Automated functional system integration testing of Suomi 100 satellite

Juha-Matti Lukkari

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo May 18, 2018

Thesis supervisor:

Prof. Esa Kallio

Thesis advisor:

D.Sc. (Tech.) Antti Kestilä

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Juha-Matti Lukkari

Title: Automated functional system integration testing of Suomi 100 satellite

Date: May 18, 2018 Language: English Number of pages: 7+118

Department of Electronics and Nanoengineering

Professorship: Space Science and Technology

Supervisor: Prof. Esa Kallio

Advisor: D.Sc. (Tech.) Antti Kestilä

A large portion of launched CubeSats have failed early on their missions. Potential source of failures has been identified from statistical data of CubeSat missions as being inadequate functional system integration testing. In this thesis test automation was used to perform functional system integration testing for the Suomi100 CubeSat. Reusable software library, called *CubeSatAutomation*, was developed for test automation and testing was conducted with a widely used open source test automation framework known as Robot Framework. With the performed tests proper functionality was verified for essential satellite features such as radio communication, telemetry, safe resets and battery recharging through the solar panels among others. The testing however identified certain issues in the integration of the payload radio instrument. The tests included the "Day in the life" testing and it is possible to anticipate that this test can increase the overall success rate of CubeSat missions. A testing guideline that includes this test is recommended to be added to the CubeSat project.

Keywords: CubeSat, Satellite failures, Suomi100, System integration testing, Test automation, Day in the life of a satellite

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Juha-Matti Lukkari

Työn nimi: Automatisoitu Suomi 100 satelliitin funktionaalinen järjestelmän integraatiotestaus

Päivämäärä: May 18, 2018 Kieli: Englanti Sivumäärä: 7+118

Elektroniikan ja nanotekniikan laitos

Professuuri: Avaruustiede ja -tekniikka

Työn valvoja: Prof. Esa Kallio

Työn ohjaaja: TkT Antti Kestilä

Suuri osa avaruuteen laukaistuista CubeSat –satelliiteista on epäonnistunut jo mission alkaessa. Toteutuneiden CubeSat –missioiden tilastojen pohjalta onkin esitetty, että yksi merkittävä epäonnistumisen syy on ollut puutteellinen järjestelmän funktionaalinen integraatiotestaus. Tässä työssä Suomi 100 –piensatelliitille suoritettiin testiautomaatiota käyttäen funktionaalisia järjestelmän integraatiotestejä. Työssä kehitettiin uudelleenkäytettävä testiautomaatiokirjasto, nimeltään *CubeSatAutomation*, ja tätä voidaan käyttää vapaasti jaettavissa olevan Robot Framework testaustyökalun kanssa. Testien avulla varmennettiin satelliitin perusominaisuuksien toimivuus, kuten radiokommunikaatio, telemetria, turvalliset uudelleenkäynnistykset, akkujen latautuminen aurinkopaneelien avulla, ynnä muita. Ongelmia satelliitin hyötykuormaradioinstrumentin integraation kanssa ilmeni testien kautta. Satelliitille tehtiin myös niin kutsuttuja "Päivä satelliitin elämässä”-testejä. On oletettavaa, että tämän testin suorittaminen piensatelliiteille pienentäisi aikaisen epäonnistumisen todennäköisyyttä. Työn tuloksena on suositus CubeSat –satelliitti konseptiin lisättäväksi ohjeistusta järjestelmän integraatiotestaamisesta, johon sisältyy edellämainittu testi.

Avainsanat: CubeSat, Satelliittien epäonnistumiset, Suomi100, Järjestelmän integraatiotestaus, Testiautomaatio, Päivä satelliitin elämässä

Preface

I wish to thank Professor Esa Kallio and my instructor Antti Kestilä for their expert guidance. Further, I wish to thank Petri Koskimaa, Jouni Rynö, Arno Alho and Amin Modabberian from the Suomi100 satellite team for all their highly valuable discussions and help.

Otaniemi, May 18, 2018

Juha-Matti Lukkari

Contents

Abbreviations

1U	One-unit CubeSat
ADC	Analog to Digital Conversion
ADCS	Attitude Determination and Control System
AM	Amplitude Modulated
API	Application Programming Interface
BMP	Bitmap image file
Cal Poly	California Polytechnic State University
CAN	Controller Area Network
CFD	CubeSat Failure Database
CI/CD	Continuous Integration/Continuous Development
CMOS	Complementary Metal Oxide Semiconductor
COM	Communication System
CONOPS	Operation Concept Document
COTS	Commercial-off-the-self
CSP	CubeSat Space Protocol
DOA	Dead On Arrival
EPS	Electric Power System
ESA	European Space Agency
FFT	Fast Fourier Transform
FM	Frequency Modulated
FTP	File Transfer Protocol
GOSH	GomSpace Shell
GPIO	General Purpose Input-Output
GPS	Global Positioning System
HF	High frequency
HK	Housekeeping
HTML	Hypertext Markup Language
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
JPEG	Joint Photographic Experts Group
LNA	Low-noise Amplifier
MCU	Microcontroller Unit
MF	Medium frequency
NASA	National Aeronautics and Space Administration
OBC	On-Board Computer
PA	Power Amplifier
PCB	Printed Circuit Board
P-POD	Poly Picosatellite Orbital Deployer
RAW	Raw image file
RF	Radiofrequency
RISC	Reduced Instruction Set Computer
RTOS	Real-time Operating System
SDRAM	Synchronous Dynamic Random Access Memory
SDR	Software Defined Radio
SSH	Secure Shell
Stdin	Standard input stream
Stdout	Standard output stream
SUT	System Under Test
TLYF	Test Like You Fly
UHF	Ulta-High Frequency

1 Introduction

1.1 Increasing interest in space

Since the 1950s, mankind has made great steps into space [1]. Nations, industries, businesses, militaries, universities and even private entrepreneurs have sought to benefit from the opportunities that space offers [1, 2, 3]. Great advancements have been made in technology and science to propel this endeavour further forward [1]. Examples of such leaps in techology include sending the first human into space in 1957, the Moon landings in 1969, the sending of probes to other planets in the Solar System such as Mars and most recently, getting the first images of Pluto in a flyby mission in 2013 [1, 4]. The use of space technology has also entered into household items through, for example, the use of the *Global Positioning System* (GPS) satellite network in mobile phones, cars, and so forth [5]. People and industries have began increasingly to be reliant on spaceborne technologies and devices [5].

From the end of the 1990s, new inventions have additionally brought the design and manufacturing of these technologies relatively closer to everyday people, away from the assembly sites of large nations and large organizations into laboratories run, for instance, by university students [6]. Advancements leading to this can be attributed to the space industry catching up with the advancements of electronics as well as to cheap launch opportunities that have become available [7, 8]. More concretely, development of the *CubeSat* nanosatellite concept in 1999 in California Polytechnic State University (Cal Poly) has in recent years brought about hundreds of new satellite developers and hundreds of new space missions based on this nanosatellite concept [6, 7, 9].

These satellites typically are relatively small and usually use commercial off-the-self (COTS) components, yet are still capable of operating in space around Earth [6, 7]. CubeSats have in recent years emerged as a new viable platform for carrying out space missions [7]. Moreover, due to their small size their launch costs are smaller [8]. The use of COTS components makes these satellites relatively quick to design and cheap to manufacture. Several companies have shown great interest in the concept and other organizations (such as the US military) have shown interest as well [6, 7, 8].

As an example, the Finnish government recently passed a new law regarding space and is pursuing the creation of a new industry of space technology related companies in Finland [10]. The first CubeSats built in Finland at *Aalto University* have even created new space companies which are building their own satellites to be launched into space, for instance *Iceye* [11]. In all, over 600 CubeSat missions have been launched since the first mission in 2003 [12]. The trend seems to be so that more and more CubeSat missions are to come, and they are starting to take a clear share of the space industry market [13]. Already in 2014, approximately half of the flown space missions in that year were CubeSat missions [14].

