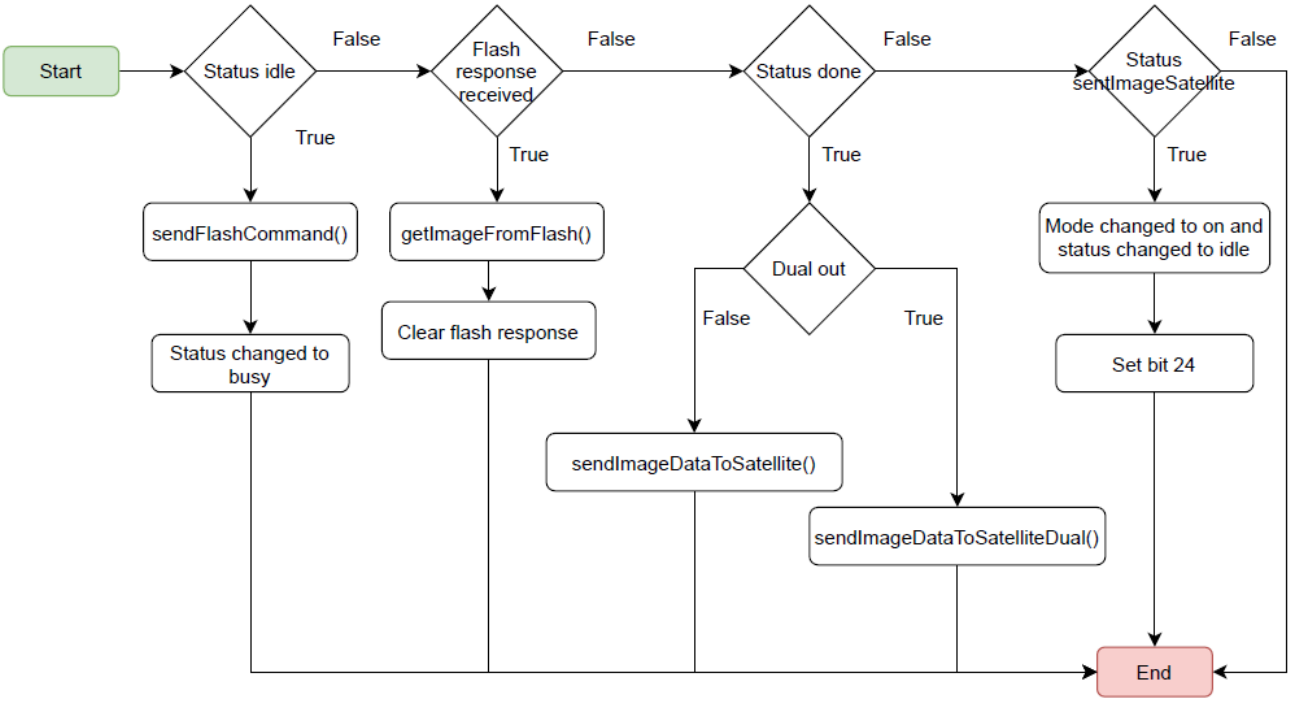


Figure 4.9: Flowchart for readingData()

v. Pre-imaging mode

This is a transition mode. When the pre-imaging mode is commanded, the *verifySensorReady()* function is called. The function is to check if the sensor is on and configured correctly. If the response from the sensor indicates its off, the board mode and status are changed to idle and bit 17 and 18 are not set in register 0x0C. If the response from the sensor indicates its trained correctly or not trained correctly and on, the board mode is changed to imaging and the board status is changed to idle and bit 17 and 18 are set in register 0x0C.

vi. Imaging mode

When the imaging mode is commanded, the *Imaging()* function is called. The function accepts the following two commands: image to flash and turn off sensor. To change the board mode to image to flash, the following sequence must be followed:

- The flash initialisation has to be checked in register 0x02.
- The image ID needs to be set in register 0x05.
- Set sensor to capture image data by clearing bit 4 of register 0x02.
- Once this is done, bit 2 can be set in register 0x02 to start the image to flash mode.

If bit 2 is set in register 0x02, the board mode is changed to image to flash. The image to flash command is for taking an image using the sensor based on the configured settings. When bit 0 is cleared in register 0x0A, the sensor is turned off, the board mode is changed to on and board status is changed to idle. If sensor settings are changed while the sensor is on, the settings will only be set if the sensor is turned off then turned on again.

vii. Image to flash mode

This mode is used to send a take picture command to the sensor, receive the image data and store the image data to the flash. The *imageToFlash()* function is called in this mode and the flowchart for the function is shown in Figure 4.10.

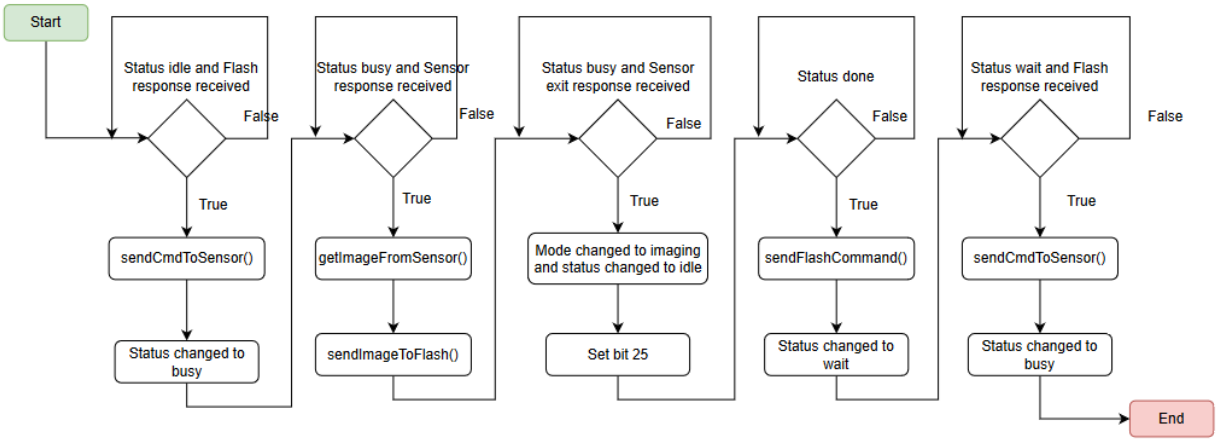


Figure 4.10: Flowchart for imageToFlash()

The *imageToFlash()* performs five operations. The first operation is to wait for the flash response to confirm its on while status is idle. If the flash is on, then the take picture command is sent to the sensor using *sendCmdToSensor()* and the board status is changed to busy.

The second operation is to wait for a response from the sensor indicating that it is done taking a picture and the image data has been sent to the board while status is busy. Then the *getImageFromSensor()* and *sendImageToFlash()* functions are called. The *getImageFromSensor()* function reads the image data on the serial line and stores the data in a buffer. The *sendImageToFlash()* sends the image data stored in the buffer to the flash and the board status is changed to done.

The third operation is to wait until the board status is changed to done. Then a write command is sent to the flash, using *sendFlashCommand()*, for the image data to be written to the memory blocks and board status is changed to wait.

The fourth operation is to wait for a flash response to indicate that the writing of the image data to the memory blocks is done. When the response is received, an acknowledgement command is sent to the sensor and the board status is changed to busy.

The fifth operation to wait for a sensor exit while status is busy. Then the mode is changed to imaging and status to idle and bit 25 is set to indicate image to flash is done. If another image needs to be stored, the board will receive another picture captured response and the process starts from the second operation again.

viii. Dead mode

Dead mode is only set if the input voltage supplied to the board is greater than 5.25 V. No commands will be executed in this mode and all other connected models will be off. The board cannot be recovered once in this mode.

4.3 The Image Sensor

The image sensor model is used to capture the images and immediately send the data to the NanoCUBoard to store the image. The *fromSystemIF()* function handles all the inputs connected to the model. The *computeFunctionalModel()* executes the commands received and changes the mode and status of the image

sensor. The five modes of operation for the image sensor are standby, configuration, idle, image to board and dead. The sensor status is either on and off and sensor operation status can be idle, busy, and done. The sensor operation status determines if the sensor is currently executing an operation and in what process of that operation.

4.3.1 *fromSystemIF()*

The following functions are called: *getPwr()*, *checkPower()* and *getCmds()*. The *getPwr()* function reads the input voltage supplied on the powerline and stores the value in the current voltage variable. The *checkPower()* function determines if the supplied input voltage is between the voltage operating range. If the voltage is within the range, then the *voltageOut* and *currentOut* data connection lines are set as well as *statusPower* is set to true.

The *voltageOut* is set to the current voltage and *currentOut* is calculated as power consumed divided by current voltage. If the voltage is not in the operating range then *voltageOut* and *currentOut* are set to 0 and *statusPower* is set to false. The *getCmds()* function reads the commands received on the serial line.

4.3.2 *computeFunctionalModel()*

The flowchart for the *computeFunctionalModel()* is shown in Figure 4.11. The image sensor mode is checked in the function, which then decides the which function under each mode is executed.

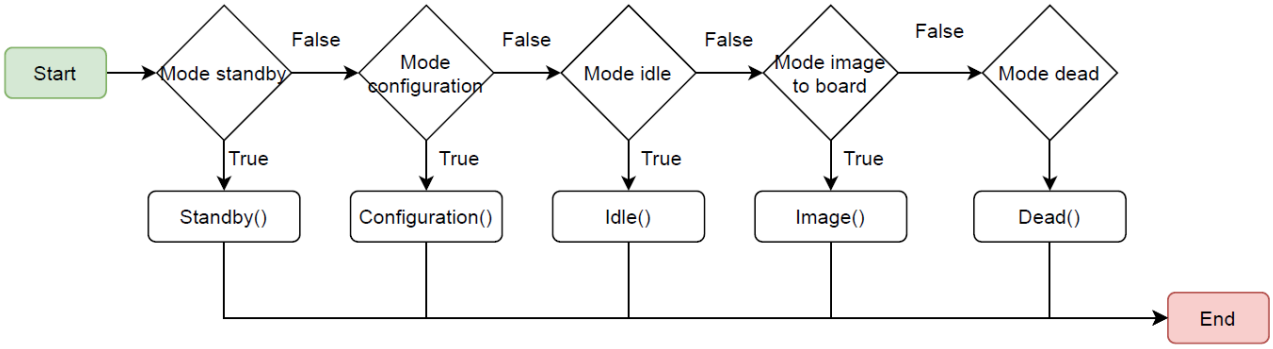


Figure 4.11: Flowchart for *computeFunctionalModel()*

i. *Standby mode*

In the *Standby()* function, only the configuration and turn on commands are valid. If the turn on command is received, the mode is changed to idle. The turn on command is only executed if *statusPower* is true. A response is sent to the NanoCUBoard using *sendResponse()* to confirm sensor is on and configured correctly and *statusSensor* is changed to on. The sensor configuration is indicated by *sensorTrained*. If the command is for configuration, the mode is changed to configuration and the *sensorTrained* is set to false.

ii. *Configuration mode*

In the *Configuration()* function, the settings of the image sensor are only changed when the sensor is off. When the sensor is off, the *configureSensorSettings()* is called. If the sensor is on, the sensor must be switched off then back on again for the settings to be configured. The *configureSensorSettings()* changes the settings according to the address number received. The list of addresses for each setting is shown in Table 4.7.

Table 4.7: List of setting addresses[35]

Address number	First byte	Second byte	Third byte
0x0D	PGA	ADC	Sensor offset
0x06	Frame amount MSB	Frame amount	Frame amount LSB

0x0B	Frame rate MSB	Frame rate	Frame rate LSB
0x0E	Exposure time MSB	Exposure time	Exposure time LSB

The frame amount is used to define the number of frames the sensor is meant to capture in one image to flash command. The frame rate is the time each frame shall be captured i.e., the delay between each image. The exposure time is how long each frame is exposed to the light. The sensor will be trained correctly if the right address is received else it's not trained correctly. The mode changes back to standby when finished.

iii. *Idle mode*

When idle mode is commanded, the *Idle()* function is called. Inside the *Idle()* function, the following commands are valid: image to board, turn off sensor, configure sensor settings and check if sensor is on. The flowchart for the function is shown in Figure 4.12.

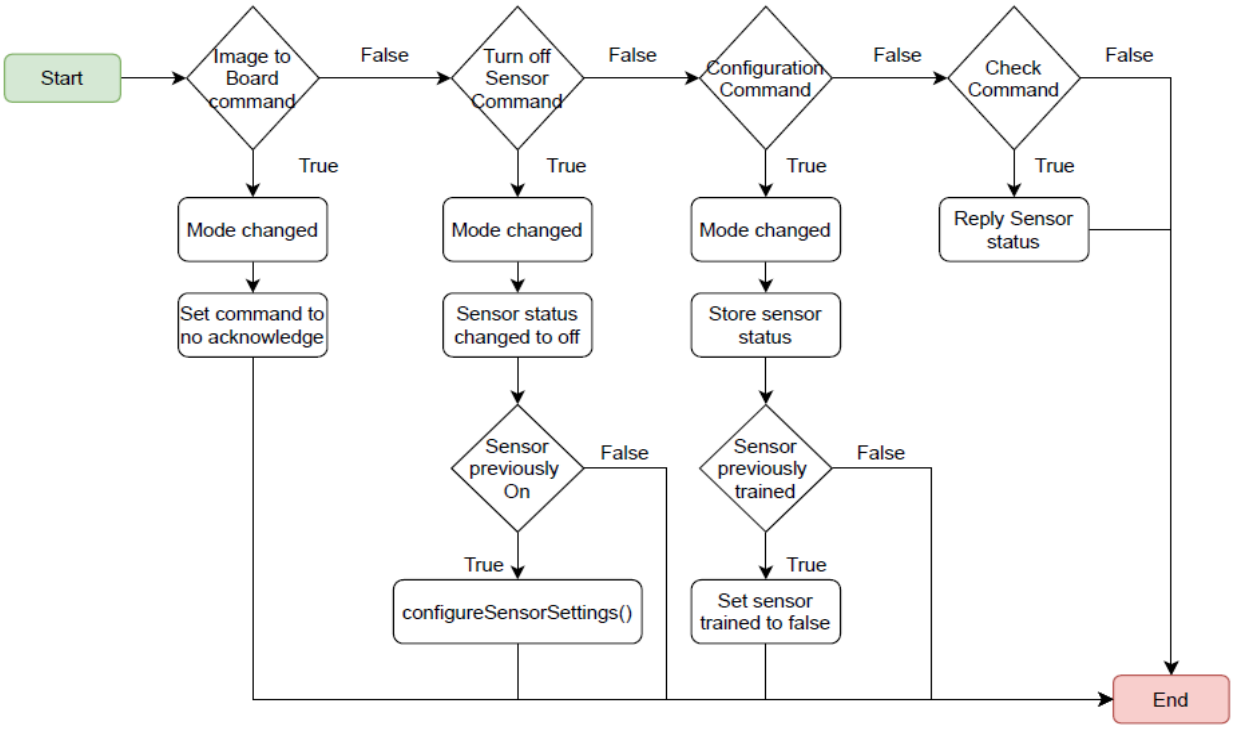


Figure 4.12: Flowchart for Idle()

If the image to board command is received, the sensor mode is changed to image to board and command is changed to no acknowledgement. If the turn off command is received, the sensor is changed to standby, sensor status is set to off and a check is done to see if the sensor was previously in on mode. If the sensor was previously in on mode, the *configureSensorSettings()* is called. This check changes the sensor settings when the sensor is off.

If a configure sensor settings command is received, the new settings are temporarily stored. Settings are changed when the sensor is switched off. The check command sends a response to indicate if the sensor is off or in idle mode.

iv. *Image to board mode*

When image to board mode is command, the *Image()* is called to capture an image/s of the target and the flowchart for the function is shown in Figure 4.13.