1.2 Substantial proportion of failed CubeSat missions

Even though the CubeSat concept has rapidly created a large amount of new space missions, a large portion of the missions launched have ended in failure due to various reasons [14, 15]. Several surveys done in recent years have found a failure rate of 40 % for CubeSat missions made by newcomer developement teams [14, 15, 16, 17]. However, if the satellites that were manufactured by teams with earlier experience in space missions, failure rates have demonstrated considerably lower. Suggestions have been made in these surveys that the missions that have failed have not performed proper functional testing on ground at a system integration level or that such testing has been missing completely [14, 15, 16].

Ph.D. Michael Swartwout has made several studies into this subject [12]. His research suggests that the majority of CubeSat failures could be attributed to inadequate testing of the satellite in a flight-equivalent state on the ground [14, 16, 17]. He believes that functional testing of the whole integrated satellite system has been lacking completely or has been done in a very limited sense. Failures that have been attributed to improper functional system integration include the solar panels not being properly connected, insufficient power generated for the transmitter, unrecoverable processor errors and so forth [16].

Though the concept of CubeSats shows promise, from the statistical data it can be seen that there remain some challenges for the satellite concept [12, 14, 15, 16, 17]. Overall reliability of CubeSats needs to be improved, if they are to become a valid alternative to traditional space missions that are long lasting and resource consuming.

1.3 Suomi100 CubeSat

The satellite involved in the research is called *Suomi100* [18] and it is a *one-unit* (1U) CubeSat. The project was conceived in the interest of celebrating Finland's 100 years of independence. The mission of the satellite is to take images of Finland from space and to measure different radio signals present in the Ionosphere. An artistic impression can be seen in Figure 1 on the next page.

1.4 Research purpose and goals

The aim of this thesis is to investigate *how to carry out functional system integration testing in order to improve CubeSat reliability.* Further, we wish to perform the testing in a systematic manner. By using software tools for automation of the testing, we can achieve a certaing degree of rigour and a systematic approach to testing. Similarly, verification tests for mechanical stress required by the CubeSat standard are done systematically in an automated fashion [9]. It would also be preferable that the functional system tests could be performed automatically in a systematic manner.

From the technology point of view, the goal will be achieved by developing new generic and reusable function library with *Python* programming language which can be used with the *Robot Framework* [19] along with appropriate test suites and test setups.

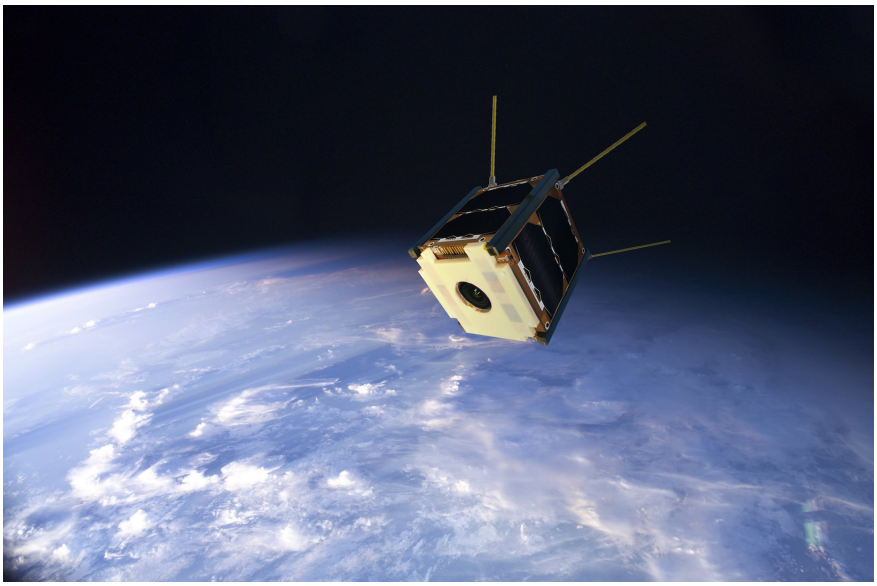

Figure 1: Depiction of Suomi100 satellite in space. Courtesy of Mr. Jari Mäkinen.

1.5 Main questions and problems

The main problem analyzed in this thesis is the unreliability of CubeSats, which the thesis tries to partly solve from the standpoint of system integration testing. The main question is, firstly, could this type of testing detect unrecoverable failures in the satellite operation and system integration, and, secondly, could we, in addition verify that the satellite fulfills its functional requirements?

1.6 Outlining the scope of research

In this thesis we study the use of one industry-proven automated acceptance framework to carry out the automated functional testing on Suomi100 and investigate the use of certain space-industry proven testing philosophies and methodologies into functional system integration testing with CubeSats.

In conjunction with the space industry test methodologies, we attempt to some degree simulate the environment related to functional operations of the satellite. In addition, we require that the simulation has to take into account the relatively small funds of university-led CubeSat projects.

2 Background

2.1 CubeSat failures

The CubeSat project was started in 1999 in Cal Poly [9]. Over 100 universities and other organisations have since contributed to the project. The purpose of the project is to provide a standard for the design of nanosatellites in order to decrease development costs and to make accessability to space easier [9]. In fact, the number of launched CubeSats has increased quite dramatically over the past few years [14], and it has been estimated that the number of CubeSat missions will increase in the coming years [13]. Nevertheless, a large portion of these missions have failed due to various reasons [14, 16, 17]. The most common reasons include communication being lost with the satellite, batteries not recharging and the OBC not restarting, with an average of only 20 % of missions being able to complete their full missions [14].

2.1.1 The CubeSat satellite specification

The CubeSat project defines a CubeSat to be a nanosatellite with dimensions of 10 cm × 10 cm × 10 cm and with a mass up to 1.33 kg [9]. A satellite of this size is considered to be a 1U CubeSat. These units can be stacked together to form larger CubeSats, with some satellites consisting of even 12 units. A stack containing three units appears statistically to be the preferred size for a CubeSat [14].

Another important part of the concept is the *Poly Picosatellite Orbital Deployer* (P-POD), which is a Cal Poly's standardized CubeSat deployment system [9]. This deployment system is integrated with the launch vehicle and the springs in the system release the nanosatellites into space [9]. Usually a launch vehicle carries some primary payload, which is much larger than the CubeSats [20]. If an extra weight can be launched along with the main payload, then the deployment pods with the CubeSats contained inside are integrated into the launch vehicle as secondary payloads [20].

Although these nanosatellites are considerably smaller than most "traditional" satellites, they nonetheless are able to perform the regular operations of a satellite [21]. As the classification, *nanosatellite* defines, CubeSats are satellites in miniature size. Even though smaller, the same general class of subsystems are part of CubeSats as they are part of larger satellites [21]. Subsystems containing electronics usually follow *PC/104* standard which defines form factor and computer bus [22]. A single subsystem in a CubeSat can fit into one *Printed Circuit Board* (PCB) following the PC/104 standard [22]. Figure 2 illustrates the internal structure of Suomi100 CubeSat and the different subsystems that are integrated into the satellite.

From this Figure we can identify five subsystems that are common to all satellites. Power to the satellite is produced by the *Electric Power System* (EPS) and this subsystem consists of the solar panels and batteries in the satellite. Additionally, the subsystem usually has some power regulation and power distribution features along with some features for reliable power production and storage [23].

The central computer of the satellite is called the *On-Board Computer* (OBC) and the task of this subsystem is to orchestrate the operation of all the other subsystems.