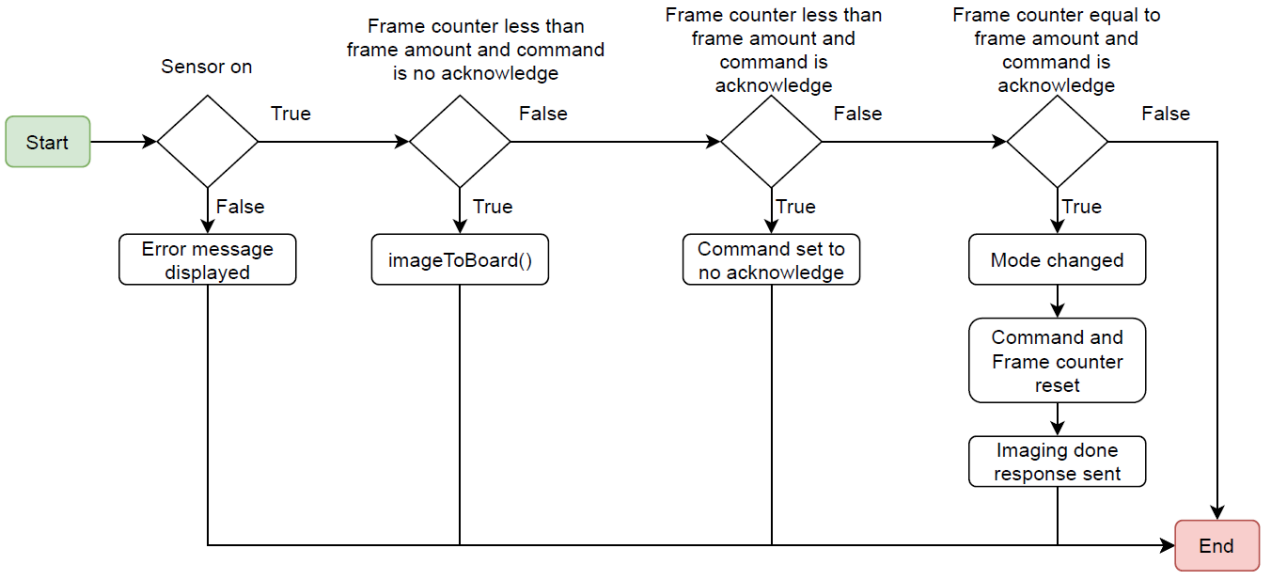


Figure 4.13: Flowchart for Image()

The *image()* function performs three operations. The first operation to capture a picture, therefore, the sensor status must be on, the frame counter must be less than the frame amount and the command must be no acknowledgement. If all conditions are true, the *imageToBoard()* function is called. The *imageToBoard()* calls three functions: *getCoordinates()*, *takePicture()* and *sendImageData()*.

The *getCoordinates()* function retrieves the longitude and latitude values for the target. The *takePicture()* function opens a PNG image and stores the image data in a buffer [36]. It also stores the longitude and latitude values in a text file, that will be read by a program called StaticMap. StaticMap is a python-based library that creates map images [37][36]. These images represent the instantaneous field of view of a camera payload at any instant. The ground spectral density (GSD) for the gecko imager is 39 m at 500 km altitude and a swath width of 80 km at 500 km. The GSD at 700 km is $\frac{700km}{500km} \times 39 = 54.6\,m \approx 55\,m$. To calculate the swath width, we use the following formula: [34]

$$\boldsymbol{Swath = 2 \times Altitude \times \tan\left(\frac{Field\ of\ view}{2}\right)} \quad 4\text{-}1$$

Using a field of view (FOV) angle of 9.15^{o} (calculated using swath at 500 km), the swath at 700 km is 112 km. To calculate the length of the FOV, we use the following formula:

$$\boldsymbol{Length = Altitude \times \tan(Field\ of\ view)} \quad 4\text{-}2$$

Using the same FOV angle value, the FOV length is 112.74 km. These dimensions provide the tile size that the camera instantaneously views from the satellite. This information will be used in the StaticMap program.

The library creates the images based on the pixel width and length, the longitude and latitude of the target and the zoom feature. The images created are tile-based. The zoom features correlate to altitude of the satellite in space. The pixel width and length are 2048 and 1088 as stated in [35] and the zoom value is 12 to approximate the altitude of the satellite at 700 km for the constellation. This creates an approximate size of the actual image the real Gecko captures.

Once the image data has been read, the *sendImageData()* function is called. The data stored in the buffer is sent to the NanoCUBoard and when its finished, the sensor status is changed to done. If status is done, a finished capture response is sent to the NanoCUBoard to indicate the image can be stored to the flash, the

sensor status is changed to idle, the frame counter is increased and the command is set to wait for acknowledgement.

The second operation is to check if the frame counter is less than the frame amount and the command is equal to acknowledge, therefore, another picture is taken and command is set to no acknowledgment. The command changing from wait for acknowledgement to acknowledgement is because the NanoCUBoard has confirmed that the image sent has been stored to the flash.

The third operation is to check if the frame counter is equal to the frame amount and command is equal to acknowledge, then an imaging done response is sent to the NanoCUBoard. The mode is changed to idle and frame counter is set back to zero.

v. Dead mode

The sensor changes to this mode only when the voltage supplied is greater than the maximum voltage limit.

4.4 The Flash

The flash model handles the erasing, writing, and reading of image data from the five sub flashes. These modes of operation are handled in the *computeFunctionalModel()* function and the *fromSystemIF()* function handles all the inputs into the model.

The flash has five sub flashes labelled A - E, each holding 22978 (size of each image is 2.28 Mb[35]) images which is equivalent to 51.20 Gb with a total storage capacity of 256 Gb. The sub flashes are objects of the class flashmemory. Each flash is initialized with a minimum and maximum image number as shown in Table 4.8.

Table 4.8: Sub flash initialization values

Flash	Minimum image id number	Maximum image id number
A	1	22978
B	22979	45956
C	45957	68934
D	68935	91912
E	91913	114890

The flashmemory class has four important functions: *readDataFromMemory()*, *writeDataToMemory()*, *eraseDataFromBlock()* and *generateMemoryBlocks()*. The *generateMemoryBlocks()* function creates a map of unique image id numbers using the minimum and maximum image id numbers entered in the constructor of the class. Each image is stored in one block and each block is 4 Kb [38]. To represent that the image id number is full or not, a Boolean element is used for the map value.

The *readDataFromMemory()* function determines which image data needs to be sent from the flash to the NanoCUBoard using the image number commanded. To represent an image data being read from memory, an actual image file is read from a folder where all the pictures captured are stored.

The *writeDataToMemory()* function determines where the image data is stored according to the image id's available in the specified flash and the *eraseDataFromBlock()* function deletes the specified image data from the flash. To represent the delete of an image, the map value is set to false and represent an image being added, the map value is set to true.

4.4.1 *fromSystemIF()*

The following functions are called: *getPower()*, *checkPower()* and *getCommands()*. The *getPower()* function gets the input voltage supplied and stores the value in a variable. The *checkPower()* function determines if the input voltage supplied is within the flash voltage operating conditions. If the input voltage supplied is within the operating conditions, then output voltage, output current and temperature are sent to the NanoCUBoard on the data connection lines and the flash mode changes to on. Output voltage is equal to input voltage, output current is calculated as power consumed divided by the voltage supplied and temperature out is set to 300 K. If the input voltage is not within the operating conditions, then the mode is off, and all sensor outputs are set to zero.

The *getCommands()* function receives all the commands sent from the NanoCUBoard. The format of the commands is the same as for the NanoCUBoard. The first byte in the packet is for the command value. The remaining bytes for erase and read image commands are for the image number. Its starts from the most significant byte to the least significant byte. The list of commands for the flash are listed in Table 4.9.

Table 4.9: Flash Commands

Command	First Byte	Second byte	Third byte	Fourth byte	Fifth byte
On	0x15	0x00	0x00	0x00	0x00
Off	0x16	0x00	0x00	0x00	0x00
Erase image	0x30	0x00	0x00	0x00	0x00
Read image	0x40	0x00	0x00	0x00	0x00
Write image	0x50	0x00	0x00	0x00	0x00
Check-flash-on	0x12	0x00	0x00	0x00	0x00

4.4.2 *computeFunctionalModel()*

The *computeFunctionalModel()* is where all the commands are executed, and the different modes of the flash are determined. The flash has 5 modes of operation with the off mode being the default. If a turn on command is received, the flash mode is changed to on. When the flash mode is on, the eraseImage, writeImage and readImage mode can be commanded. The flash has the following modes of operation: off, on, eraseImage, writeImage and readImage and the flash status can be idle, busy, and done. The flowchart for the *computeFunctionalModel()* is shown in Figure 4.14.

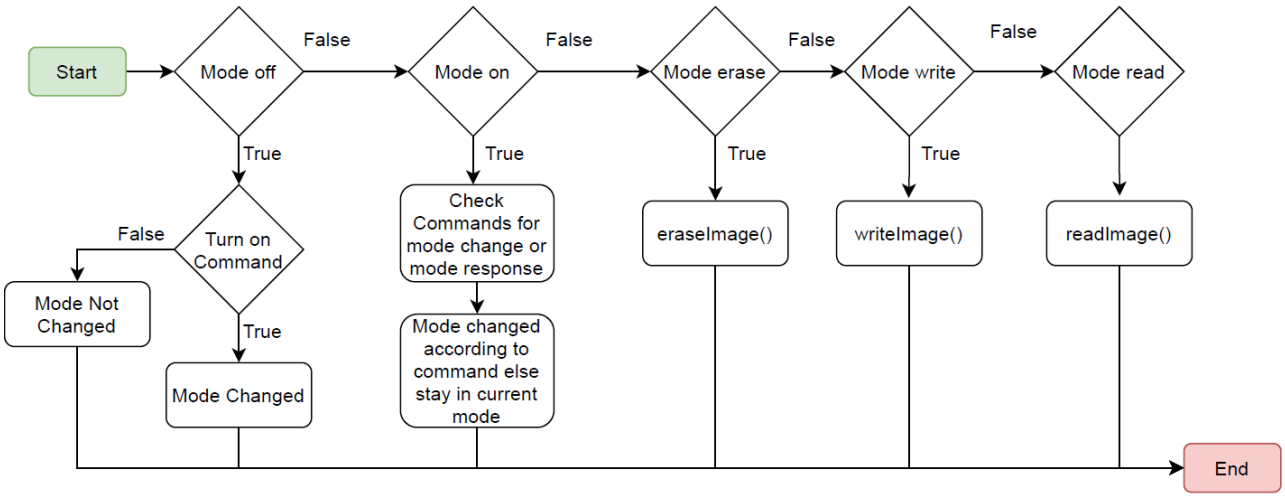


Figure 4.14: Flowchart for computeFunctionalModel()

i. Off mode

This is the default flash mode. The only valid command is the turn on command. If the turn on command is received, the flash mode is changed to on, otherwise, the mode does not change as shown in Figure 4.14.

ii. On mode

The on mode has the following valid commands to change the mode: erase, write and read. There is also an additional command to check if the flash is on. The flash mode is sent as a response in the *sendResponse()* function. The *sendResponse()* function sends any data that has been placed inside the reply buffer.

iii. Erase mode

When erase mode is commanded, the *eraseImage()* function is called. The flowchart for eraseImage() is shown in Figure 4.15.

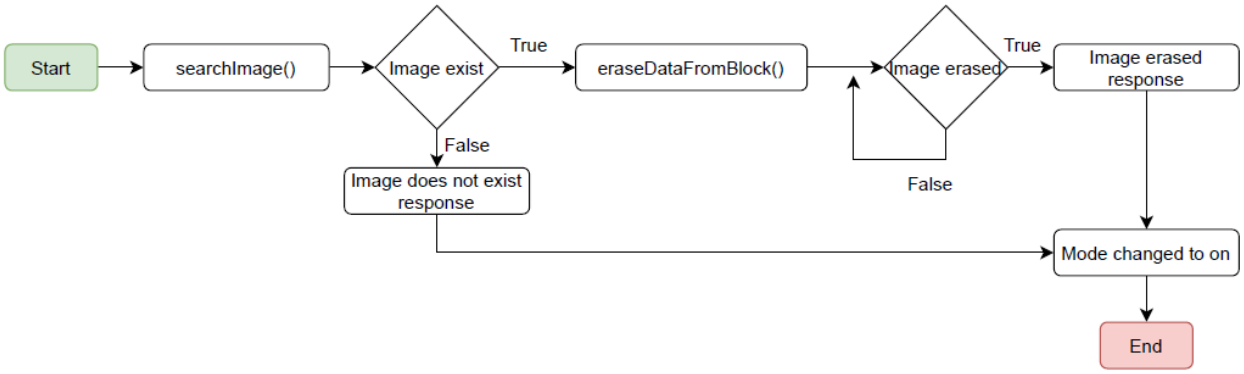


Figure 4.15: Flowchart for eraseImage()

The *searchImage()* function determines which sub flash contains the image number commanded. An image does not exist response is sent if the image number cannot be found and the mode changes back to on. If the image number is found, then the *eraseDataFromBlock()* function is called to delete the image from the specified sub flash. A delay is simulated to indicated the duration of deleting an image. Afterwards, an image erased response is sent and the mode is changed back to on.

iv. Write mode

When the write mode is commanded, the *writeImage()* function is called. The flowchart for *writeImage()* is shown in Figure 4.16.

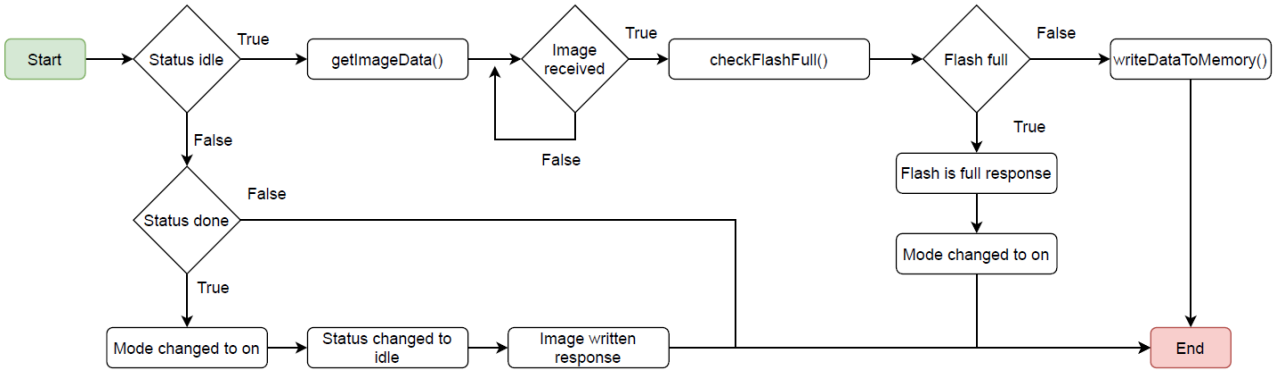


Figure 4.16: Flowchart for writeImage()

The flash status has to be idle before the *getImageData()* function is called. The *getImageData()* reads the image data from the serial line into a temporary data buffer. A delay is used to simulate the time required to

transfer the data. The *checkBankFull()* function determines if any of the sub flashes are full. If the flash is full, then a flash full response is sent, and the mode is changed to on.

If the sub flashes are not full, then a sub flash is chosen, and the data is transferred using the *writeDataToMemory()* function. Afterwards, an image added response is sent and the mode is changed to on. Once the image data has been transferred, the status is changed to done. If the flash status is done, the mode is changed to on, the flash status is changed to idle and the image written response is sent to the NanoCUBoard.

v. Read mode

When the read mode is commanded, the *readImage()* function is called. The flowchart for the *readImage()* is shown in Figure 4.17.

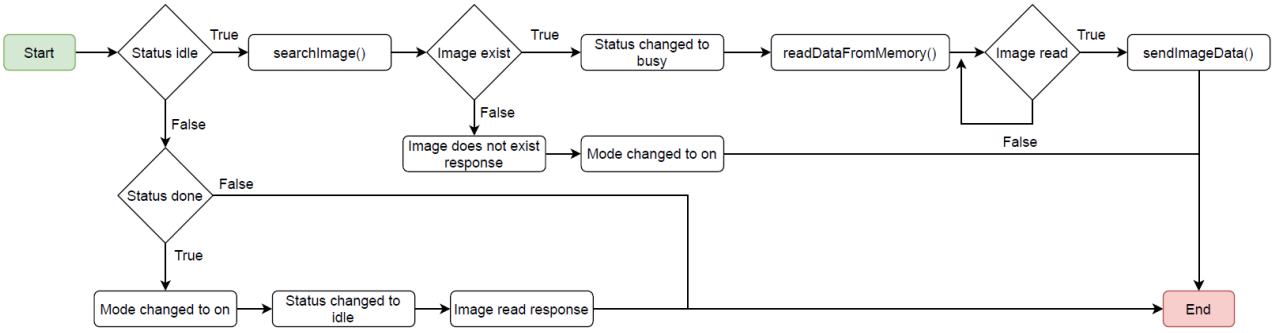


Figure 4.17: Flowchart for readImage()

The flash status has to be idle before *searchImage()* is called. If the flash status is idle, and the image exists, then the status is changed to busy and the image data is transferred from memory into the data buffer using *readDataFromMemory()* from the specified sub flash. A delay is used to simulate the time required to transfer the data.

Once the data has been transferred, the *sendImageData()* is called. The *sendImageData()* transfers the image data to the serial line and changes the status to done when the image data has been transferred. If the flash status is done, then an image read response is sent to the NanoCUBoard and both the sensor mode and status are changed to on and idle respectively. If an image does not exist, then an image does not exist response is sent and mode is changed to on.