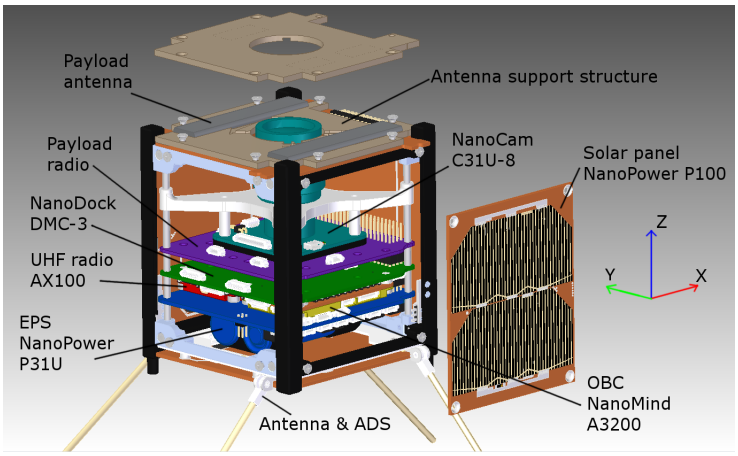

Figure 2: Depiction of Suomi100 satellite subsystems. Courtesy of Aalto University.

In addition, the processing of commands received from the *ground station* and the routing of them to the appropriate subsystem is the task of the OBC. [23]

The *Communication system* (COM) is responsible for communication with the ground station. This subsystem usually has a computer of its own for processing the received radio signals. The antennas are also part of this system. Besides receiving and processing commands from the ground station, *Housekeeping* (HK) telemetry of the satellite system is commonly broadcasted in order to inform the ground station about the state of the system. A satellite *Beacon* is a specific type of broadcasted telemetry which is used to obtain the location of the satellite as well as its state in general. [23]

The proper orientation of the satellite is controlled by the *Attitude Determination and Control System* (ADCS). For CubeSats, the orientation can be controlled either by mechanical reaction wheels, magnetorquers or by some other methods. The purpose of this system is to keep track of the orientation of the satellite and to change it according to commands received from the ground station. [23]

In addition to these subsystems, the *Support Structure* forms the subsystem which integrates all of the subsystems into a single mechanical structure [23]. The CubeSat standard defines the dimensions and materials for the support structure [9].

The ground station in itself forms another element in the entire satellite system [23]. As noted, communication with the satellite occurs via the ground station. The station tracks and monitors the satellite and all the necessary information from the satellite is downloaded to the station. A ground station consists of hardware elements such as antennas, power amplifiers and other radio components [24]. A computer is connected to the hardware elements and a ground station software on the computer is used to control and track the satellite [23, 24]. Pictures of the ground station in Aalto University are presented in Figure 3.

2.1.2 Failure rates of CubeSats

Over 600 CubeSats have been launched as of 2017 [12]. Some studies in recent years have been carried out to investigate the statistics of flown CubeSat missions.

Figure 3: Satellite ground station in Aalto University. On the left in the image a computer is running the satellite control and tracking software. Certain radio hardware elements are present as well. The antenna used for controlling Aalto-1 satellite is on the right in the picture.

These studies looked for the percentage of failed missions and which subsystems contributed to each failure. Out of these failures, the amount of *dead on arrival* (DOA) missions, where the satellite was never even able to be contacted from space, were also identified. The most active contributor to this topic has been Michael Swartwout and the representation of statistics of CubeSat failures in this thesis is based mainly on his work [14, 16, 17], as not too many papers have yet been published on this issue.

A study entitled *The First One hundred CubeSats: A Statistical Look*, that was published in 2013, identified the failure rate for the first 100 flown CubeSats. Out of these first CubeSat missions flown between the years 2000-2012, a total of 34 had failed. From these failures, a third were never able to be contacted after they were released into space (DOA cases). Since then, many CubeSats have flown with varying degrees of success [14]. Figure 4 shows the mission statistics for all CubeSats flown until 2017 [25]. The different colours show at which state a CubeSat mission is or to which state the mission ultimately reached.

In order to break down and investigate the statistics, Swartwout has been continuing yearly to publish papers about the statistics of CubeSat failures, as new missions are flown each year [14, 17]. A study published in 2016, *Secondary Spacecraft in 2016: Why Some Succeed (And Too Many Do Not)*, identified out of all CubeSats flown between 2000-2015 that reached orbit, 21 % were DOA cases, and 9.8 % were cases where the spacecraft was lost early in its life. When a CubeSat was lost early in its life, this means that communication with the satellite was established but no primary operations could be executed. When breaking down the statistics to categories based on the type of satellite and mission developer (new university teams, traditional contractors, experienced university and government teams, constellations), it was found that for the new university teams flying their first satellite, the failure rates were as follows: 44.1 % DOA, 16.2 % early loss and 16.2 % mission success. On the other hand, for the CubeSats built by traditional contractors with established

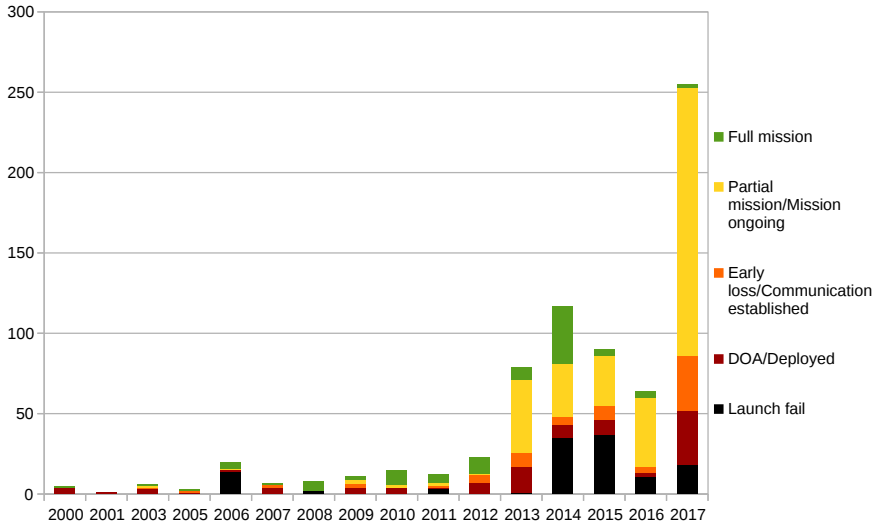

Figure 4: Mission statistics for CubeSats flown until 2017. [25]

practices for integration and testing the numbers were: 6.3 % DOA and 6.3 % early loss. This DOA failure rate, however, halved when the new university teams, educated by their first failure, flew their second satellite. [14, 17]

Figure 5 below shows the statistics of failures for CubeSats between 2000-2015 flown by new university teams sending their first satellite, excluding those missions where the satellite was lost due to launch failure.

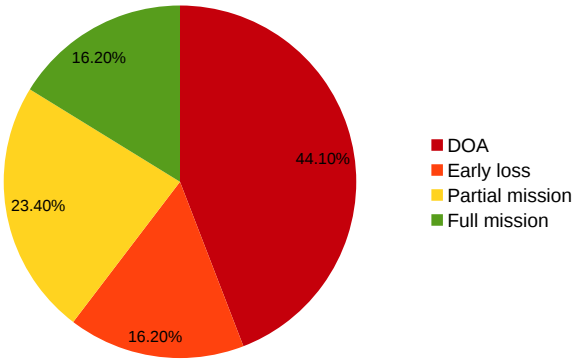

Figure 5: Statistics for CubeSats flown between 2000-2015 that were constructed by university teams without prior experience of satellite construction. [14]

For contrast, same statistics for CubeSats built by traditional contractors is shown in Figure 6. A clear difference in DOA and early loss missions is visible for these two different groups.

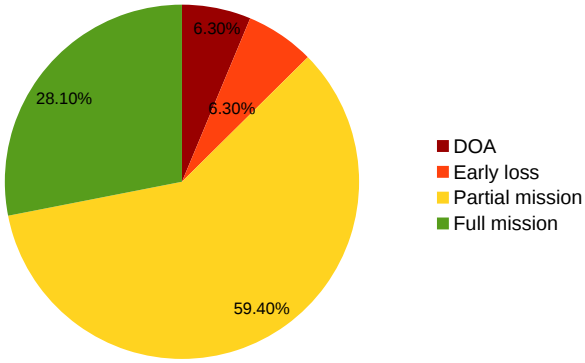

Figure 6: Statistics for CubeSats flown between 2000-2015 that were constructed by traditional contractors with extensive experience of satellite construction. [14]

2.1.3 Contribution of different subsystems to satellite failures

Swartwout studied also the contribution of different subsystems to CubeSat failures in the paper *The First One hundred CubeSats: A Statistical Look* [16]. The subsystems that were thought to be the cause of the failure were identified as follows: a configuration or interface failure between communications hardware (27%), the power subsystem (14%) and the flight processor (6%), or the COM, EPS and OBC subsystem. A failure in a subsystem means in this context that the whole satellite is lost due to the failure. A failure in the OBC can mean, for example, that the processor fails to restart anymore or gets unrecoverably stuck in some way. For EPS the error can mean, for instance, that power is not being transferred to the satellite from the solar panels, and failure in the COM subsystem can imply, for example, that there is insufficient power for the antennas to close the link with the ground station. [16]

Based on the beliefs of the satellite developers about the causes of failures, another study was carried out by Langer et al. in 2014 [15] to investigate in more detail the contribution of different subsystems in CubeSat failures. This study by Langer also used the statistical data of CubeSat failures obtained from the *CubeSat Failure Database* (CFD), at that point comprising data of about 178 CubeSat missions. With this data a reliability estimate for different subsystems was calculated using a *Kaplan-Meier estimator* for nonparametric and parametric analysis. In addition, a parametric model for total CubeSat satellite reliability was devised. Figure 7 depicts the subsystem contributions to satellite failures for the first 178 CubeSat missions. Three main subsystems causing failures were identified in order of importance: EPS, OBC and COM, in accordance with Swartwout research, but with different percentages as EPS being the main contributor to failures [15, 16].

The statistical data gathered from questionnaires sent to 987 satellite developers (with 113 returned fully completed) showed that there was a belief that within the first six months there was a 50 % chance that the satellite would fail. However, the beliefs of the developers seem to be too optimistic when compared to the data

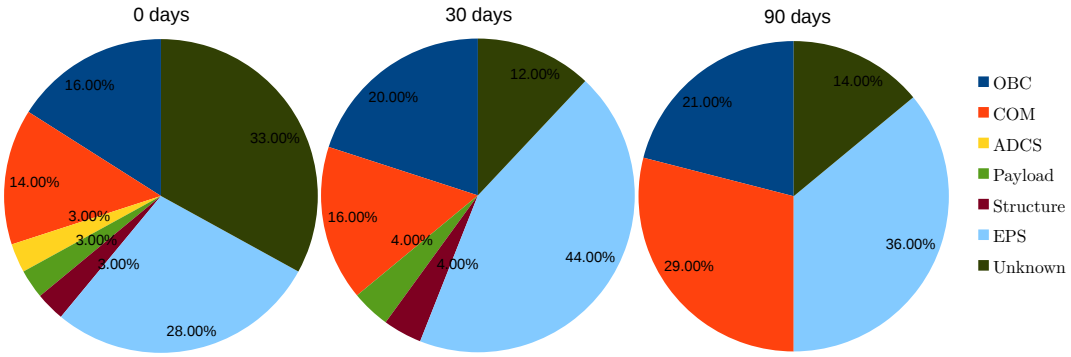

Figure 7: The beliefs of developers on the contribution of different subsystems to satellite failure. From left to right the charts present failure contribution data for 0, 30 and 90 days after launch. [15]

gathered from CFD. Nonetheless, from the subsystems having the greatest and least likelihood of causing system failure, the main subsystems were identified in the order: COM, EPS and OBC. [15]

2.1.4 Needs for system integration level functional testing

The aforementioned studies made some anecdotal guesses to what could have contributed to the failures in the satellites, ensuring that the missions failed either partially or completely. Though the current data does not clearly prove these educated guesses, it is believed by Swartwout and others that system integration level functional testing of the satellites has been lacking completely or has been inadequate [14, 15, 16, 17].

Based on his study in 2013, Swartwout came to a strong belief that the critical failures in the subsystems were caused by poor system integration. Notably, out of first 30 identified DOA cases, 24 were CubeSats made by university teams. In addition, based on his discussions with project managers and faculty leaders, it was noted that university teams constructing CubeSats have the misconception that the satellite works as expected the first time it is assemled together and thus no system integration level functional testing is performed. In the study, it was believed that operational tests demonstrating a *"Day in the life of the satellite"* would be just as necessary as the vibrational tests to certify a CubeSat ready for launch. In addition, testing of recovery from resets, power management, startup sequences etc. would be important operations for the satellite to test. [16]

In later papers Swartwout has been less reluctant to make these claims directly, yet still identifying the large number of failed CubeSat missions coming from university-led satellite teams [14, 17]. As an example, the ORS-3 mission flown in 2013 consisted of 28 secondary payloads, and 13 of these payloads were assembled by new university teams flying their first satellite, and 15 were constructed by traditional contractors [14]. While almost all (11 out of 13) of the university-built CubeSats failed, only one CubeSat built by a traditional contractor failed. Furthermore, all of these satellites had to go through the same vibrational and thermal tests and, in addition,

were subject to mission-readiness reviews by the *National Aeronautics and Space Administration* (NASA) and/or *Department of Defense* of United States. Thus, some practices applied by the experienced contractors in satellite development were most probably missing from the university-built CubeSats.

In addition, David Voss of the *Air Force Research Lab* speaking more recently at the 31st Annual Conference on Small Satellites held on 7th of August 2017 about CubeSat reliability said that, based on his experience with student and other small satellite projects, a core set of tests for power, communication and other subsystems would be needed [26]. Furthermore, Michael Johnson, also a participator in the aforementioned conference and chief technologist at NASA's *Goddard Space Flight Center* has been since 2017 working at NASA on making a reliability iniative to determine the best ways to improve CubeSat reliability [26, 27]. He noted though that the goal is not to apply the same rigorous assurance procedures used earlier in larger and more expensive spacecraft, but to design new procedures and keep some of the older methods that can be useful.

2.1.5 Comparison of CubeSat failures to failures with larger spacecraft

Besides CubeSats, failures have happened to more traditional spacecraft as well. In fact, the history of the space industry as a whole is filled with examples of failed missions [28]. As an example, in recent years several Mars landers have failed during the landing phases of the mission [29]. For instance, Mars lander *Schiaparelli* crashed on the Martian surface in 2016 when a sensor used to measure the distance to the ground read a negative value and shut off its descent thrusters [30].

Earlier research about the on-orbit failures was carried out in 2005 [31]. It investigated failures of 129 different spacecraft between the years 1980 and 2005. The study found that there were many cases where the spacecraft failed early during its mission. Majority of the failures were caused by failures in the ADCS and EPS subsystems. The investigation concluded that among improved redundancy and flexibility in system design, adequate testing on ground could as well mitigate these failures as it was noted that the early failures could have been caused by inadequate testing and inadequate modeling of the environment where the spacecraft operates in. These conclusions in fact seem to be similar to what some of the surveys done on CubeSat failures indicate. [31]

A research conducted in 2008 analyzed the contributions of different subsystems in the failures of 1584 satellites that were launched between 1990 and 2008 [32]. Solar array deployment and failure in the communication system were the major contributors to satellite failures for satellites that failed before 30 days after launch. After much longer operation period (i.e. years), the main subsystems contributing to failures were identified to be the ADCS and COM. Some similarity to the subsystem failures with CubeSats can be drawn here, with the communication subsystem and the solar panels playing a crucial role in the infant mortality of CubeSats.

Further indication for the need of extensive testing was found after NASA initiated in the 1990s a more streamlined verification strategy based on the best commercial practices, commonly known as *"Faster, Better, Cheaper"* [33]. This led to poor results

with commercial satellites that were launched during that period and a return to the more rigorous specifications and standards was expressed [33]. In addition, a study conducted during this period in 1999 called *"When Standards and Best Practices Are Ignored"*, found that out of 50 major space system failures 32 % were related to inadequate verification and test processes [34].