5. EPS Simulation Model

The electric propulsion system (EPS) model chosen is the μHEMP-T propulsion developed by Airbus. The propulsion system was designed and simulated in [11]. The model is developed in synchronous mode on the SimTG; therefore, it follows the structure illustrated in Figure 4.2. The synchronous mode allows the EPS simulation model to continuously work for long periods of time. The EPS is comprised of different components which are the power processing unit, neutraliser, mass flow control, tank, and pipework.

i. ***Power Processing Unit***

The power processing unit is not an intelligent unit, its capabilities are limited to a high voltage converter. Therefore, μHEMP-T thruster makes use of a constant high voltage input for the anode voltage. The anode voltage will require 700 V from the 28 V satellite bus. The mass for the Power Processing Unit (PPU) is about 1 kg [11]. The FLP Power Control and Distribution Unit (PCDU) fuse commands the thruster on/off and monitors the switch system.

ii. ***Neutraliser***

The model uses tungsten filaments to emit electrons due to thermionic emissions [11]. Two neutralisers are used mounted on one thruster and feed respectively. They are separated by power lines from two different PCDU fuses.

iii. ***Mass Flow Control***

The mass flow controller system operates using bang-bang valves which are operated by the PCDU. The system uses three valves which can be configured to give six different distinct mass flow rates to the thruster. The basic mass flow rates are: 0.5, 1.0 and 1.5 sccm. By switching on different valves simultaneously, further mass flow rates are: 2.0, 2.5 and 3.0 sccm. [11]

iv. ***Tank***

The μHEMP-T uses Xenon as the propellant. The tank has a total mass of 2 kg with a dimeter of 20 cm. The hull thickness is 3 mm and has a titanium density of 4.5 g/cm^3. The location of the thrusters along with the tank are at the bottom of the "Flying Laptop". [11]

v. ***Pipework***

The pipework length is about 1.5 metres and includes fittings and has a pressure regulator which adds 0.5 kg to the system. [11]

To simplify the design of the EPS simulation model, all controls and data monitoring are handled by the PCDU model. The connections to the PCDU are the PPU, the NEUT(Neutraliser), and all the valves. These connections are powerlines between the two models. All the connections to the EPS are input interfaces because the model only consumes power. Another interface is for the thermal system. There are two PT1000 thermal sensors used to monitor the thruster to avoid overheating and damage to the magnets. The PCDU monitors the logs, the temperature measurements of the model and can transmit the temperature measurements telemetry of all the models connected to the satellite to the ground when requested. An input for the overall temperature of the spacecraft and output of power that needs to be dissipated are part of the model.

There were a few additional interfaces created as compared to the original design of the EPS developed. In the first design, the thrust output was a dummy holder to the EnvDyn model. In the new design, the thrust is connected to the ForceAndTorqueSum Model. The datatype for the thrust value is GForceData and it acts in

the y-direction of the satellite. The second addition made was for the fuel tank capacity. This is connected to the Structure model. This is necessary for fuel consumption because it affects the dynamics of the satellite when the EPS is in operation. All interfaces are shown in Figure 5.1.

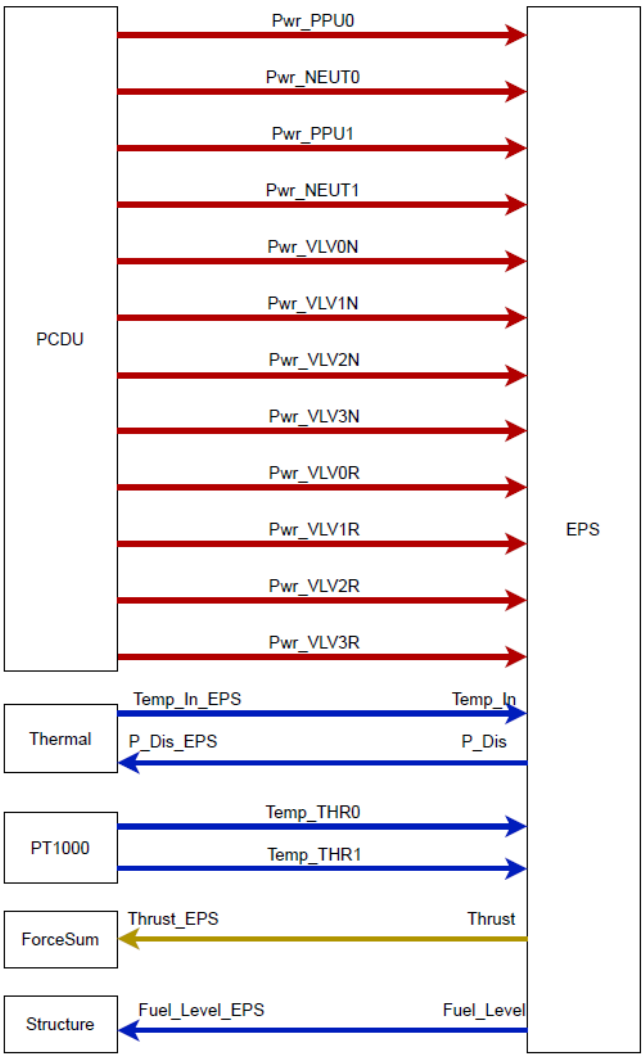


Figure 5.1: EPS simulation model interface

To switch on the EPS, the PCDU must switch on the PPU, the NEUT and the control valve. Once all the important systems are set, the operation point can be changed by changing which mass control valve is set. If any of the major parts are faulty, the redundancy component can be set while the faulty component is switched off.

The *fromSystemIF()* function handles all inputs into the model. The functions found inside are *getPower()* and *getThermal(). getPower()* reads the voltages applied to the powerline and *getThermal()* converts the input temperatures in kelvin to Celsius. The *toSystemIF()* handles all the data propagation out of the model. The functions found inside are *setPower(), setThermal()* and *setDynamics()*. The *setPower()* outputs the currents for the voltages read. The *setThermal()* only sets the dissipated heat for the thermal system and *setDynamics()* propagates the thrust and fuel tank capacity.

5.2 computeFunctionalModel()

Inside the *computeFunctionalModel()* function illustrated in Figure 4.2 is where all the models subsystems functionality is executed. The flowchart for the function is shown in Figure 5.2. When the *computeFunctionalModel()* is called, the *checkFailureState()* checks if the failure flag was set in the previous

iteration. The failure flag is only set if both Power Processing Units are on at the same time or if the temperature of the thruster exceeds the max limits.

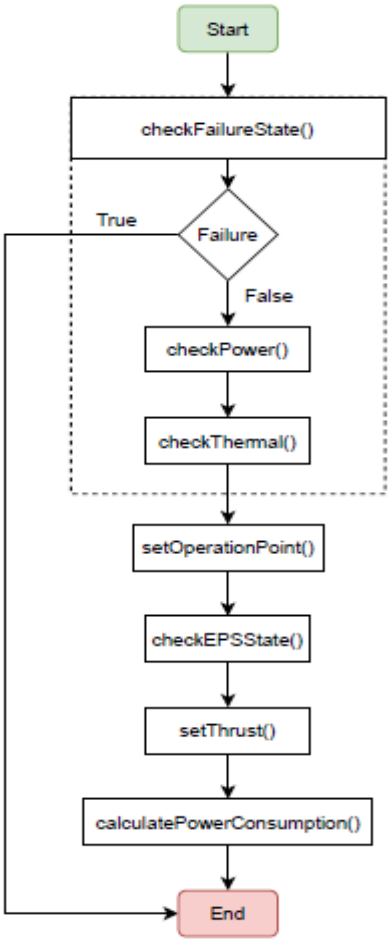


Figure 5.2: Flowchart for computeFunctionalModel() [11]

If not set, the *checkPower()* and *checkThermal()* are called. The checkPower() checks the voltages on all the powerlines between the EPS and PCDU and determines a status for each connection. The status for each connection can be no voltage, low voltage, nominal voltage and over voltage (e.g. StatusPPU0 = 0 means no power). *checkThermal()* does a similar function but with thermal levels of the equipment.

The *setOperationPoint()* determines the correct operation point depending on the mass flow valves that are open. Each operation point and its mass flow rate is shown in Table 5.1. There are seven operation points and each operation point has different thrust, power requirements and turn on/off sequences which are found in [11]. E.g. operation point 0 implies all mass flow valves are closed.

Table 5.1: Mass flow rate for each operation point

Operation Point	Mass flow rate (sccm)
0	0
1	0.5
2	1
3	1.5
4	2
5	2.5
6	3

The *checkEPSState()* checks all the components status that were checked during the checkPower() function to determine if the thruster can be started. If all the conditions for the thruster sequence are met, the thruster is started else the thruster is stopped. The flowchart for the *checkEPSState()* is shown in Figure 5.3.

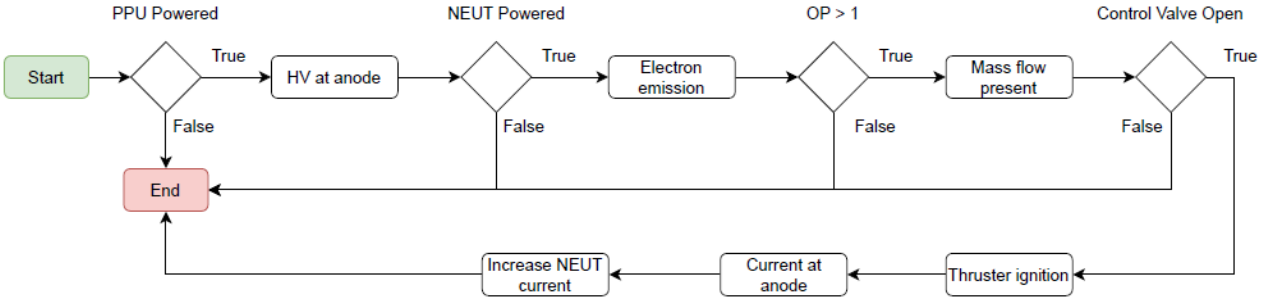


Figure 5.3: Flowchart for checkEPSState() [11]

The first is to check the PPU status. If one is powered, then the anode is provided high voltage. The second is to check the NEUT status. If powered, the electron emission is started. The third check is for the operation point to be grater than zero, which informs us there is a mass flow present. The last check is for the control valve to be open. If all conditions are met, the thruster is operational. There is current at the anode due to the power associated with the operation point and the neutralizer experiences a current increase for the ionisation process to neutralize the ejected ion beam at the thruster exit.

The *setThrust()* simulates the thrust arc and determines if the tank has enough propellant for operation. The *calculatePowerConsumption()* determines the required power needs based on the previous functions. This is a summarization of the EPS and a more detailed design of the model is found in [11].

6. Satellite Model Modifications

The models developed for the "Flying Laptop" needed adaptions to connect the Gecko and EPS simulation models. The satellite models that require modifications are the PCDU, the Input/Output Board (IOB), ForceSum and Dynamics model. The PCDU model controls and provides power to all the satellite subsystems. The IOB model connects all the models to the satellite depending on the model interface. The ForceSum model adds all the forces acting on the satellite. The Dynamics model handles the orbital dynamics of the satellite, the space environment and all the forces acting on the satellite.

6.1 PCDU model

Several aspects of the PCDU model needed adaptations for the two simulation models. Firstly, the fuse numbers, switch numbers and temperature sensor values needed to be updated for the new values (see Table 6.1 and Table 6.2).[39]

Table 6.1: Gecko and EPS fuse and switch values

Fuse Number	Number of Switches	Switch Number	Component	Component Number	Connection Type	Connection Type	Default State
27	1	77	Gecko Camera	52	Power Bus	Payload	Offline
28	2	78, 79	PPU 0	53	Power Bus	Nominal	Offline
	1	80	Neutralizer 0	54	Power Bus	Nominal	Offline
29	2	81, 82	PPU 1	55	Power Bus	Nominal	Offline
	1	83	Neutralizer 1	56	Power Bus	Nominal	Offline
30	1	84	Valve 0 N	57	Power Bus	Nominal	Offline
	1	85	Valve 1 N	58	Power Bus	Nominal	Offline
	1	86	Valve 2 N	59	Power Bus	Nominal	Offline
	1	87	Valve 3 N	60	Power Bus	Nominal	Offline
31	1	88	Valve 0 R	61	Power Bus	Nominal	Offline
	1	89	Valve 1 R	62	Power Bus	Nominal	Offline
	1	90	Valve 2 R	63	Power Bus	Nominal	Offline
	1	91	Valve 3 R	64	Power Bus	Nominal	Offline

Table 6.2: Gecko and EPS temperature sensors

Temperature Sensor Number	Component
32	Gecko Camera
33	EPS thruster 0
34	EPS thruster 1

The first modification to the PCDU model was to add the new powerline ports as output interfaces. To reduce the number of changes required to the existing code, the fuses for the Gecko and EPS were added at the end. The temperature sensors for the Gecko and the EPS are input interfaces to the PCDU model. Changes to the external model also require changes to the PCDU code. The *initInternalWiring()* [39] function initialises all the components connected to the PCDU model with additional parameters. The entry for the Gecko and the EPS has the following form:

initPCDUComponentList("Gecko", &_Pwr_GECKO, nullptr, nullptr, nullptr, 27, 0.55, SINGLE, POWER_BUS, PAYLOAD, OFFLINE, 52)

initPCDUComponentList("PPU 0", &_Pwr_PPU0, nullptr, nullptr, nullptr, 28, 5.0, DUAL, POWER_BUS, NOMINAL, OFFLINE, 53)

The parameters refer to the component name, power line connection (up to 4 are possible for this reason three “nullptr" entries), fuse number, maximum current, switch type (dual and single), connection type, component type, default state and the component number. There are five different component types for models: nominal, heater, payload, safe (used for safe mode) and working (use for the OBC and I/O board). Hence, the default state for the Gecko Camera and EPS is offline and online for all safe mode devices. The remaining components for the EPS follow the same structure.

Additional changes also need to be made to the *processTMpacket()* function[39]. The function gathers telemetry data for all the connections to the PCDU to transmit to the OBC. Due to the changes to the fuse, switch and temperature sensor numbers, the byte numbers for each need to be updated to accommodate the new values.

6.1.1 *EPS command control*

To command the EPS through the PCDU, the command protocol for the EPS is separated into two parts. The first part makes use of the PCDU basic command protocol [11][39]. The 8-byte structure for the command protocol is shown in Table 6.3.

Table 6.3: PCDU Basic protocol structure

Byte Number	0	1	2	3	4	5	6	7	8…
Value	LEN1	LEN0	CMDC	CMDID	P1	P0	CRCH	CRCL	Data

The meaning of the bytes are as follows:

1. LEN1 and LEN2 represent the length of the data block following the command. It is in big endian and an initial 8 bytes of a command are mandatory.
2. CMDC is the command count. It is used for every sent command for acknowledgement identification.
3. CMDID is the command identification. It is used to identify the command group.
4. P1 and P2 represent the command parameter.
5. CRCH is the cyclic redundancy check (CRC) 16-bit higher byte (Byte 0 - 5).
6. CRCL is the CRC 16-bit lower byte (Byte 0 - 5).
7. Data is the optional block of data to be transmitted. 65536 is the maximum number of bytes plus 2 bytes for its CRC.

The PCDU has a list of commands that can apply to the EPS such as “Request Telemetry Packet”, “Change Status of Switch”, “Check Status of Switch”, “Command status of Fuse: On”, “Check Status of Fuse”, “Read Current of Fuse” and “Read Temperature”. [11]

Table 6.4: PCDU command list

Command	Explanation
Request Telemetry Packet	Creates a telemetry data packet for all the connections to the PCDU and sends the packet to the OBC.
Change Status of Switch	Status of switch is changed to on/off
Check Status of Switch	Returns the status of all the switches
Command status of Fuse: On	Changes the status of the fuse to on
Check Status of Fuse	A packet is sent to the OBC with the status of all the fuses.
Read Current of Fuse	A packet is sent to the OBC with the currents of all the fuses.

Read Temperature	A packet is returned with all the temperatures values.

All these commands follow the 8-byte structure of the PCDU. These commands can be used to switch on/off the EPS PPU, neutraliser and control valve switches. To switch on a switch in the PCDU, the fuse must be enabled hence the commands of "Command status of Fuse: On" and "Check Status of Fuse".

The second part was to develop a new command to control the mass flow valves for the different operation points. To simplify the complete thrust manoeuvre, the "Command Thrust" command was added to the PCDU [11]. The structure of the command follows the PCDU 8-byte protocol and has the following form:

Command	LEN1	LEN0	CMDC	CMDID	P1	P0	CRCH	CRCL
Command Thrust	0x00	0x00	CMDC	0x40	On/Off	0…7	CRCH	CRCL

The value of 0x40 for CMDID is the command ID for the command thrust. The first parameter is to switch the valves on/off to start or stop the thrust sequence. The value of 0xF0 is for on and 0x0F is for off. The second parameter is to determine the operation point for thrust and power. An example is to start the thrust with a mass flow rate of 2 sccm (Operation point 5) would be:

00 00 01 40 F0 05 23 3F

The CRC calculation code is listed in A. 1. When the PCDU receives the "Command Thrust" command, the *epsTMTCControl()* is called. The function handles all the operations of controlling the mass flow valves depending on the data received. The flowchart for the *epsTMTCControl()* [11] is shown in Figure 6.1.