In conclusion, based on the experiences found by the traditional space industry, the allegation that many CubeSat failures are due to poor testing at system integration level could be correct.

2.2 Satellite testing

Satellites and many other spacecraft can be classified as *embedded systems* [35]. An embedded system is defined in *Real-Time Systems Design and Analysis: Tools for practitioner* as *"A system which contains one or more computers (or processors) having a central role in the functionality of the system, but the system is not explicitly called a computer"* [36]. Systems such as these can contain many parts and consist not just of software but of hardware elements as well [37].

The testing of embedded systems has thus to take into account software testing as well as hardware testing and the interplay of software and hardware [37]. For example, mechanical stress and thermal vacuum tests are required by the CubeSat standard to be performed for CubeSats before launch [9]. Presenting different methodologies into hardware testing is out of the scope of this thesis as the topic of research is functional testing at system integration level. Performing a test such as this is, in practice, testing the software in the final hardware environment [37]. The general methodologies and practices used in software testing as well as some testing practices used by the traditional space industry are presented in this section.

2.2.1 Practices for software testing

First a relevant question, *why do we need to perform testing?* One reason is that humans make errors and often make optimistical assumptions about their work. Another aspect would be to call testing as a method of proving that the *System Under Test* (SUT) works as we want it to work. Just as a scientist carries out experiments to prove his theory, so too testing is done to prove that the system works as expected. A system in the context of testing can refer to anything from a single software function to entire operating system of a spacecraft. [38]

Another important aspect of testing is to find defects in the system and to recognize where they exist so that they can be corrected effectively [36, 39]. A book by Glenford J. Myers titled *The Art of Software Testing*, defines software testing as *"Testing is the process of executing a program with the intent of finding errors"*, which is a general enough definition to contain many aspects of software testing [39].

Methods of testing

Various different methods for software testing exist. So called box approach is one common method for testing [36, 39]. Testing can either be done automatically

by some computer-run script or manually by a tester who follows a specified test plan. Some of the testing methods are explained in the following pages in more detail.

Black box testing
Black box testing is performed with no knowledge about the internal structure of the SUT, which can be a single function of a software or a whole operating system for example. A select set of inputs are given to the system and from the outputs we see how the system performed. If the outputs were what we expected, the system passed the test. Black box testing is usually used when we are interested in the outward functionality of the SUT. Figure 8 below illustrates black box testing method. [39]

White box testing
White box testing is performed when we are interested about the internal functionality of the SUT. Testing of the internal functions rather than the outwardly expected functionality is the goal of white box testing. Usually this type of testing is performed at the smaller component or unit level of the system. Figure 8 below gives an illustration of white box testing. [39]

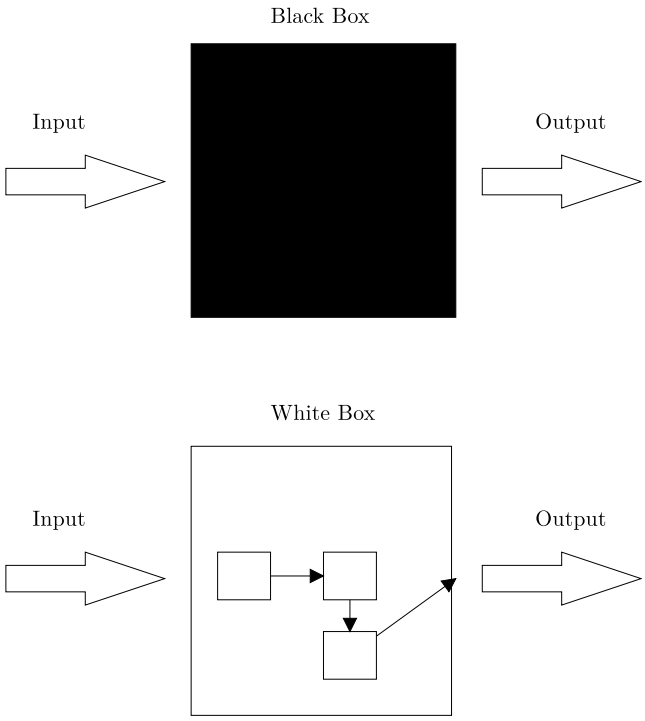

Figure 8: Illustration of black and white box testing methods. [39]

Input selection
There exists different methods for choosing the appropriate inputs for testing. With *exhaustive testing* all possible input combinations are investigated, which usually leads to combinatorial explosion, and the testing of all of them can, in some cases, even take millions of years. *Boundary-value testing* and *Equivalence partitioning*, for

example, solve this issue by having a logical set of combinations and not all possible ones. [39]

Another approach in choosing the right inputs is to look for the requirements and specifications defined for the software [39]. Especially for higher-level testing this approach is preferred. The set of inputs can in such cases also be derived from specified *user documentation* of the program, which defines how the program is to be used and what are the effects of the actions performed [39]. A single *use case* describes how to use the system in order to reach a particular goal [36]. In other words, a use case is a description of an operational scenario of the system.

Test case
A set of inputs along with the test preconditions and expected results form a *test case*. The purpose of a test case is to drive the execution of a testable item to meet the objectives defined for the test case, such as verifying proper implementation, detecting errors and so forth [40]. A collection of test cases with the focus on testing a specific area of a system is referred to as a *test suite*.

Levels of testing

Testing can be carried out at different levels of the system and at each level we investigate different aspects of the system. Usually testing of an entire software program is performed by starting from smaller parts and gradually moving to the larger components of the system [36, 38]. One can define different levels of testing as follows [36, 39, 40]:

Unit testing
Unit testing of software is the most basic level of testing. On this level, individual components of a software program are tested separately. For example, testing the outputs of a given software function is considered a unit test. Usually these tests are written or performed by the person who also wrote that particular part of the software. Black box and white box methods are usually applied for unit tests. Inputs for test cases are commonly derived by using some of the methods for input combination, such as equivalence partitioning.

Integration testing
Integration testing tests the functionality of larger software components, consisting of several smaller units. With this level of testing we ensure that the smaller units interact with each other properly and that the biggest component itself works properly. Both black box and white box methods can be applied to carry out the tests. Inputs for test cases are commonly derived in the same manner as with unit testing.

System testing
On this level, testing is done on a complete integrated software system to see whether it conforms to the requirements specified for it by the development team. With this testing, we see whether the integrated parts of the software work together and also see how the whole system functions. Black box testing is usually applied at this level.

The test cases derive their inputs with some of the methods specified for higher level testing. For example from documented use cases of the system.

Acceptance testing
At this level testing is usually performed by some outside team that had no involvement in the development of the software. This team could be specifically selected final users or there could be a separate testing team to ensure that the final product conforms to the original requirements defined by the orderer of the software. Black box testing methods are applied at this level. The test cases derive their inputs with some of the methods specified for higher level testing. The more commonly used method is to derive the inputs from the documented use cases of the system.

System integration testing
In the interest of embedded systems, system integration testing is performed to test the overall embedded system assembled from sub-components once the sub-components have passed the previous testing levels. System integration testing is performed to verify that the integrated system meets its requirements and for one to detect any issues emerging when the sub-systems are brought together. Black box testing is used to perform this testing. Some of the methods specified for higher level testing are used to derive the inputs for the test cases. [37, 41]

Types of testing

Besides the method and level of testing chosen, there exists multiple types of different testing that can be performed. The most common ones are the following [36, 37, 38]:

Functional testing
Functional testing is performed when we are interested in knowing what the SUT does based on the inputs to it. Black box testing methods are mostly applied here. Functional testing can be done at all levels of testing.

Non-functional testing
Non-functional testing on the other hand is interested in how the SUT operates, rather than what it actually does. Several different testing types can be considered to belong under this category, such as performance testing or security testing for example. Both black and white box methods can be applied to this type of testing.

Performance testing
Performance testing is done for the interest of knowning how stable and responsive the system is under a certain load. This testing can be done at all levels and the methods can vary.

Regression testing
Regression testing is usually performed after the software has changed from the previous version which had been tested. For example, when a new feature is added

or some defects are fixed, regression testing is done to see whether the old parts of the software still work as expected. Usually a fixed set of unchanging test cases can be executed once every change has been made to the software.

Smoke testing
Smoke testing is carried out to verify that the most important parts of the system work properly. Usually the test set is small as we are only interested in seeing if anything fundamental is not working in the system.

2.2.2 Space industry methodologies

In testing of the satellite software at different levels, one has to take into account the effect of different system environments respective to the level of testing [37, 42]. From simulating the target hardware on a computer to running tests on an integrated satellite, different methodologies exist to account for different environments [42]. In addition, satellites consist of several subsystems (as described in section 2.1.1) and each can have their own software. The software of each of these subsystems is tested individually and finally together on the integrated satellite [36, 42].

At NASA, at different levels of system development, different environments and different teams are used for testing [42]. During the course of the Apollo program, NASA adopted the four-level software testing practice [43]. At the lowest level, unit and integration tests of software of a hardware component/subsystem are carried out by the software developers on a desktop environment. At the system and acceptance levels the tests are performed by developers and separate test engineers respectively. Simulators and *testbeds* are used when testing the software at this level [42, 44]. This environment contains assembled subsystems, interface emulators and ground and flight software. The goal is to properly verify the required software functionality. As an example, a system testbed was used to test the operation of singular and several subsystems of the Cassini-Huygens space probe [44]. Several inputs to the subsystems simulated the space environment while tests were being carried out. On the system integration level, the whole spacecraft is assembled and the integrated system is tested with different scenarios of satellite operation [42, 45]. For example, downlink procedures, maneuvres, payload operations and so forth are tested at this level. This test is usually performed by a separate integration test team [42]. Figure 9 describes the levels and methods of testing at different levels of spacecraft development.

At the *European Space Agency* (ESA) the preferred levels for testing of the spacecraft are equipment, subsystem, element, segment and overall system [46]. ESA also states that a system verification by testing shall consist of testing of system performance and functions under representative simulated environments [46]. As can be deducted, testing is done in a way similar to what is done at NASA.

Emphasis on testing at the highest level of assembly or in other words, testing the whole assembled spacecraft has always been a NASA priority [47]. A mantra commonly used in the space industry has been *"Test like you fly"* (TLYF), meaning that a spacecraft should be tested on ground in the same way as it would be operated in-orbit [33, 45]. In general, the TLYF philosophy provides a basis for acquiring and

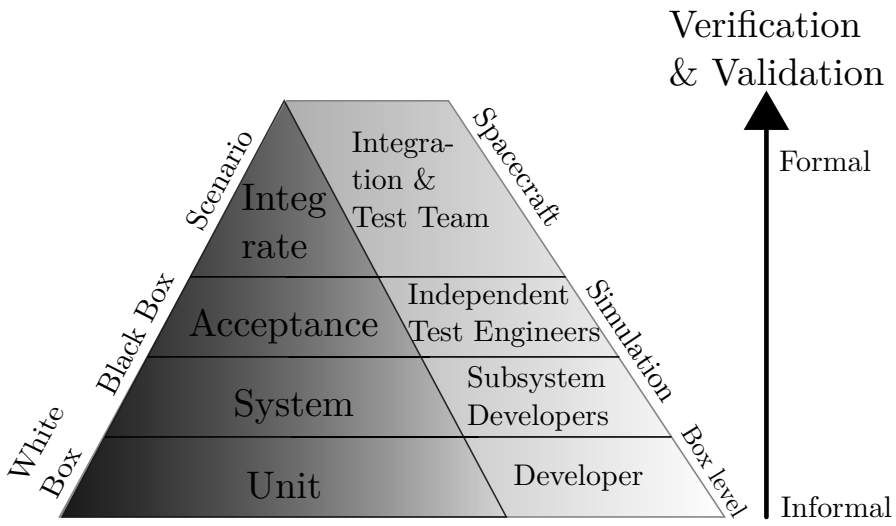

Figure 9: Illustration of spacecraft testing on different levels of system development. [42, 45]

verifying a system and gives a mission-centric focus on space system validation and verification [45]. As such, the same software and hardware should be used in testing as that which will be used when the spacecraft is launched into orbit [33]. One such test on system integration level using this philosophy is commonly referred to as the *"Day in the life"* or operational satellite scenario testing [33, 45].

In the "Day in the life" testing, tests are derived from the mission operations requirements documents [45]. A document called *operations concept document*, or CONOPS, is commonly referred for this type of documentation [45, 48]. The focus with this testing is on verifying whether the space and ground segments can accomplish the mission as it was envisioned in these documents. The test involves having the integrated and assembled spacecraft on the ground being flown in a flight-like manner to the extent feasible. In addition, controlling and communicating with the spacecraft from the ground station is tested in the way that has been envisioned in the mission operations requirements document [45, 48]. This type of testing has been deemed necessary as many failed space missions had actually been succesfully tested to meet all their requirements, but were not tested to verify succesful completion of mission objectives [45]. This test is required by NASA's Goddard Space Flight Center and ESA to be performed before a spacecraft can be verified for mission readiness [45, 46].

It has to be further noted, that space as an environment itself provides extra challenges for maintaining proper reliability of spacecraft. For example, variations in temperature as well as particles carried by the solar wind bring unique demands for reliable design. These devices operating practically out of our physical reach impose further demands for system reliability. If a satellite orbiting at an altitude of 500 km develops an unrecoverable processor error, there is no practical way for us to go there and physically press the reset switch to get the satellite operating again. Therefore, the testing of spacecraft has to be thorough and systematic. Higher demands are

usually set for the testing of spacecraft than for terrestial systems. [23]

2.3 Test automation

Software test automation has been a topic of interest among software projects for the past few decades [49]. It has been heralded as a solution for decreasing costs related to testing and enabling the release of human resources for other tasks [50, 51]. Test automation can be performed on many levels of testing, from unit level to acceptance and beyond. It has been found to be most useful in automation of repetive tasks and in automating execution of repetive test cases [51]. Several software tools and frameworks for test automation have been developed over the brief history of test automation [49, 51].

2.3.1 Test automation frameworks

A test automation framework is an integrated system that sets the rules of automation for a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. When some changes are made to the system under test, only the test cases need to be modified. [52]

A common practice is that test cases are written into separate scripts with a scripting language specific to the framework [19, 52]. Function libraries are written into their own source code files with some of the more common programming languages such as Python, *Java, C++* and so forth [19, 53]. The test scripts then call for the functions in the function libraries to perform the actual automation [53]. For example, a script simulating a network system being used could call for functions in the function library to send commands over the network.

2.3.2 Robot Framework

Robot Framework is generic test automation framework for acceptance testing originally developed in Nokia Networks [19]. The framework emerged from a Master's thesis written by Pekka Klärck for one Finnish software testing consultancy company known as *Qentinel Oy*. The title of his thesis was *"Data-Driven and Keyword-Driven Test Automation Frameworks"* and it was written in 2005 [19, 54]. In turn, the writer of this thesis at hand has also been working at Qentinel and thus has become quite familiar with the Robot Framework. This is one of the reasons why the Robot Framework was chosen for the test automation of the Suomi100 satellite.

Robot Framework is, in addition, open-source under the *Apache 2.0* license, and the modularity of the framework allows people and companies to write their own testing libraries either with Python or Java. The core of the framework is implemented with Python. Instructions for installing the framework can be found from Robot Framework *GitHub*. The modularity and flexibility of the framework has thus made it possible to use it to perform test automation on various different projects. Some companies such as *ABB, Nokia, Kone, Metso, Axon, Zilogic* and others have used the Robot Framework in the testing of embedded systems. Other companies such as *Finnair* have also been utilizing it to test their web based applications. Some

companies such as ABB, Metso and others are performing their testing with the Robot Framework in many different areas. *U.S. Naval research laboratory* has also been using the framework with their *SAGE* multi-agent system. [19]

Based on how many companies have been confidently using the framework [19] and that it has been used in many different areas (embedded systems, web applications, etc.), we feel confident to develop the test automation of the Suomi100 satellite with the Robot Framework. In addition, the framework being open-source makes it even more appealing for this task [19]. Future CubeSat projects could use the Robot Framework as well and possibly also use the generic libraries that are created in this thesis.