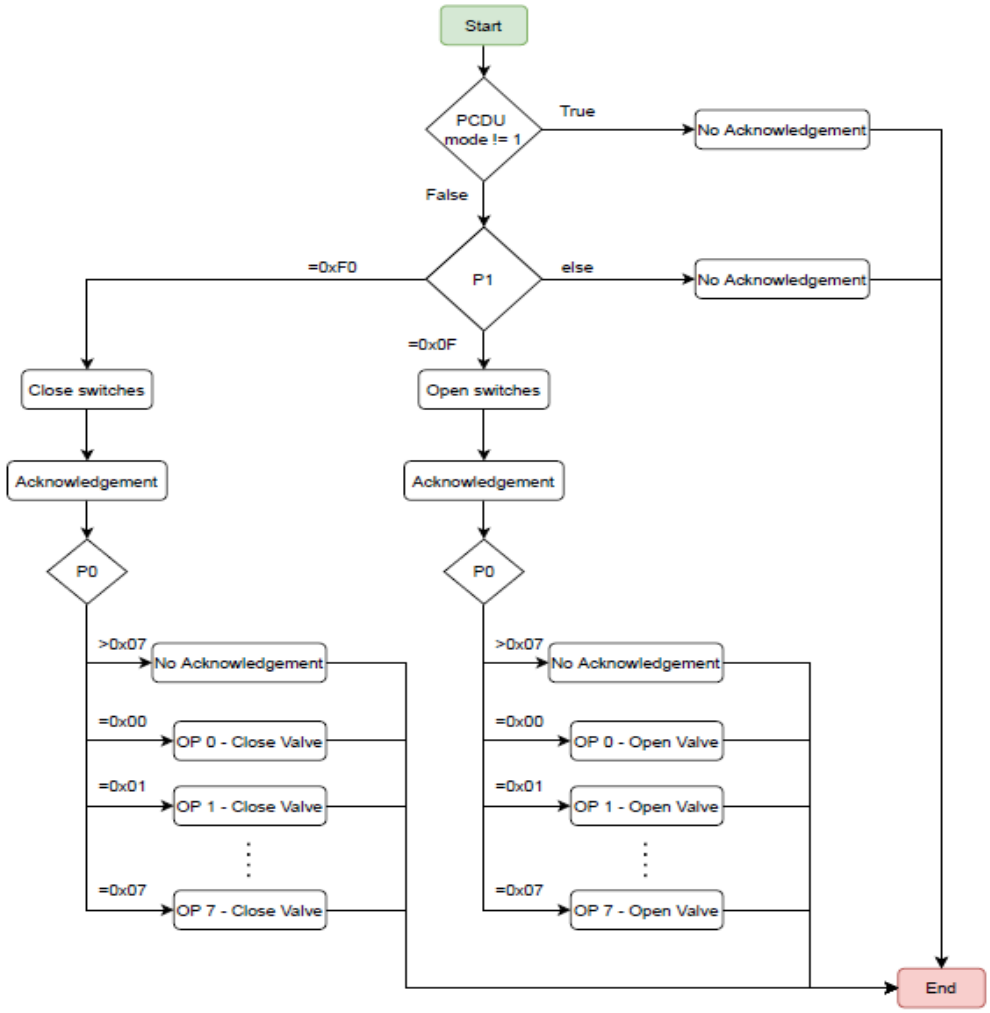


Figure 6.1: Flowchart for epsTMTCControl()[11]

6.2 IOB model

Several aspects of the IOB model were adapted for the Gecko model. The first step was to add the I2C serial line ports for the input and output interfaces. Changing the external model requires changes to the IOB model code. Inside the main IOB file, two functions need to be edited: the *initI2C()* and the *createI2CInterface()* functions. The *initI2C()* function initialises all devices that use I2C serial line ports[40]. The entry for the Gecko is shown below:

initI2CLine(_I2C_GECKO_Rx, _I2C_GECKO_Tx, "_I2C_GECKO", GECKO_ID)

The parameters refer to the input port and output port for the Gecko, the name of the I2C connection and the specific ID found in the mutex registry file. The *initI2CLine()* function adds the Gecko parameters to a list of I2C devices. The *createI2CInterfaces()* function creates an RMAP (Remote Memory Access *Protocol*) I2C interface for the Gecko[40] . It has the following form:

CREATE_DEFAULT_RMAP_I2C_INTERFACE(GECKO, "GECKO_I2C_Interface")

The parameters refer to the component name and the interface name. Additional changes were made to the IOBInterfaceMacros, IOBMutexRegistry and the IOBInterface files. Inside the IOBInterfaceMacros file, the Gecko definition of the RMAP-address pairs is added[40]. It has the following structure:

DEF_RMAP_ADDRESS_DATA(GECKO, 0x01, 0x81A4, 80, 32)

The parameters refer to the component name, the extended address, the address for the component, the size of the received data, the size of the transmitted data. The IOBMutexRegistry file defines all mutexes used to synchronize the data via the IOB interfaces[40]. For the Gecko, it has the following structure:

#define GECKO_IF_MUTEX IOBRMAPPacketHandler::_IOBInterfaceMutex32

Inside the IOB interface file, additions were made for the I2C interface class. In the *I2CInterface::rmapWrite()* function, the original code only worked for the MGTBoard (Magnetic Torquer Board). A re-work was constructed for additional I2C devices to be added. The re-work separated the code that only worked for the MGT and other I2C devices.

The re-work code checks to see if the packet address is for the MGTBoard. If it is not, then the code shown in Figure 6.2 is executed. If a write command is received i.e., read equals zero, the commands are sent to the device. If a read command is received, the *getReplyFromDevice()* is called. The function *getReplyFromDevice()* checks if data has been received from an I2C device. The re-work allowed I2C packets received from the OBC to have the structure shown in Table 6.5.

Table 6.5: I2C packet format

Address byte	Read byte	Write byte	I2C data	I2C data	I2C data	I2C data	I2C data
0x33/0x32	0x50	0x05	0x00	0x00	0x00	0x00	0x00

The address byte is important for the sending and receiving data. This is the write and read commands are only executed using the RMAP write function. If the address of the packet is 0x33, this means a read command and if its 0x32, it is a write command. The data size that is read is indicated by the read byte and the data size that is written is indicated by the write byte. The I2C data bytes are for the data to be written to the I2C device. The Gecko uses the address values shown in Table 6.5.

```
/* Packet to device:
 * | byte 0     | 1            | 2            | ...          |
 * | I²C data 0 | I²C data 1   | I²C data 2   | I²C data ..  |
 * | <var>      | <var>        | <var>        | <var>        |
 */
if(read == 0x00){
    I2CpacketLength = writeBytes;
    uint8_t I2Cpacket[I2CpacketLength];

    for (uint8_t i=0; i<writeBytes; i++) {
        I2Cpacket[i] = Data_[i+I2C_HEADER_LENGTH];
    }
    _TxInterface_ptr->RMAPWrite_noMutex(addr_, I2Cpacket, I2CpacketLength);
}
else if (read == 0x01) {
    return getReplyFromDevice(address, readBytes);
}
```

Figure 6.2: I2C Interface re-work code

6.3 ForceAndTorqueSum Model

The ForceSum model has input interfaces for all the forces acting on the satellite. It adds up all the forces to get a total force and calculates the total angular momentum acting on the satellite. The input interface for the thrust value was added.

6.4 Dynamics Model

The Dynamics model simulates the space environment and the satellites dynamics as it travels through its determined orbital path. The model calculates the position, acceleration, velocity, and altitude of the satellite relative to Earth. The only acceleration acting on the satellite is the J2 acceleration and the acceleration generated from the thrust. Additions were made for the longitude and latitude values calculated to be connected to the Gecko camera.

6.5 OBC and OBSW

To simulate the operation of the Gecko and the EPS with the On-board computer, the on-board flight software (OBSW) is adapted for both simulated models. The first addition to the OBSW is to edit the IOB board addresses and the PCDU switches values to include the Gecko and EPS components. These changes are reflective of the changes done in the IOB and PCDU simulated models. The datapool of the OBSW was also updated to handle the Gecko and EPS models.

Other aspects of the OBSW that were adapted were the PCDU handler and a handler was created for the Gecko. Handlers allow the operators to control the simulated models from the OBC through the Mission Control System. The OBSW then processes the commands and send the commands to the specified handler.

6.5.1 *PCDU handler*

The PCDU handler controls all commands sent and telemetry data received from the PCDU model. First additions were made to include the updated values for the number of fuses, switches, and external temperature sensors. Many values were hard-coded, and this has been changed to handle any number of fuses, switches, and external temperatures sensors. The command for the EPS had to be added. The building of the packet for the command was slightly different to the normal PCDU commands.

6.5.2 *Gecko handler*

The handler communicates with the Gecko device receiving telemetry data and sending commands. All the commands for the Gecko are defined in the handler. The constructor for the handler defines all the parameters of the Gecko. The constructor structure is as follows:

> *GeckoHandler(object_id_t setObjectId, uint32_t parent, uint8_t deviceId, uint8_t deviceSwitch, uint32_t thermalStatePoolId, uint32_t thermalRequestPoolId);*

The parameters refer to the Gecko handler address, the parent for the Gecko is the payload subsystem, the deviceId is the IOB board address, the device switch is the switch number defined in the PCDU and the last two parameters are for the thermal indications of the camera.

The Gecko handler has the following functions:

- *doStartUp()*
- *doShutDown()*
- *doTransition(Mode_t modeFrom, Submode_t subModeFrom)*
- *getTransitionDelayMs(Mode_t modeFrom, Mode_t modeTo)*
- *buildNormalDeviceCommand(DeviceCommandId_t * id)*
- *buildCommandFromCommand(DeviceCommandId_t deviceCommand, const uint8_t* commandData, size_t commandDataLen);*
- *fillCommandAndReplyMap()*
- *scanForReply(const uint8_t *start, uint32_t len,DeviceCommandId_t *foundId, uint32_t *foundLen);*

- *interpretDeviceReply(DeviceCommandId_t id, const uint8_t *packet)*
- *buildTransitionDeviceCommand(DeviceCommandId_t* id)*
- *modeChanged()*
- *setNormalDatapoolEntriesInvalid()*
- *addressRead(uint8_t address)*
- *addressWrite(uint8_t address, const uint8_t *commandData, size_t commandDataLen)*
- *reg0x02Analysis(const uint8_t *packet)*
- *setReg0x02(const uint8_t *cmdData)*

The first 12 functions are virtual functions which are all declared in the DeviceHandlerBase files. The *doStartUp()* and *doShutDown()* functions are called when the handler is started or ended. They are meant to switch on and switch off the device automatically.

The *doTransition()* function changes the mode of the handler which relates to the mode of the device. The *getTransitionDelayMs()* function has the required delay the device needs to changes modes. The *buildNormalDeviceCommand()* function generates commands that are sent when the handler is in normal mode.

The *buildCommandFromCommand()* is for when commands are built into the correct structure. The Gecko command packets generated are 8-byte. The first three bytes are for the I2C packet header and the remaining bytes are the commands for the Gecko. The *fillCommandAndReplyMap()* function sends the commands and defines the response length from the device. The *scanForReply()* and *interpretDeviceReply()* are called when the OBSW has received a response from the device. The *scanForReply()* checks if the reply is valid and the *interpretDeviceReply()* checks which response is received and performs operations accordingly. If a telemetry response is received, the data is either added to the datapool or it is sent to the ground station for analysis.

The *buildTransitionDeviceCommand()* is called when the device mode is changed. The *modeChanged()* is empty and the *setNormalDatapoolEntriesInvalid()* is called if the received telemetry data is not valid.

The *addressRead()* function develops the command structure for sending a read command to the Gecko. The *addressWrite()* function is used to set bits in the desired register depending on the commands received from the ground station. The *reg0x02Analysis*() function analyses the register 0x02 data received from the Gecko. The *setReg0x02()* function set the bits in register 0x02 depending on the commands received.

7. Gecko Model Verification

The verification of the Gecko Camera and the EPS model are divided into multiple stages. The verification tests start at the software level and as more parts of the model are included it leads to the "Controller in the loop" stage which involves the OBC and OBSW.

The test cases for the models are written in Java and are performed in the SimOPS environment of the SimTG. SimOPS has many predefined commands to aid in the simulation testing process and simply the test script code. The test script format is the same for both models with minor differences. All model functionality tests use the format shown in Figure 7.1. To use the SimOPS environment, the SimOPSTestBase is included as well as the function *sequence()*. This allows the predefined commands to be used without issues. Inside the *sequence()* function is where the object is created and the commands for the test are written.

```
public class GeckoCamera extends SimOPSTestBase {

    String _library = "swarmFLP";

    public static void main(String[] args) {
        start(new GeckoCamera());
    }

    @Override
    public boolean sequence() throws SimopsException {

        sim.run("GeckoCameraSim");
        //generateObjectListsFromXML("tests/SimModelTest_5_00_00.xml");
        instantiateObjects(_library);
        scheduleObjects();

        sim.createObject(_library, "PowerSupply", "PS");
        sim.createObject(_library, "DataSource", "DS");
        sim.createObject(_library, "NCUBoard", "NCUBoard");

        sim.createLine(_plhs, "Powerline", "PS", "NCUBoard", "VoltageSupply", "NCU_pwr");
        sim.createLine(_slhs, "s1", "DS", "NCUBoard", "Data_Tx", "I2C_Rx");
        sim.createLine(_slhs, "s2", "DS", "NCUBoard", "Data_Rx", "I2C_Tx");

        DataSourceControl i2c_Line = new DataSourceControl(sim, "DS");

        sim.activateMethodCyclically("NCUBoard.nano", 0.1);

        double csvPeriod = 0.01; // sec
        double csvOffs = 0.0;

        String filename = "test/Subsystems/payload/gecko/nanoCUBoard/testresults/GeckoCamera.csv";
        CSVsampler CSV = new CSVsampler(csvPeriod, csvOffs, filename, sim, sys);

        sim.init();

        // parameters to observe
        CSV.addToCSVsampler("NCUBoard.Param.inputVoltage");

        // Commands added here

        _success = true;
        return _success;
    }
```

Figure 7.1: Basic Gecko test script structure

The *run()* command creates the test instance. The *instantiateObjects()* and *scheduleObjects()* creates all the objects and lines defined in the project and connects them to their defined step timesteps. The model classes are created using the *createObject()* function. Depending on the functions that need to be tested, powerlines and serial lines can be created using the *createLine()* function. The *createLine()* connects the output interface to an input interface between two models.

The DataSourceControl creates an instance object for data. This allows the input of data to be sent to the model and receive data from the model. The *activateMethodCyclically()* function defines the time the *step()* is called periodically. The *CSVsampler()* creates a CSV file where parameters of the internal model can be stored and checked. After the *init()* is called. This function starts the simulation of the model. The *addToCSVsampler()* function saves all the parameters observed to the CSV. These parameters can also be serial lines or powerlines. The data is saved when the *step()* is called. The commands are executed and the return line stops the simulation.

7.1 Software Level Functional Testing

The individual models of the Gecko Camera are tested separately using the format shown in Figure 7.1. Each model has its own test cases to test its functionality.

7.1.1 *NanoCUBoard*

The test cases for the NanoCUBoard are listed in

Table 7.3. In test case one, it checks that the correct mode of operation is determined depending on the input voltage value. Test case two and three, the erase and read commands are tested with the flash. The data transfer to the satellite is tested in both single and dual mode. Test case four and five check the imaging and image to flash commands when one and two pictures are captured. The last test case is for sending the telemetry data packet over the I2C connection. Table 7.1 and Table 7.2 provide definitions for the mode and status of the NanoCUBoard for the test cases simulation results.

Table 7.1: NanoCUBoard Mode Definition

Board Mode Value	Board Mode
-1	Dead
0	Off
1	On
2	Erase
3	Read Image
4	Pre-imaging
5	Imaging
6	Image to Flash

Table 7.2: NanoCUBoard Status Definition

Status Value	Status
0	Idle
1	Busy
2	Done
3	Sent Image to Satellite
4	Wait

Table 7.3: NanoCUBoard testcases

Test No.	Description
1	Different input voltage values to check mode of operation.
2	The erase and read commands and flash responses with data transfer over the serial lines in single mode. This is not the actual flash model, but rather expected responses are inputted.
3	The read command and the response from the flash with data transfer over the serial lines in dual mode. This is not the actual flash model, but rather the expected responses are inputted.
4	The imaging and image to flash commands with the image sensor and flash. This is not the actual flash and image sensor models, but rather the expected responses are inputted. One picture is captured.
5	The imaging and image to flash commands with the image sensor and flash. This is not the actual flash and image sensor models, but rather the expected responses are inputted. Two pictures are captured.
6	The telemetry data packet received over the I2C serial lines.