Robot Framework uses a *keyword-driven* testing technique and the test scripts have a tabular data syntax. With keyword-driven testing, the test cases consist of keywords and each keyword performs a specified action. The keywords in a Robot Framework test case are executed in order from top to bottom. The keywords themselves can be written to be fairly abstract sentences (which perform many different actions) or can be simple function calls (performing one action). Below in Figure 10 a Robot Framework test suite script is shown. [55]

```
*** Settings ***
Library         String
Library         ../libraries/CubeSatAutomation.py
Suite Setup     Start Suite
Suite Teardown  Client Close

*** Test Cases ***

Basic Communication
        [Documentation]         Communicate and reboot
        [Tags]                  OPMODE-COMM
        Satellite State         Unknown
        Send Command            ping 8
        Clear Replies           All
        Verify Reply Contained  Reply in
        Send Command            reboot 2
        Verify Reply Contained  Welcome to nanomind

EPS Housekeeping Data
        [Documentation]         Call EPS housekeeping routine
        [Tags]                  OPMODE-POWER
        Satellite State         Idle
        Send Command            eps hk
        Verify Reply Message    Voltage
        Verify Reply Message    Input   5       0
        Verify Reply Message    Output  5       0
```

1 2 3 4 5

Figure 10: Example of a Robot Framework test suite with two test cases.

In Figure 10, the number (1) on the script shows how function libraries are included in the test suite. These libraries can simply be Python files or Python classes. In that case, a simple Python file can be included to the suite by defining a relative path and the filename. Python modules which are included in the operating system *PATH* variable can be included without relative paths.

Number (2) on the script illustrates how the test suite setup and teardown scripts are included. These calls to start and close the test execution can also be Python functions included in the function libraries. Call for *Suite Setup* defines what operations are peformed at the start of the test execution, such as opening connection to the SUT, initiating the system to a default state and so forth [55]. Calling for *Suite Teardown* defines what actions are performed when test execution ends [55], such as resetting and closing the SUT.

Number (3) shows in what way the test cases are defined. The names of the test cases are arbitrary. However, they should be named differently from each other and the naming preferably should represent the activity of the test case [55].

The lines next to number (4) present how keywords are called. The keywords can be direct calls to Python functions or methods of the same name, or they can be other keywords defined in some Robot Framework script file. The underscores and letter cases in the Python function definition are translated so that the representive Python function can be called with the keyword in any way the keyword is written [55]. Spaces and letter cases do not concern the keyword when calling for a Python function from the Robot Framework.

Under number (5) the parameters for the representive keywords are defined. These parameters are separated from the keywords and from each other by arbitrary number of tabular spaces [55].

When the script is executed, Suite Setup is first performed and then all the test cases in order from top to bottom are executed. If any keyword in a test case fails, the entire test case is counted as failed. Finally, Suite Teardown is called and the test execution log files are generated in *Hypertext Markup Language* (HTML) by the Robot Framework. Several Robot Framework log files can be seen in Appendix D. [55]

2.4 Suomi100 satellite mission

The software programs related to Suomi100 satellite comprise the SUT in this thesis. The satellite mission was conceived in 2015 in the interest of celebrating Finland's 100 years of independence [56]. The original design called for a 2U CubeSat, but was later changed to a 1U CubeSat. The mission demanded having two payloads on board the satellite. The first payload is a white light camera for taking images of Finland from space, and the second payload is a science instrument which measures the radio static in the Ionosphere [18, 56].

2.4.1 Mission requirements

The requirements and definitions for the mission are presented in this section. The satellite and its software are described in section 3.1. The first requirement defines the functional requirement of the mission as:
"Take images of Finland and measure RF radiation caused natural and man-made sources."

Table 1 below shows functional requirements derived from this requirement.

Table 1: Suomi100 functional mission requirements derived from the requirement to take images of Finland and to measure natural and man-made *Radiofrequency* (RF) radiation

1st Derivation	2nd Derivation	3rd Derivation
Take 1 image of Finland per day	Capable of pointing camera towards Finland	Camera points with 5 degree accuracy
	Must compress images for faster downlinking	RAW, BMP and JPEG output formats
	Image resolution shall be adequate to discern geographical features	250 m / pixel resolution
	Must take images at both day and night time	Polar orbit (SSO noon/midnight)
Capable of measuring entire frequency range at all points over Finland	Payload capable of measuring RF radiation between 1 -10 MHz	1-10 MHz region measured in 6 kHz strips
		Sampling frequency 32-48 kHz
		50 samples from each 6 kHz band
	Adequate resolution for scientific measurements	Radio resolution 16 bits
		AGC resolution 5 bits
	Must compress data for faster downlinking	Calculates summed value of the 50 data points of each frequency band
Can communicate with ground station	Satellite sends and receives data via cubesat space protocol	Ground station uses Cubesat space protocol
	Software includes scheduler	

The second requirement defines the operational requirement as:
"Suomi100 is a CubeSat."

This requirement defines in general that Suomi100 must meet the requirements defined for the CubeSat standard and other operational performance requirements such as power consumption and downlink speed. For the interest of this thesis, going through them in detail is unnecessary. It is necessary only to note that Suomi100 meets the requirements set for the CubeSat standard, and that the requirements for power consumption and downlink speed are met.

2.4.2 Satellite operation modes

As the satellite has several operations it needs to perform, different *operation modes* were identified for the mission. The operation modes are presented in detail below:

Target mode/measurement mode
The payload radio performs several sweeps over the entire frequency range. Because the orientation of the satellite has little effect on the payload radio antenna, the ADCS system is turned off. This is done to mitigate noise caused by the magnetorquers. The OBC calculates the average values of the received signal power to reduce the size of the data. Alternatively, the raw data may also be stored in case the operator requests it. The satellite gathers telemetry at least every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Low observation mode
This mode is similar to Target mode except that the payload radio takes measurements at a single frequency. This mode can be used to track ionosonde signals. The satellite gathers telemetry at least every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Communication mode
Measurement and housekeeping data is sent down to the ground station via the Ultra-High frequency (UHF) link whenever possible, as data downlinking is the most restrictive factor of the mission. This mode is also used to send commands to the satellite. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Power charge mode
Only the essential components of the satellite are operating so that the solar panels can charge the satellite's batteries. Additionally, ADCS is used for optimal solar panel efficiency. Housekeeping is gathered. The satellite gathers telemetry at least every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Imaging mode
The onboard camera is used to take images of the Earth, which requires the ADCS to accurately point the camera toward the Earth. The images are either compressed by the camera or stored as raw images in the internal memory of the camera module. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Software update mode
Similar to the communication mode, the largest data traffic goes now up, with only the most essential telemetry being sent down. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Idle mode (everything goes back to this mode)
The satellite always returns to this state if its not doing any of the other modes. The ADCS is off. The satellite gathers telemetry at least every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Observation & imaging mode
The onboard camera is used to take images of the Earth, which requires the ADCS to accurately point the camera toward the Earth. The images are either compressed by the camera or stored as raw images in the internal memory of the camera module. The payload radio performs several sweeps over the entire frequency range before and/or after the image is taken. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded. This is a data intensive mode.

Debug/status mode
This mode is specifically for checking out the satellite's health. Housekeeping can be gathered as quickly as 10 seconds, beacon is sent every 1 minutes, and all subsystems should be possible to be used. Use examples: e.g. timing of ADCS turning, EPS solar panels functionality check, radio functionality check.

Deployment mode
The satellite starts in this mode - i.e. antennas are ready to deploy, 30 minutes switch-on time for EPS and 45 minute UHF radio beacon broadcast start time are ready to start immediately when the satellite is deployed. The correct commands for the thermal knives that cut the antenna lines are known and ready to start as soon as the EPS starts 30 minutes after deployment.