The simulation results are for test case one, two and four. These test cases cover the full simulation of the NanoCUBoard and its' mode of operation. Other test cases are very similar with the only exceptions being the dual line mode and capturing multiple pictures during imaging mode. The simulation result for test case one is shown Figure 7.2. The corresponding mode values shown in the graph relate to the values in Table 7.1 for each different mode. When the applied voltage is less than 5V, the board mode is off (shown in the graph as value zero) and as the applied voltage is 5V, the board mode is on and when the applied voltage is above 5V, the board mode is dead (shown in the graph as value minus one).

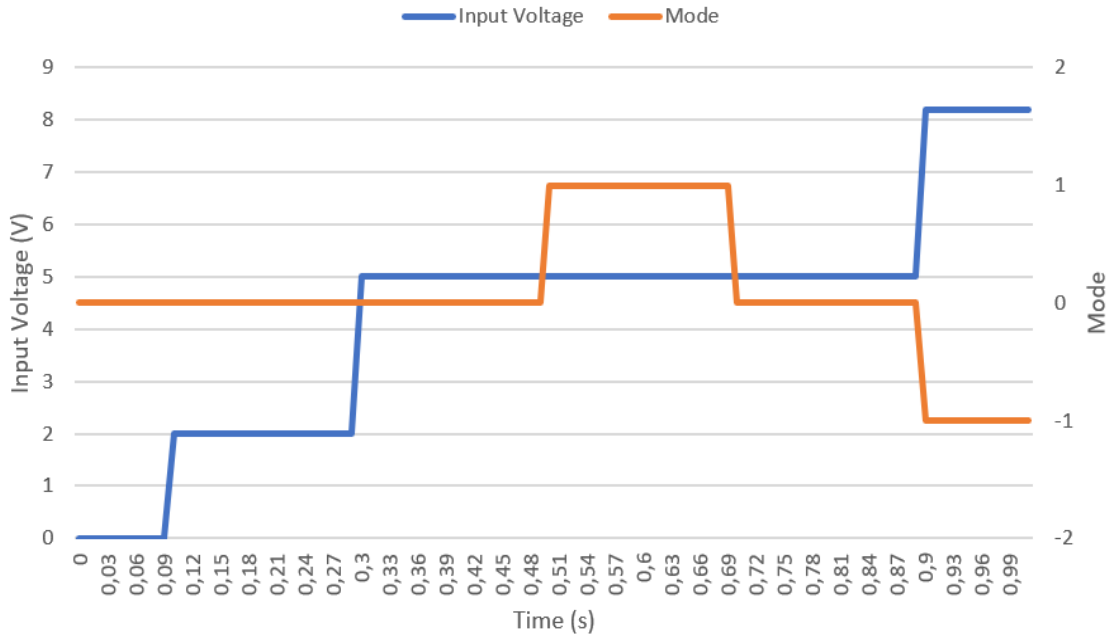


Figure 7.2: Test case one simulation result

The simulation result for test case two is shown in Figure 7.3. This case simulation relates to the erase and read mode of the NanoCUBoard. The corresponding mode values shown in the graph relate to the mode definitions in Table 7.1. At about 0.08 seconds, register 0x0A is set to turn on the Gecko and the board mode changed to on. This only occurs if the applied voltage is 5V. When the mode changed to two, the start erase bit was set. When the mode changed to three, the start read bit was set. After each mode change, the board returned to idle mode waiting for the next command.

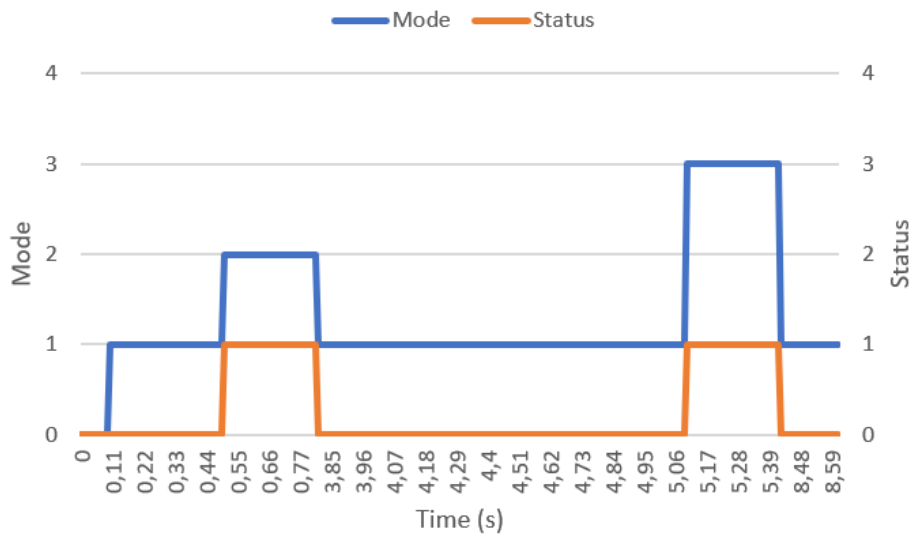


Figure 7.3: Test case two simulation result

The simulation result for test case four is shown in Figure 7.4. This test simulation shows all the steps required to get the NanoCUBoard into pre-imaging mode to imaging mode and then image to flash mode. The corresponding mode values shown in the graph relate to the mode definitions in Table 7.1. The board is immediately commanded to on mode. When the start pre-imaging bit is set, the mode is changed to mode four (pre-imaging mode) and status is busy, during this mode, the verification bits are set to confirm sensor is configured and on. The status then changes to idle and the board mode changes to mode 5 (imaging mode).

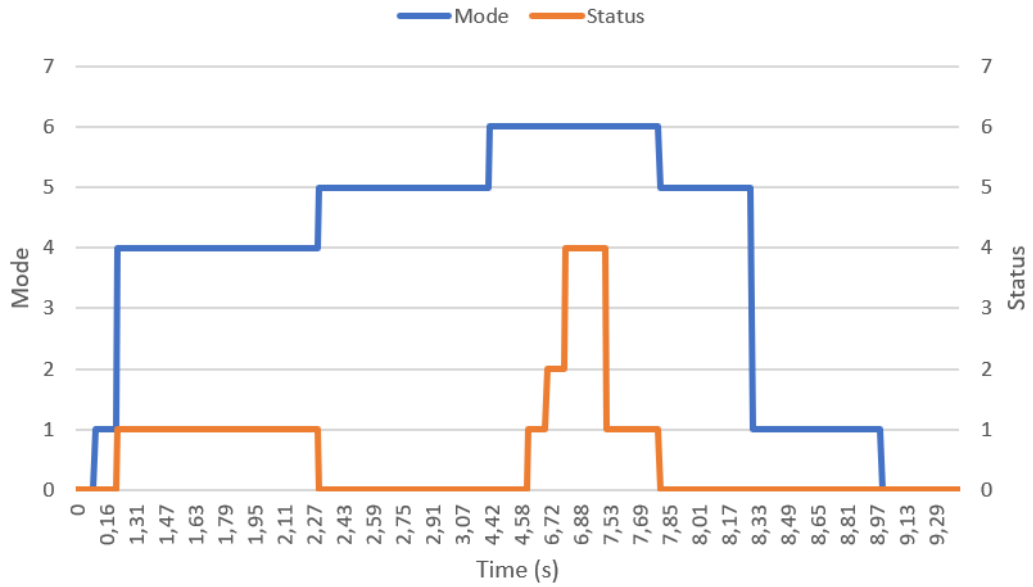


Figure 7.4: Test case four simulation result

When the start image to flash bit is set, the board mode changes to mode 6(image to flash mode. and status to busy. The status then changes to done when reading the image data and then to sentImageDataToSatellite and back to busy then idle when the sensor sends an imaging done response. The three simulation results demonstrate overall the working model of the NanoCUBoard.

7.1.2 *Flash*

The test cases for the flash are listed in Table 7.6 and the definitions for the flash sensor status and mode for the simulation results are listed in Table 7.4 and Table 7.5. Test case one is very similar to test case one for

the NanoCUBoard. Test case 2, 3 and 4 focus on the flash models response to the erase, read and write commands.

Table 7.4: Flash Sensor Status Definition

Status Value	Status
0	Idle
1	Busy
2	Done

Table 7.5: Flash Mode Definition

Flash Mode Value	Flash Mode
0	Off
1	On
2	Erase Image
3	Write Image
4	Read Image

Table 7.6: Flash testcases

Test	Description
1	Mode determined based on power received.
2	The erase command of an image.
3	The read command with image data output.
4	The write command with image data input.

The simulation results for test case two to four is shown in Figure 7.5 and Figure 7.6. Each spike in Figure 7.5 corresponds to a mode change. The first spike is for the erase mode, the second is for the read mode and the third is for write mode. Figure 7.6 shows the flash voltage and current telemetry data being sent to the NanoCUBoard.

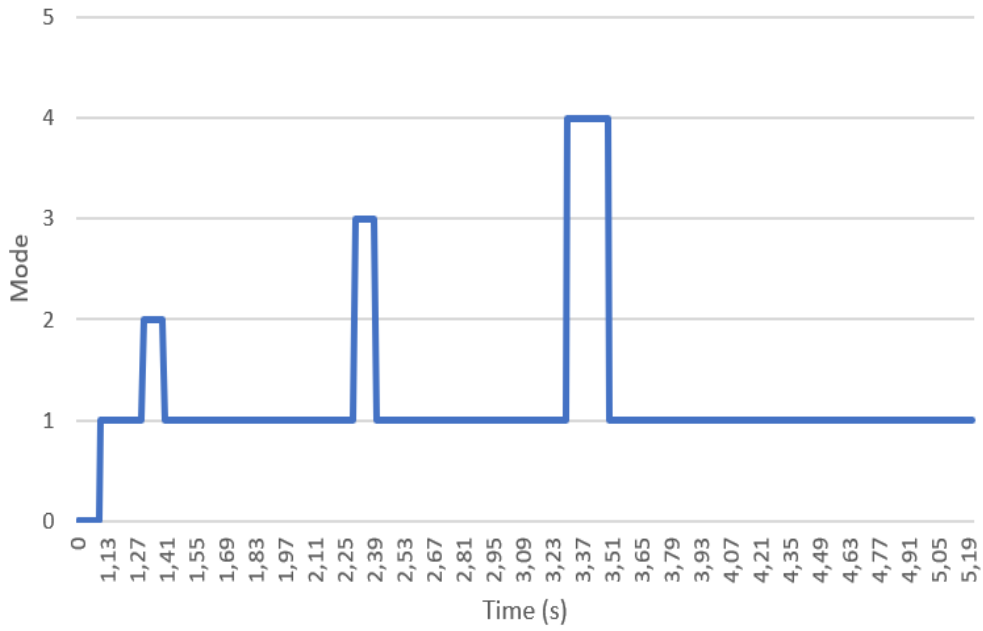


Figure 7.5: Test case 2, 3 and 4 simulation results

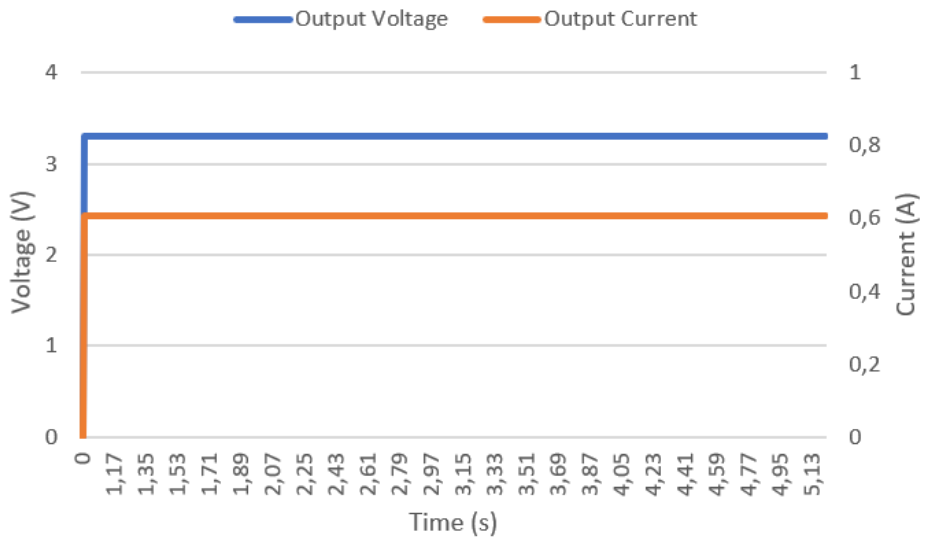


Figure 7.6: Voltage and current sensor data

7.1.3 *Image sensor*

The test cases for the image sensor are listed in Table 7.10. Table 7.7 displays the different statuses of when an image is being sent from the image sensor to the NanoCUBoard. Table 7.8 displays the status of the image sensor. It is either on or off. Table 7.9 shows the different modes of the image sensor when the sensor is on. Test case one is very similar to test case one for the NanoCUBoard. Test case two checks that the correct mode is selected based on the command sent. Test case three checks take picture command when the configuration for the image sensor is one or two images. It also checks if the data is sent on the serial line for both configurations.

Table 7.7: Image Sending Status Definition

Image Status Value	Image Status
1	Idle
2	Busy
3	Wait
4	Done

Table 7.8: Image Sensor Status Definition

Sensor Status Value	Sensor Status
0	Off
1	On

Table 7.9: Image Sensor Mode Definition

Sensor Mode Value	Sensor Mode
-1	Dead
0	Standby
1	Configuration
2	Idle
3	Image to Board

Table 7.10: Image sensor testcases

Test	Description
1	Image Sensor mode determined based on power received.
2	The changing of the image sensor configuration settings when the sensor status is off and a configuration command is received.
3	The changing of the sensor configuration settings when the sensor is in idle mode and a configuration command is received.
4	The take picture command with image data sent with the configuration settings set for the sensor to take three images.

The simulation results for test case two is shown in Figure 7.7. In the upper diagram, when the turn on command is received, the mode is changed to idle and when the turn off command is received, the mode is changed back to standby. The sensor status being one indicates its ready to be used for capturing images (i.e., on) and if its zero, the sensor is off. The spike in the middle indicates the sensor has been commanded to configuration mode.

When the configuration command is sent, the mode is changed to configuration. Once configured, the mode is changed back to off. When the sensor is turned on, a configured response is received confirming the sensor is trained as shown in the lower diagram of Figure 7.7. A sensor response of one, indicates the sensor is not configured and a sensor response of three, indicates the sensor is configured.

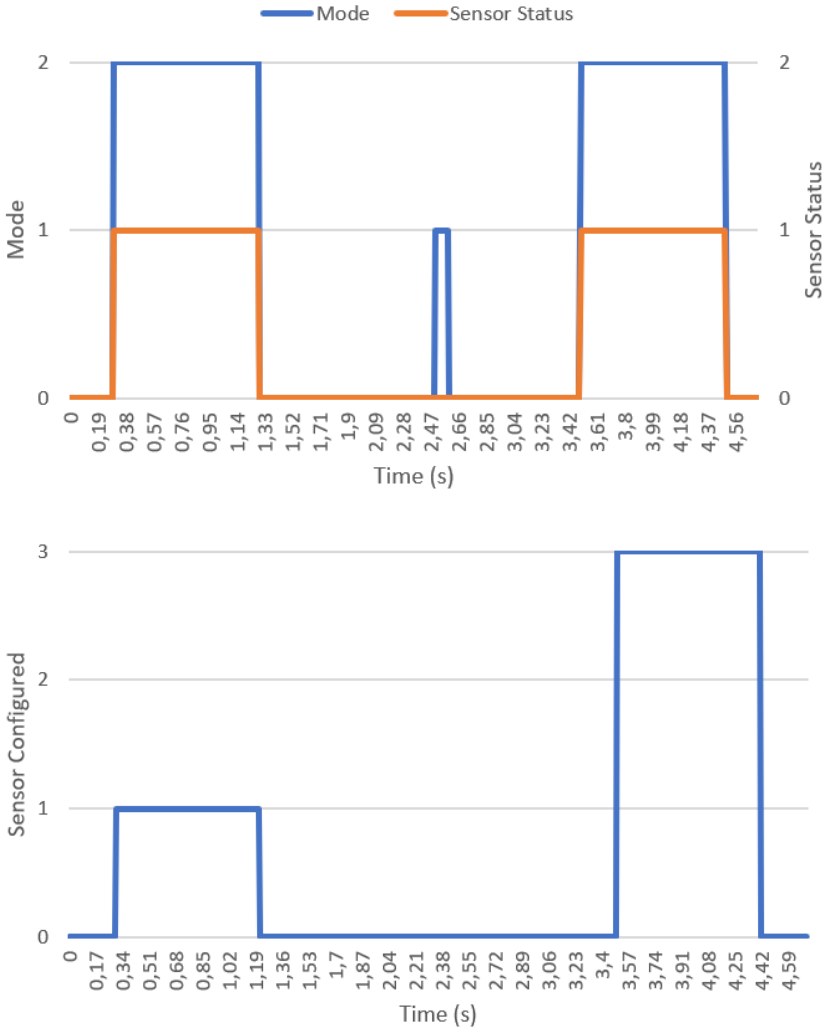


Figure 7.7: Test case two simulation results

The simulation results for test case three is shown in Figure 7.8. In the upper diagram to best represent when a configure command is received during idle mode, the mode of the sensor is changed to configuration for one timestep then back to idle. The settings have not changed during this mode change. When the sensor is switched off, the settings of the sensor are changed. When the sensor is switched on, the sensor sends a configured response as shown in the lower diagram.

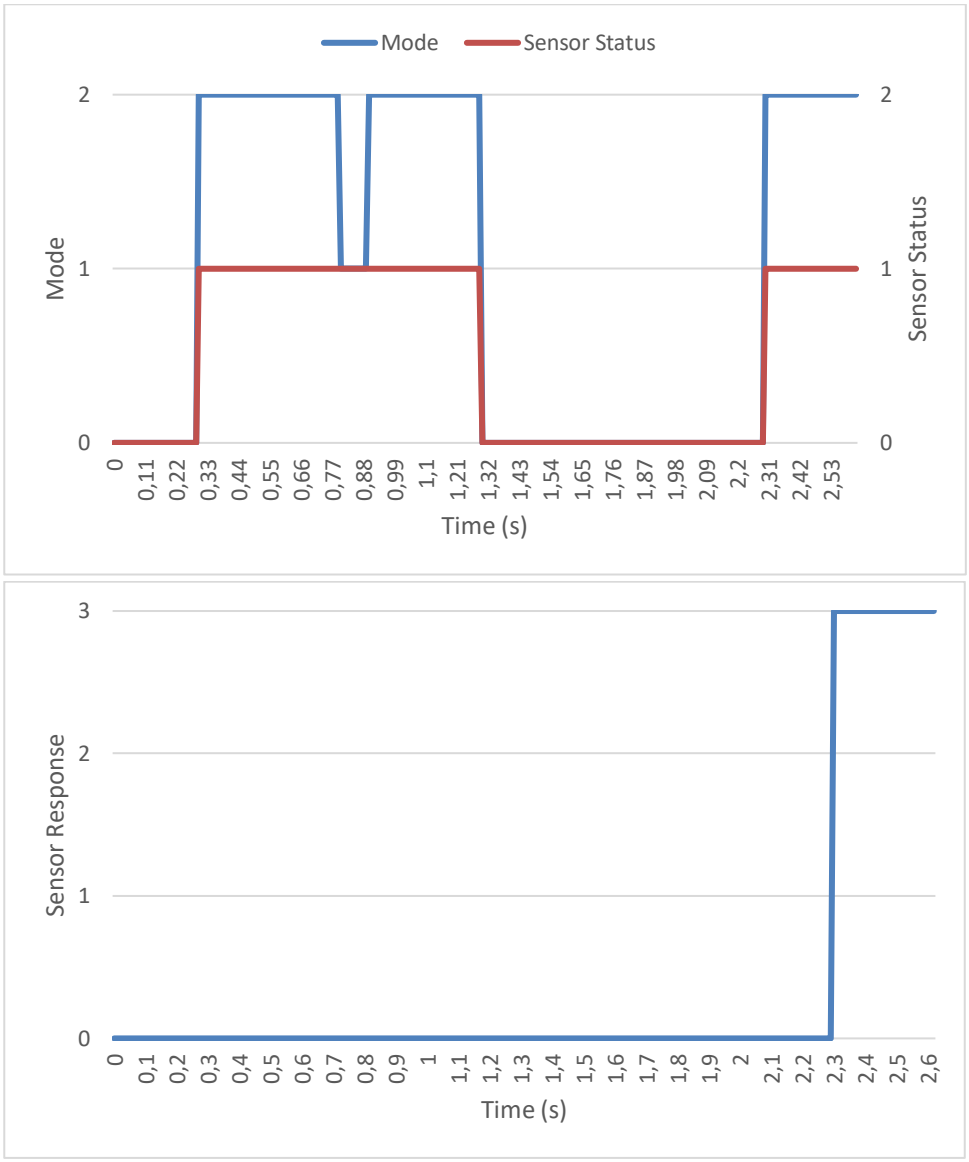


Figure 7.8: Test case three simulation results

The simulation result for test case four is shown in Figure 7.9. The sensor is first configured for three images. After a few milliseconds, the sensor is turned on and then the image to board command is sent. The sensor board is changed to image to board mode and the three images are captured. After the sensor is done sending the data, the mode automatically changes back to idle.

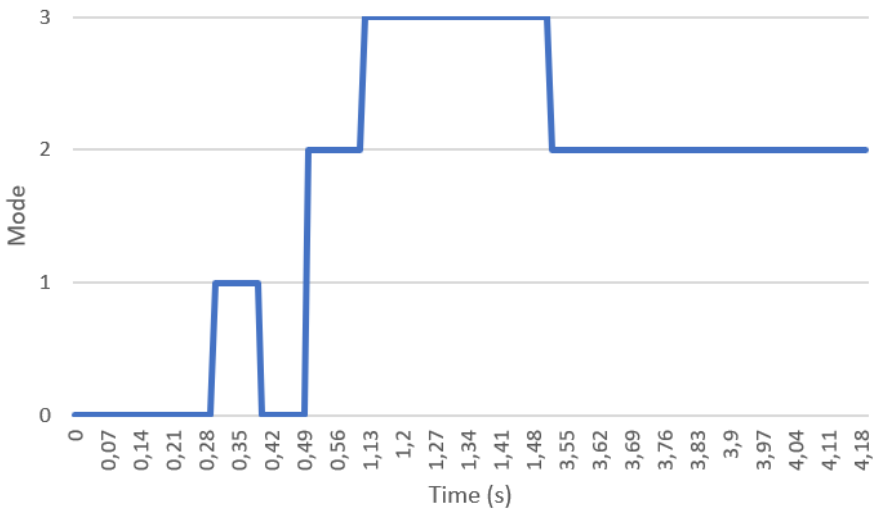


Figure 7.9: Test case four simulation results

Because the images are captured very quickly, a clear representation of the status cannot be shown but instead a textual representation is shown in Figure 7.10. The figure shows the number of pictures taken and also when the imaging done response is sent.

```
ImgSen_X_00_C03Sim : imgsen: Camera initialised
ImgSen_X_00_C03Sim : Image sensor response sent
ImgSen_X_00_C03Sim : Image sensor response sent
ImgSen_X_00_C03Sim : Picture Number 1
ImgSen_X_00_C03Sim : Image sensor response sent
ImgSen_X_00_C03Sim : Picture Number 2
ImgSen_X_00_C03Sim : Image sensor response sent
ImgSen_X_00_C03Sim : Picture Number 3
ImgSen_X_00_C03Sim : Sensor Imaging done
ImgSen_X_00_C03Sim : Image sensor response sent
```

Figure 7.10: Test Script output

7.1.4 *IOB and Gecko testing*

After successful testing of the Gecko models, the Gecko models were tested with the Input/Output-Board (IOB) model. To test the Gecko with the IOB model, the Power Subsystem (PSS) models need to be included. The PSS provides power to the Gecko and IOB and it includes the PCDU, the battery and the solar panel models.

The test script used is shown in Figure 7.11. The testing of the IOB and PCDU is complex because the script has to send commands as though it's the OBC. A special *integrationTest* file was created to be able to send the Gecko commands via the IOB. The file takes the Gecko commands and creates the RMAP packets that the IOB reads and executes. The object *PCDUControl* allows the PCDU to be controlled through the test script.

```java
public class Gecko_X_00_C02 extends SimOPSTestBase{

    String _library = "swarmFLP";

    integrationTest _integrationTest;

    String[] _InjectLines = new String [integrationTest._nSources];
    String[] _CollectLines = new String [integrationTest._nSinks];

    PCDUControl _PCDU_0;

    public static void main(String[] args) {
        start(new Gecko_X_00_C02());
    }

    @Override
    public boolean sequence() throws SimopsException {
        sim.run("Gecko_X_00_C02Sim");

        String[] Board = {"IOB_0","flash","NCUBoard","imagesensor","PCDU_0","battery_0","battery_1",
                "battery_2","solarpanel_0","solarpanel_1","solarpanel_2","balancingBoard_0","balancingBoard_1","balancingBoard_2"
                ,"Heater_battery_0","Heater_battery_1"};

        generateObjectListsFromXML("tests/SimModelTest_5_00_01.xml",Board);
        instantiateObjects(_library);
        scheduleObjects();

        sim.activateMethodCyclically("NCUBoard.nano", 0.1);
        sim.activateMethodCyclically("flash.f", 0.1);
        sim.activateMethodCyclically("imagesensor.f", 0.1);

        sim.enableSchedulableObjectStatistics();
        int mode = integrationTest._simtg_mode;
        _integrationTest = new integrationTest(sim, this, _library, Board, mode);
        createDataLines(Board[0]);

        double csvPeriod = 0.01; // sec
        double csvOffs = 0.0;
        String filename = "test/Subsystems/payload/gecko/geckoTest results/Gecko_X_00_C02.csv";
        CSVsampler CSV = new CSVsampler(csvPeriod, csvOffs, filename, sim, sys);
```

```
        _PCDU_0 = new PCDUControl(CSV, sim, this);

        sim.writeBoolean("solarpanel_0.In.eclipse", true);
        sim.writeBoolean("solarpanel_1.In.eclipse", true);
        sim.writeBoolean("solarpanel_2.In.eclipse", true);

        CSV.addToCSVsampler(Board[0] + ".Param.mode");
        CSV.addToCSVsampler(Board[4] + ".Param.pcduMode");
        CSV.addToCSVsampler(Board[2] + ".Param.powerStatus");
        CSV.addToCSVsampler(Board[2] + ".Param.inputVoltage");
        CSV.addToCSVsampler(Board[2] + ".Param.mode");

        sim.init();
        sim.writeInt(Board[0] + ".Param.messageLevel", 255);
        CSV.writeData();

        sim.writeInt("PCDU_0.Param.deploymentFlag", 0x55);
        sim.writeInt("PCDU_0.Param.initialHoldingTimeRP", 0xFFFF);

        _success = true;

        _PCDU_0.bootPCDU();

        // wait for PCDU to power up IOB
        msSleep(500);
        CSV.start();
        sim.start();

        _integrationTest.test();

        CSV.stop();
        sim.stop();

        sim.disconnect();
        _success = true;
        return _success;
    }

    private void createDataLines(String board_) throws SimopsException {

            _InjectLines[0] = sim.createLine(_slhs, "SpwSourceLine", integrationTest._SourceNames[0], board_, "Data_Tx", "Spw_Rx");

            _CollectLines[0] = sim.createLine(_slhs, "SpwSinkLine", board_, integrationTest._SinkNames[0], "Spw_Tx", "Data_Rx");

    }
```

Figure 7.11: Gecko and IOB test script

The *generateObjectListsFromXML()* function is added, because all the models have many input and output interfaces. The *generateObjectListsFromXML()* reads in an xml file and generates and creates all the connections between the models such as the powerlines, data lines and serial lines. The xml file contains a list of all the models that need to be simulated and their internal parameters.

The *enableSchedulableObjectStatistics()* function sends the commands created randomly to the IOB to simulate full functionality. The integrationTest object is created with the set values and the same applies for the PCDUControl. The *writeBoolean()* is able to set true and false to the internal parameters of a model. The solar panels are set to true because they only deploy when the satellite is in eclipse. The *writeIn()* function is able to set any integer value to an internal model parameter. Both functions tell the PCDU that the solar panels have been deployed and the time taken for them to deploy is set. The *bootPCDU()* functions sends the boot commands to the PCDU to switch it on. A delay is created for the PCDU to start up and the *test()* function starts sending the PCDU and Gecko commands to the IOB. Once all the commands have been executed, the test script ends.

7.2 Mission Control Testing

The last commanding of the Gecko is to include the OBC simulated model and OBSW and all models required for satellite operation. The Software verification facility (SVF) uses the CCS5 as the control software on the FLP Generation 2.

The communication between a spacecraft and ground station uses the Consultative Committee for Space Data Systems (CCSDS) communication standards. The telecommand (TC) packets are segmented and framed and converted to Command Link Transmission Units (CLTU) in the ground segment. After the modulation of the CLTUs, they are transferred to the spacecraft. The CCSDS boards extract the TC frames from the CLTUs which are processed by the OBSW to unpack the TC packet. The telemetry (TM) packets are framed and converted to Channel Access Data Units (CADU) in the spacecraft. Thereafter, the CADUs are modulated and they are transferred to the ground segment to be unpacked.[41]

The FLP2 uses a database called the Mission Information Base (MIB) which is a collection of telecommands (TC) and telemetry (TM) commands specific to the mission and subsystems. The database includes definitions for all TC/TM packets, the information of TC and TM parameters and many further features like graphical displays.[41]

The definitions for the Gecko commands had to be added to the MIB. The FLP uses the Packet Utilization Standard (PUS) for its TC and TM commands. The different PUS services define how the TC and TM commands are handled by the ground or on the satellite. There are 19 predefined services and each is subdivided into subservices which are either a request or a report to/from a service provider. More services can be added for specific missions.[41]

PUS service 2 includes the main functions for Gecko commanding. The service is used for direct device commanding but also for reporting of TM data which is not part of the housekeeping data report. Subservice (2,130) [42] is for direct commanding and subservice (2,135) [42] is for the reply packet of the device.

An example of subservice (2,130) for commanding the Gecko is shown in Figure 7.12. The first line is the TC name “GCK00000” with the target APID address 53. The address is for the OBSW. Next the subservice type and subtype are mentioned. In the TC command packet, they are 2 and 130. In the lower part of the figure, the TC structure is shown. The first ten entries are for the packet header including the TC subservice numbers as entries 7 and 8 in hexadecimal. The rest of the TC structure defines the TC targeted object ID and the command ID. Hence, the object ID “44501300" is assigned to the Gecko handler address and the command ID is “000000C” which is for requesting register 0x0C data.

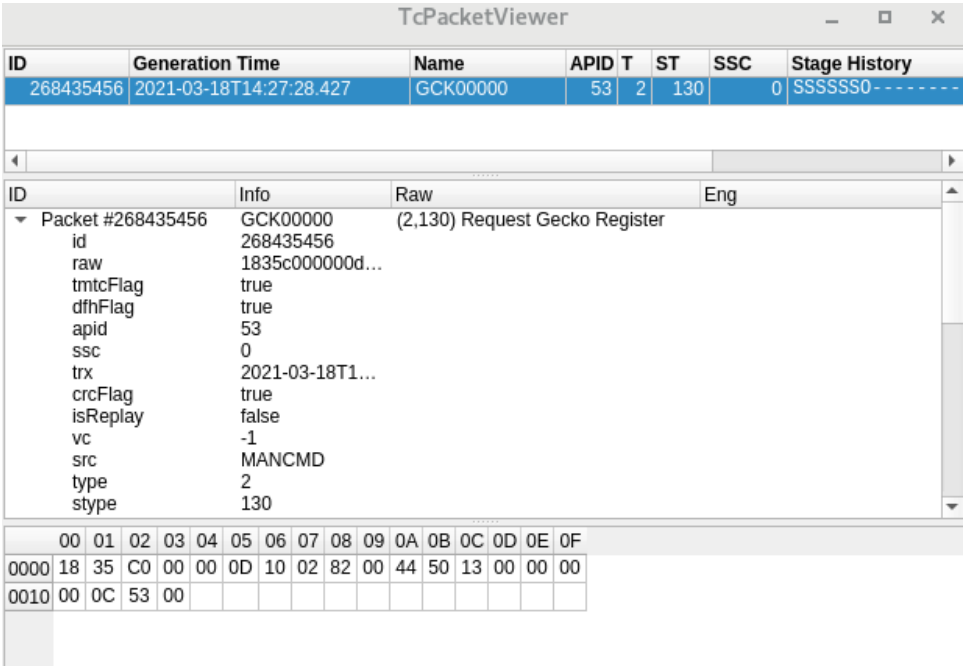


Figure 7.12: Subservice (2,130) Telecommand in CCS5

An example of subservice (2,135) for the Gecko telemetry was created but due to the MGM handler, the OBSW did not finish its boot sequence to be able to test the Gecko telemetry packet received from the OBSW. In contrast to the telecommand, the TM does not include any data and only consists of the packet header. The test cases for the mission control testing were similar to the test cases done in
Table 7.3.

8. EPS Model Verification

The verification of the EPS model follows the same steps as the Gecko Camera. The test script of the EPS is very similar to the Gecko but there are a few differences as shown in Figure 8.1. The difference is the model is created using the *createObject()* function and only the powerlines are created using the *createLine()* function. A line is created for each nominal and redundancy component. The main observation parameters are for the operation point and the thrust value for all test cases with additional parameters required where necessary. All the data is stored in a csv file for analysis.

To turn on each component in the EPS, the *powerLineSetVoltage()* was used. The function needs two arguments. The first argument is the line name (e.g. PPU0, NEUT0) and the second argument is the applied voltage to the line, e.g. 28V, 0V. The EPS components need 28V applied except for the neutraliser which requires 8V to simulate the component has been turned on.

```
public class testEPS extends SimOPSTestBase {

    String _library = "swarmFLP";

    public static void main(String[] args) {
        start(new testEPS());
    }

    @Override
    public boolean sequence() throws SimopsException {
        sim.run("testEPSSim");

        sim.createObject(_library, "eps", "eps");
        instantiateObjects(_library);
        scheduleObjects();

        String  PPU0 = sim.createLine(_plhs ,"EPS_Pwr_PPU0","","eps","","Pwr_PPU0");
        String  PPU1 = sim.createLine(_plhs ,"EPS_Pwr_PPU1","","eps","","Pwr_PPU1");

        String  NEUT0 = sim.createLine(_plhs ,"EPS_Pwr_NEUT0","","eps","","Pwr_NEUT0");
        String  NEUT1 = sim.createLine(_plhs ,"EPS_Pwr_NEUT1","","eps","","Pwr_NEUT1");

        String  VLV0N = sim.createLine(_plhs ,"EPS_Pwr_VLV0N","","eps","","Pwr_VLV0N");
        String  VLV1N = sim.createLine(_plhs ,"EPS_Pwr_VLV1N","","eps","","Pwr_VLV1N");
        String  VLV2N = sim.createLine(_plhs ,"EPS_Pwr_VLV2N","","eps","","Pwr_VLV2N");
        String  VLV3N = sim.createLine(_plhs ,"EPS_Pwr_VLV3N","","eps","","Pwr_VLV3N");

        String  VLV0R = sim.createLine(_plhs ,"EPS_Pwr_VLV0R","","eps","","Pwr_VLV0R");
        String  VLV1R = sim.createLine(_plhs ,"EPS_Pwr_VLV1R","","eps","","Pwr_VLV1R");
        String  VLV2R = sim.createLine(_plhs ,"EPS_Pwr_VLV2R","","eps","","Pwr_VLV2R");
        String  VLV3R = sim.createLine(_plhs ,"EPS_Pwr_VLV3R","","eps","","Pwr_VLV3R");

        sim.activateMethodCyclically("eps.step", 0.1);

        double csvPeriod = 0.01; // sec
        double csvOffs = 0.0;
        String filename = "test/Subsystems/eps/testresults/testEPS.csv";
        CSVsampler CSV = new CSVsampler(csvPeriod, csvOffs, filename, sim, sys);

        sim.init();

        // Parameters to observe
        CSV.addToCSVsampler("eps.Param.opPoint");
        CSV.addToCSVsampler("eps.Param.thrust");

        // Test commands
        CSV.closeFile();
        sim.disconnect();
        _success = true;
        return _success;
    }
```

Figure 8.1: Basic EPS test script structure

8.1 Software Level Functional Testing

The EPS model testing was separated into two parts: general model testing and mass flow control testing in [11]. A mixture of these test cases is shown in Table 8.1. Test cases one to four refer to the thrust operation of the EPS. Different configurations of the mass flow valves are turned on and off for the different mass flows. Test three uses a combination of redundant and nominal valves.

Table 8.1: EPS model test cases

Test	Description
1	Nominal thrust sequence for 3 sccm.
2	Nominal thrust sequence for 2 sccm.
3	Nominal thrust sequence for 1.5 sccm using redundant and nominal valves.
4	Nominal thrust sequence for 1.5 sccm to 2.5 sccm.
5	Neutralizer is cut off after 5 seconds.
6	Mass flow control valve is cut off early after 17 seconds.
7	No power to neutralizer from the start.
8	Neutralizer powered first then PPU.
9	Both PPU powered.

Test five and six are for testing what happens when some of the EPS components shut down unexpectedly during EPS operation. Test seven is when there is no power supplied to the neutralizer. Test case eight is for testing if the EPS starts if the neutralizer (NEUT) is powered first and then the PPU. Test nine is to check if the EPS operates if both PPU's are powered.

The sequence to fully operate the EPS, a sequence must be followed to switch it on and switch it off. To switch on, the control valve is opened, then the PPU, then the neutralizer and lastly the mass flow valves. The same arrangement is followed but in reverse for switching off. Examples of code to switch on and off the EPS is shown in A. 2.

The simulation results for test case one is shown below. In Figure 8.2, the mass flow and the thrust value are shown. In Figure 8.3, the electrical parameters are shown. Based on the commanding sequence, the control valve is opened and the anode voltage is first increased to 700 V. When the neutraliser is powered, the electron emission starts which implies there is cathode current. The mass flow valves are opened according to the set operation point. At this point the propellant enters the discharge chamber and the thruster is ignited. The thrust builds up linearly until it reaches the maximum thrust. At the same time the anode current rises which is followed by a cathode current rise at equivalent magnitude to balance the electron losses[11]. When the thruster is switched off, all values go back to zero except for the thrust value because this is to represent the real functionality of a thruster.

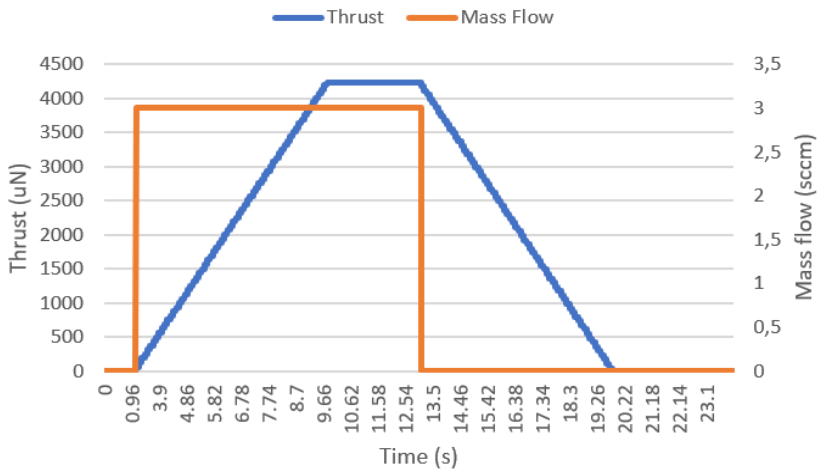


Figure 8.2: Thrust and mass flow simulation

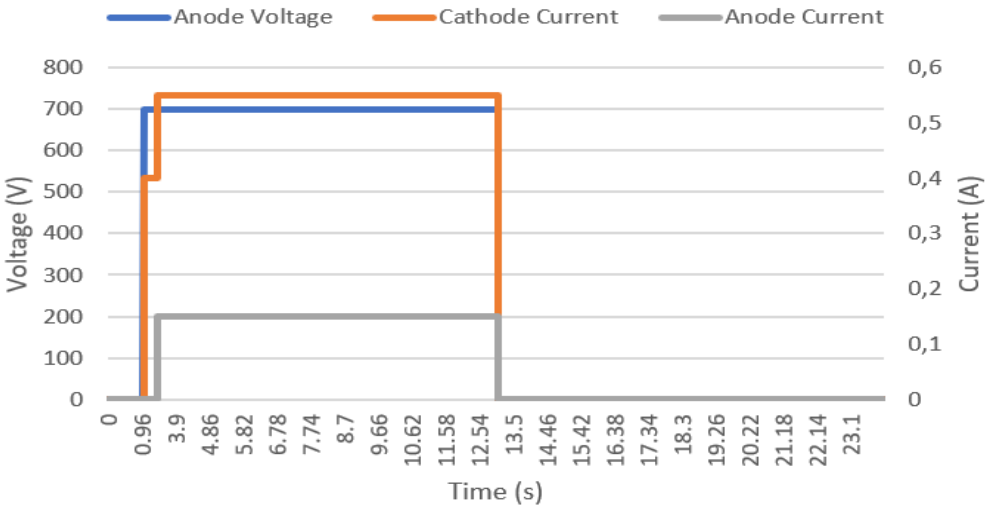


Figure 8.3: Electrical parameters simulation

The power consumption during the thrust manoeuvre is shown in Figure 8.4. The spikes at the beginning of the simulation and the 20 second mark represent the control valve opening and closing. When the neutraliser is powered, the first response is to recognize the power consumption before ignition takes place. When ignition occurs, the power consumption increases to the maximum. The power overshoot at maximum power refers to the mass flow valves being opened. The remaining test cases were done to fully verify the correct functionality of the EPS as done in [11].

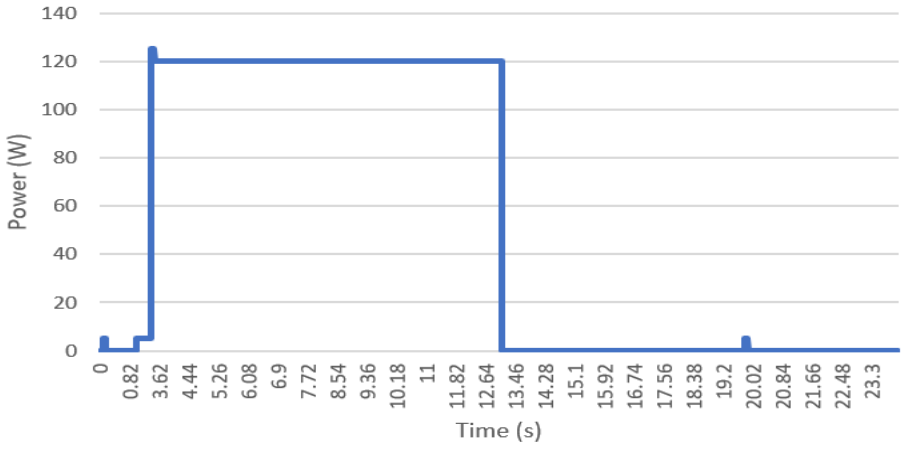


Figure 8.4: Power consumption during thrust operation

8.1.1 *PCDU and EPS test*

After successful testing of the EPS model on its own, the EPS was tested with the PCDU model. To test the EPS with the PCDU, a *PCDUPacketCreation* file is created. The file generates packets that can be sent to the PCDU to command any of the powerlines for the EPS.

The test case for the EPS can execute all commands by using the *CommandPacket()* function. The functions parameters are the command ID, parameter 1 and parameter 2. The function generates the packets required to send to the PCDU including the CRC in the format shown in Table 6.3. All the testcases in Table 8.1 are repeated in this scenario. Before the test cases can be executed, the PCDU mode is changed to miniops and then idle mode. In idle mode, PCDU commands can be sent. The simulation results of the integration of the EPS with the PCDU model was similar to the software level verification.

8.2 Mission Control System Testing

The mission control testing of the EPS is very similar to the Gecko. The definitions found for the PCDU had to be modified to include commanding for the EPS. Both the TC and TM commands were modified and added. The subservice (2,130) for the EPS commanding is similar to Figure 7.12. The only difference is the TC name is “EPS00000”, the object ID is “44003200" which is for the PCDU handler address and the command parameter is “00000040F006” which is to command the thruster ignition. The subservice (2,135) for telemetry is similar to the one in [11].

9. Flight Scenarios

Since both models have been verified, the two flight scenarios that will be completed are the capturing of the Western Cape and Nairobi and the second flight scenario will be the orbit raise manoeuvre. The orbit's altitude shall be raised from 600 km to 700 km. The determined flight altitude for the ARMC satellites is 700 km.

9.1 Flight over Cape Town and Nairobi

The selected orbit details are shown in Table 9.1 are from the NigeriaSatX orbital parameters. [43]

Table 9.1: Orbital Parameters

Orbit Parameters	Values
Nodal period	98.8 minutes
Eccentricity	0.013644
Semi-major axis	7078.25 km
Inclination	98.18 degrees
Apogee	716.9 km
Perigee	697.6 km

The test script for the flight simulation is very similar to Figure 7.1. The only difference is the addition of a *captureImage(longitude, latitude)* function. The function takes in two parameters for the desired longitude and latitude for the target. It checks if the satellite longitude and latitude is within the range of the target location to be able to send a take picture command to the Gecko Camera. Once a take picture command is sent to the Gecko, the coordinates will be stored. The stored coordinates are then used in the StaticMap code to generate images. The target locations are shown in Table 9.2. The images captured of both locations are shown in Figure 9.1 and Figure 9.2. The two images represent the image data if the real Gecko camera was to capture the two locations.

Table 9.2: Target Locations

Coordinates	Location
33.9249° S, 18.4241° E	Cape Town, South Africa
1.9441° S, 30.0619° E	Nairobi, Kenya

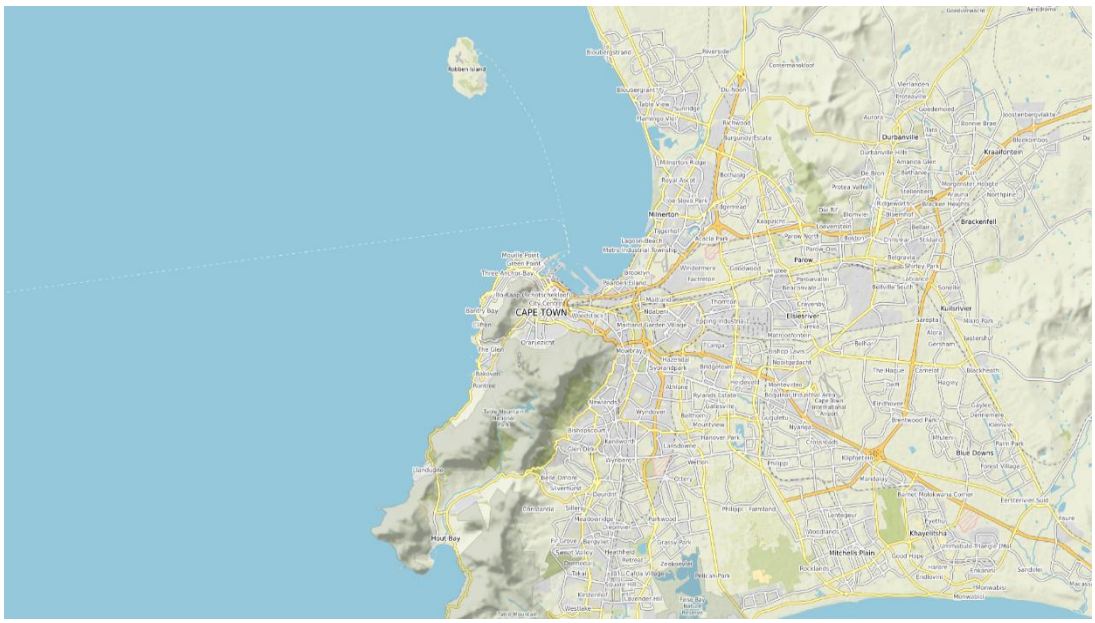


Figure 9.1: Cape Town Image

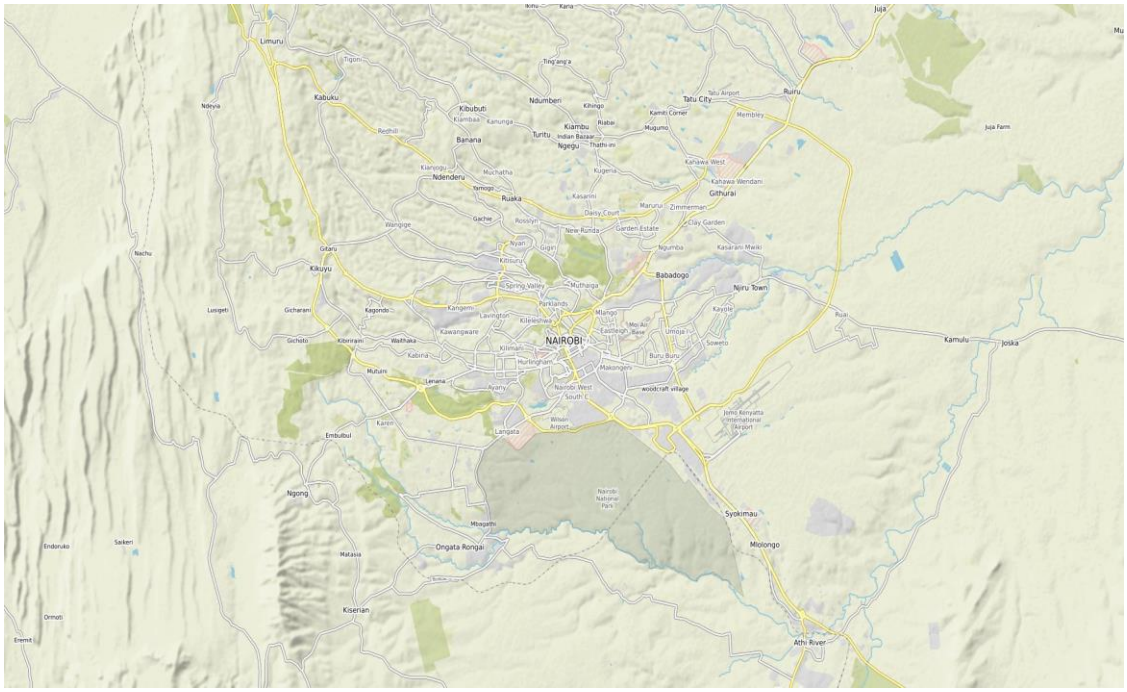


Figure 9.2: Nairobi Image

9.2 EPS Raise Manoeuvre

The EPS raise manoeuvre will raise the altitude of the satellite from 600 to 700 km. The thruster will be set to operation 7 which will produce a thrust (T) value of 4 mN with an I_{sp} of 1500 s. The transfer time with the Earth-shadow effects is calculated using the formulas below:

$$mass\ flow\ rate = T/(g * I_{sp}) \quad \textbf{9-1}$$

$$\tau = \frac{m_i}{mass\ flow\ rate}/86400 \quad \textbf{9-2}$$

$$\tilde{c} = \sqrt{\pi * \mu / R_E} * 1000 * (\tanh^{-1}\left(\sqrt{\frac{R_E}{\pi * a_i}}\right) - \tanh^{-1}\left(\sqrt{\frac{R_E}{\pi * a_f}}\right)) \quad \textbf{9-3}$$

$$Transfer\ time = \tau\left(1 - e^{\frac{-\tilde{c}}{c}}\right) \quad \textbf{9-4}$$

The derivations for equations 9-1 to 9-4 can be found in [44]. After entering the values for the EPS characteristics and the orbit characteristics, the transfer time for our EPS is 26.14 days at operation point 7.

The EPS raise manoeuvre test script is very similar to Figure 8.1. The only difference is the inclusion of the length of the operation time. The EPS raise manoeuvre test results are shown in Figure 9.3.. The first part of the figure shows the satellite orbiting around the 600km mark with a variation of 20km. The change in the figure is when the EPS thruster is operating shown by the thrust line in orange. There is no altitude change as the generated thrust cannot overcome atmospheric drag. During the period of thrust operation, the altitude of the satellite oscillates substantially, and as the thruster stays in operation, the oscillations worsen. The thruster was in operation for 10 days.

After 10 days, EPS is turned off and the orbital altitude stabilizes but the orbital variation at 600km altitude is much larger than when the thruster was turned on. The orbital altitude was supposed to change but there is no significant change when the propulsion system is operating.

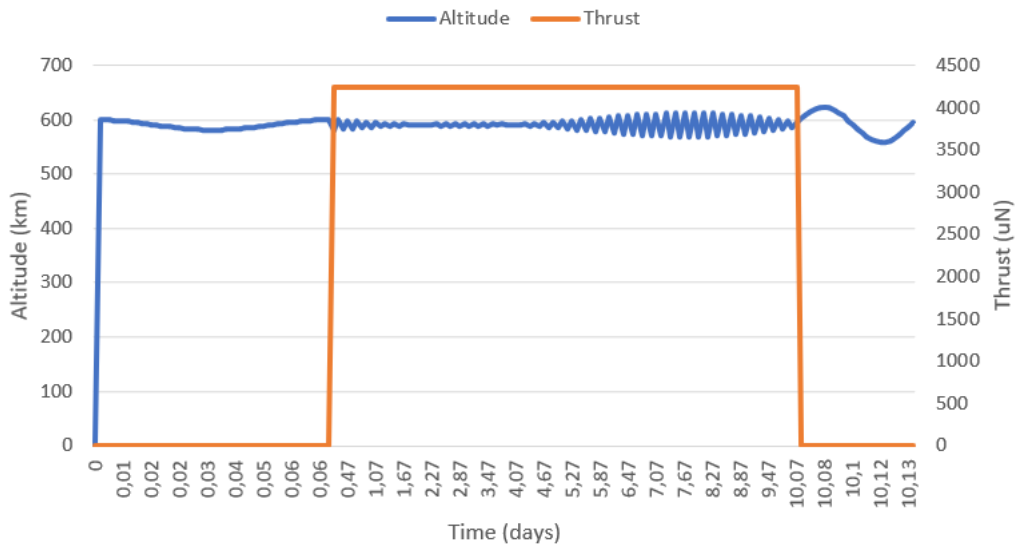


Figure 9.3: Orbit Raise Manoeuvre

10. Conclusions and Recommendations

The modelling and simulation of the Gecko Camera and the µHempt electric propulsion system was successfully implemented in the FLP Generation 2 simulator. The functionality of the models was rigorously tested in different test cases and performed correctly. The enhancement of the flight software was successful and mission control system was successful for both models except the testing of the telemetry packets for both models.

The realistic flight simulation for the Gecko Camera was successful and images for the target locations in South Africa and Kenya were correct and validated. The success of the realistic flight simulation proves the simulated Gecko was able to perform the satellite vegetation monitoring over the African continent especially South Africa and Kenya.

The realistic flight simulation of the electric propulsion system was successful but the orbit raising of the satellite was not successful. The thrust output of the electric propulsion system was not enough to overcome the atmospheric drag at 600 km altitude.

The inclusion of the both models prove the FLP Platform Generation 2 can handle different sized payloads such as those meant for nano-satellites to microsatellites and handle low-cost effective propulsion systems meant for small space missions.

Therefore, the ARMC satellite constellation can be simulated in the SimTG and be extended by the Gecko Camera and µHempt propulsion system for future satellite designs. The designs can be verified with the OBSW and each simulated model can be replaced by the real hardware to provide a fully function satellite constellation system.

10.1 Recommendations for future work

- Develop the next generation ARMC satellite system and include the simulated Gecko Imager and the electric propulsion system. This will allow the ARMC satellite constellation to have multiple payloads and increase its lifetime in orbit.
- Development of future thruster designs to improve the maximum thrust provided for similar power configurations/requirements. This will provide the thruster with enough power to overcome the J2 accelerations and any other perturbations.
- The development of a more functional PPU model for the electric propulsion system to provide better satellite orbit control during mission operations.
- Future ARMC satellite constellations should consider using micro propulsion systems and including nano cameras for Earth Observation missions.[45]
- The addition of another Gecko model to provide simultaneous image capturing and independent image capturing. This will provide more data for target locations due to increased storage capacity while also providing options for video and image data to be collected simultaneously.
- The inclusions of a Mass Memory Unit for the Gecko Camera and other payloads. The simulation of data being stored from the Gecko to the Mass Memory Unit when commanded is important and that data being transferred to the ground.
- The synchronization of satellite constellations orbits being raised during orbit operation to avoid nonalignment between satellites
- The enhancements of the PCDU to provide different power values for equipment. Currently, the power bus only provides 25V using Solar panels and 28.5V using batteries.[39]

- The development and testing of image processing algorithms for the data transferred from the payloads. This should increase the volume of data sent from the satellite to the ground station for analysis.
- The development of a hybrid propulsion system between chemical and electrical propulsion systems for accurate attitude and orbit control of constellation in orbit.
- Include a section discussing related works on the simulation using the testbench for verification of space systems.
- Include a section for test traceability for test cases for the simulation of models. The discussion and presentation of percentage of testing coverage. How much the code of the model was exercised? The functions of the models.
- For each simulation in Chapter 7, the inclusion of the simulation runs, the number of runs and simulation step used should be shown.
- In chapter 7, when test results are being analysed, before each graph an expected graph and actual graph should be compared which are related to the command sent for each simulated model.
- The inclusion of a basic work breakdown structure and Gannt chart for the plan of development in Chapter 1.

References

[1] (Aug 22,). *Four African Countries Reignite Plan To Launch Pan-African Satellite Constellation*. Available: https://africanews.space/four-african-countries-reignite-plan-to-launch-pan-african-satellite-constellation/.

[2] K. Leidig *et al*, *Multi-Mission Operations System Supporting Satellite Constellations.* 2021.

[3] J. Eickhoff, *The FLP Microsatellite Platform: Flight Operations Manual.* Springer International Publishing, 2016.

[4] S. W. Janson, "25 years of small satellites," in *25th Annual AIAA/USU Conference on Small Satellites*, 2011, pp. 1-13.

[5] J. R. Kopacz, R. Herschitz and J. Roney, "Small satellites an overview and assessment," *Acta Astronautica*, vol. 170, pp. 93-105, 2020. Available: https://doi.org/10.1016/j.actaastro.2020.01.034.

[6] *SmallSat by the Numbers, 2022*. Available: https://brycetech.com/reports/report-documents/Bryce_Smallsats_2022.pdf.

[7] *Meet the African Countries With Their Own Satellites in Orbit*. Available: https://en.sputniknews.africa/20230914/here-are-african-countries-with-their-own-satellites-in-orbit-1062105968.html.

[8] ("Feb 4, "). *South Africa has just deployed its first truly operational satellites*.

[9] *Small Satellites - SANSA*.

[10] D. CC-BY 3.0, "Flying Laptop," *German Aerospace Center,* .

[11] J. Knippschild, "Simulation Model of an Electric Propulsion Subsystem for Small Satellites.", University of Stuttgart.

[12] *ZARM: Satellite System Models*. Available: https://www.zarm.uni-bremen.de/en/research/space-science/micro-satellite-systems-and-modelling-methods/research-areas/satellite-system-models.html.

[13] J. Eickhoff, *Simulating Spacecraft Systems.* (1st ed.) Berlin Heidelberg: Springer-Verlag, 2009.

[14] J. Eickhoff, B. Chintalapati and et al, "The flexible LEO platform for small satellite missions," in *Deutscher Luft- Und Raumfahrtkongress 2018, Friedrichshafen,* "Dec 14, 2018", pp. 1-9.

[15] *DMC Constellation*. Available: http://www.dmcii.com/?page_id=9275.

[16] *Our Constellations*. Available: https://storage.googleapis.com/planet-ditl/day-in-the-life/index.html.

[17] ("Dec 12, "). *Optics or Radars? What is Better for the Earth Observation Purposes?*.

[18] Space Advisory Company, "SCS Gecko Brochure," Nov 14, 2016.

[19] P. Gurfil and P. Kenneth Seidelmann, "Satellite orbit control," in *Celestial Mechanics and Astrodynamics: Theory and Practice*Anonymous Springer-Verlag Berlin Heidelberg, 2016, pp. 369-410.

[20] C. Bombardelli, G. Baù and J. Pelaez, "Asymptotic solution for the two-body problem with constant tangential thrust acceleration," *Celestial Mechanics and Dynamical Astronomy,* vol. 110, pp. 239-256, 2011. . DOI: 10.1007/s10569-011-9353-3.

[21] J. Barrie Moss and J. P. W. Stark, "Propulsion systems," in *Spacecraft Systems Engineering*, P. W. Fortescue, G. G. Swinerd and Graham, Eds. United Kingdom: John Wiley & Sons, Ltd, 2011, pp. 177-219.

[22] S. Mazouffre, "Electric propulsion for satellites and spacecraft: established technologies and novel approaches," *Electric Propulsion for Satellites and Spacecraft: Established Technologies and Novel Approaches,* vol. 25, *(3),* pp. 033002, 2016.

[23] G. Kornfeld, N. Koch and H. Harmann, "Physics and evolution of HEMP-thrusters," in *The 30th International Electric Propulsion Conference,* Florence, Italy, pp. 1-18.

[24] P. Kindberg, "Development of a Miniature Gridded Ion Thruster." , Lulea University of Technology, 2017.

[25] K. Matyash *et al*, "Kinetic simulations of SPT and HEMP thrusters including the near-field plume region," *IEEE Trans. Plasma Sci.,* vol. 38, *(9),* pp. 2274-2280, 2010.

[26] K. Matyash *et al*, *Kinetic Simulation of the Stationary HEMP Thruster Including the Near–field Plume Region.* 2009.

[27] W. P. Wright and P. Ferrer, "Electric micropropulsion systems," *Prog. Aerospace Sci.,* vol. 74, pp. 48-61, 2015. . DOI: 10.1016/j.paerosci.2014.10.003.

[28] D. Lev *et al*, *The Technological and Commercial Expansion of Electric Propulsion in the Past 24 Years.* 2017.

[29] D. Krejci and P. Lozano, "Space Propulsion Technology for Small Spacecraft." vol. 106, *(3),* pp. 362–378, Mar 1, 2018.

[30] (September 20,). *The future small satellites open up with development of propulsion systems*.

[31] *Starlink*. Available: https://www.starlink.com/technology.

[32] (Feb 6,). *Launches of Busek Thrusters Push OneWeb Constellation Towards Completion*. Available: https://www.prnewswire.com/news-releases/launches-of-busek-thrusters-push-oneweb-constellation-towards-completion-301739030.html.

[33] O. Zanon and O. Meaurant, "SimTG Model Development User Manual," *(1.0),* "Jan 23, ", 2012.

[34] V. Hertel, "Experiment Design for a Dream Chaser Orbital Mission Flight Opportunity." , University Of Stuttgart, 2019.

[35] Space Advisory Company, "Gecko Imager Datasheet," .

[36] A. Isaac BARQUIN MURGUIA, "SmallSat Payload Simulation for Onboard-Software VeriCation." , Universite Toulouse III and Lulea University of Technology, 2016.

[37] *StaticMap*. Available: https://github.com/komoot/staticmap.

[38] Micron, "Technical Note: Small-Block vs. Large-Block NAND Flash Devices," 2005.

[39] K. Leidig, "PCDU Model Documentation," *(1.2),* pp. 1-38, "Feb 12, ", 2018.

[40] K. Leidig, "IO-Board Model Documentation," *(1.4),* pp. 1-34, "Oct 11, ", 2018.

[41] F. Team, "FLP2 Mission Information Base," *(1.0),* pp. 1-42, "Jun 25, ", 2020.

[42] ECSS, "ECSS-E-70-41A,Ground Systems and Operations - Telemetry and Telecommand Packet Utilization," Jan 1, 2003.

[43] ("Jul 1, "). *ST/SG/SER.E/719 - Information Furnished in Conformity with the Convention on Registration of Objects Launched into Outer Space*. Available: https://www.unoosa.org/oosa/osoindex/data/documents/ng/st/stsgser.e719.html.

[44] C. A. Kluever, "Low-thrust transfers," in *Space Flight Dynamics*Anonymous John Wiley & Sons Ltd, 2018, pp. 303-329.

[45] S. Mostert and M. Jacobs, "ARM constellation - Establishing a regional remote sensing asset," *Acta Astronaut.,* vol. 63, *(1-4),* pp. 221-227, 2008. . DOI: 10.1016/j.actaastro.2007.12.030.

Appendix

A. 1: CRC Calculation

```
unsigned short CalcCRC2 ( unsigned char * db , int count ) {
    int i;
    unsigned short crc ;
    crc = 0 xFFFF ;
    while (-- count >= 0) {
        crc = crc ^ (( unsigned short ) *db ++ << 8);
        for (i =0; i <8; i ++) {
            if ( crc & 0 x8000 ) crc = ( crc << 1) ^ 0 x1021 ;
            else crc = crc << 1;
        }
    }
    return ( crc );
}
```

A. 2: Sequence for switching on and off the EPS

```
// Command for operation point 6 i.e. sccm = 3

// Command EPS On

//Control Valve
sim.powerLineSetVoltage(VLV0N,25);
try {
    callCSV(1, CSV);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

//PPU
sim.powerLineSetVoltage(PPU0,25);
sim.timeStep(1);
CSV.writeData();

// Neutralizer
sim.powerLineSetVoltage(NEUT0,25);
sim.timeStep(1);
CSV.writeData();

//Mass Flow Valves
sim.powerLineSetVoltage(VLV1N,25);
sim.powerLineSetVoltage(VLV2N,25);
sim.powerLineSetVoltage(VLV3N,25);

try {
    callCSV(3, CSV);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

```
// Command EPS Off

//Mass Flow Valves
sim.powerLineSetVoltage(VLV1N,0);
sim.powerLineSetVoltage(VLV2N,0);
sim.powerLineSetVoltage(VLV3N,0);
sim.timeStep(1);
CSV.writeData();

//Neutralizer
sim.powerLineSetVoltage(NEUT0,0);
sim.timeStep(1);
CSV.writeData();

//PPU
sim.powerLineSetVoltage(PPU0,0);
sim.timeStep(1);
CSV.writeData();

//Control Valve
sim.powerLineSetVoltage(VLV0N,0);
sim.timeStep(1);
CSV.writeData();
```