2.4.3 Instrument modes

In addition to the mission and operation modes, the different operational modes for the Suomi100 payloads were defined as well. For the radio payload, three different modes are defined. For the white light camera, one mode is defined. In addition, a few macro modes containing both of the payloads are defined as well.

First mode for the radio instrument, *Raw data mode*, is tied to the *Low observation* satellite operation mode. With this instrument mode, we use a single frequency to measure the signals in the upper Ionosphere. Table 2 shows the arguments related to this mode.

Similarly as in the first mode, the second mode for the radio instrument is closely related to the *Low observation mode*. In this mode we use a single frequency for the measurements, but individual measurements are not stored. Instead, certain statistical values from a number of individual measurements are calculated and retained for analysis. These statistical values can be either (0) mean, (1) mean & median, (2) mean & median & standard deviation or (3) mean & median & standard deviation & minimum & maximum. Table 3 describes the arguments related to this mode.

Table 2: Raw data mode

Description	Values	Default
Mode starting time		"immediately"
Frequency	1-10 MHz	5 MHz
Number of measurements	>1	100 000
Skipped datapoints	>0	1000
Antenna	0/1	0

Table 3: Average raw data mode

Description	Values	Default
Mode starting time		"immediately"
Frequency	1-10 MHz	5 MHz
Number of stored calculations	>0	100
How many measurements used in calculations	>0	100
Skipped datapoints	>0	1000
Which calculations performed	0,1,2,3	0
Antenna	0/1	0

The third mode for the radio instrument is similar to the second mode and tied to the *Target Mode* operation mode. In this instrument mode, we store statistical data about individual measurements as in mode two. But the frequency we use is varied during the operation of the instrument. The frequency first starts at some value, measurements are made and stored, and then the frequency is increased and measurements are made again. This procedure is performed until some defined maximum frequency is reached. Several cycles of this sort can be performed. Table 4 illustrates the arguments which are part of this instrument mode.

For the camera payload, there is only one instrument mode defined. This mode defines which direction to point the camera, the image quality and other parameters. In addition, this instrument mode is part of the *Imaging mode* operation mode. Table

Table 4: Ramped average data mode

Description	Values	Default
Mode starting time		"immediately"
Starting frequency	0.1-10 MHz	1 MHz
Ending frequency	0.1-10 MHz	10 MHz
Number of frequency values	>0	10
Number of cycles	>0	100
Skipped datapoints	>0	1000
How many measurements used in calculations	>0	100
Which calculations performed	0,1,2,3	0
Antenna	0/1	0

5 describes these parameters in more detail.

Table 5: Photo mode

Description	Values	Default
Mode starting time		"immediately"
Camera direction	1-6	1 (nadir)
Image format (0) RAW (1) BMP (2) JPEG	0-2	2
Exposure time	10000-100000	10 000 microseconds
Auto gain	0/1	0 (No autogain)
JPEG quality	0-100	85

By combining some of the instrument modes for the radio instrument and for the camera, several different macro modes can be constructed. For example, first performing the first mode for the radio instrument and, secondly using the camera with its instrument mode, and finally performing another measurement with radio instrument mode 3.

2.4.4 Automated functional system integration testing

The testing of the Suomi100 satellite is performed for the purpose of (1) verifying subsystem integration and (2) satellite reliable operation as well as (3) verifying that the satellite meets its functional requirements. The functional requirements of the mission are described in sections 2.4.1, 2.4.2 and 2.4.3. Test cases are derived from the operation modes presented in section 2.4.2.

In this thesis, we have chosen *automated functional system integration testing* to cover the type of tests that we perform for the Suomi100 satellite. The testing can be identified to be Black box testing. *Automated* refers to the fact that an automated testing framework is used for test execution. *Functional* comes from the reason that we test the functions the satellite performs based on the selected inputs and commands. *System integration* refers to the fact that testing is performed on the whole integrated satellite. Testing is carried out this way because, as mentioned earlier in section 2.1, this kind of testing has most likely been lacking in previous CubeSat missions. Moreover, carrying out these tests could possibly mitigate failures that occur early in the life of a satellite. In addition, we attempt to verify that the satellite functions in accordance with the requirements set for it.

Features to be tested
Based from the research represented in section 2.1, the testing will focus on testing of features that have been believed to have caused failures with CubeSats, or that testing of them has been inadequate. In order to ensure a successful Suomi100 mission, tests will be performed for the two payloads as well.

From the operation modes, four different conglomerates of features are identified for testing: (1) functionality of the camera payload, (2) functionality of the radio payload, (3) reliable operations of the basic software features of the satellite such as housekeeping, safe reboots, software updates and so forth. Tests for the (4) "Day in the life" operational mission scenario testing will be performed likewise.

Approach to testing
Testing will focus on functional testing and it is performed at *System integration level* for all four features. Test automation is used in test execution, and the tool for test automation is the Robot Framework. The functional environment for each respective feature is to be simulated by inputs external to the satellite. For testing of the operation of the camera, natural light is used as an input. Testing of the radio payload will use externally generated radio signals as input. The tests for the "Day in the life" scenarios will use a solar simulator, and the satellite will be commanded over a radio link. All these tests will be performed for the integrated satellite.

Test case Pass/Fail criteria
All tests are considered critical, thus a failure in execution of one test step (one Robot Framework keyword) in a test case leads to test case failure. In addition, failure in a single test case marks a test suite as having failed. Test steps are failed based on the responses of the satellite control software.

3 Methods and Setup

3.1 Suomi100 satellite

Suomi100 satellite is assembled together with a 1U CubeSat platform manufactured and designed by *Gomspace* company from Denmark. This 1U CubeSat, which is known as *NanoEye* in the Gomspace product catalogue, forms the structure and bus systems of the satellite [57]. This part of the satellite is referred to as the *platform* in this thesis henceforth. A picture of the platform is shown in Figure 11. On top of the platform, another payload was added, consisting of an *Amplitude Modulated* (AM) radio on a PC-104 type PCB, two ferrite antennas and a support structure. All were designed and assembled at Aalto university by members of the Suomi100 satellite team. This part of the satellite is referred to as the *radio payload* in this thesis.

The subsystems and the satellite platform have flown in space aboard other missions successfully [58]. The platform forms a relatively well tested system with which we can investigate the development of automated functional system tests for CubeSats [58]. In addition, the radio payload and its control software integrated to the platform give another aspect for study. Namely, how to test the integration of a subsystem with the rest of the satellite, as all the rest of the subsystems were integrated by GomSpace.

One main mission goal of the Suomi100 satellite is to take pictures of the northern hemisphere, especially of Finland. The satellite flies in the upper Ionosphere in a polar orbit at an approximate altitude of 500 kilometers. With the radio payload a noise-map and natural noise levels in this area of the Ionosphere could be mapped out.

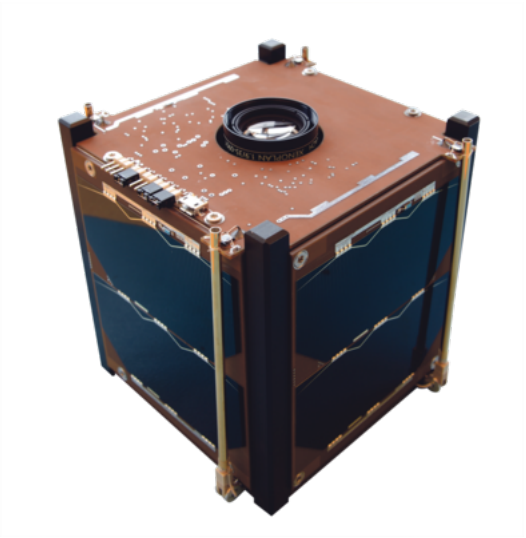

Figure 11: GomSpace NanoEye 1U. Courtesy of GomSpace A/S. [57]

3.1.1 Subsystems

The satellite consists of several subsystems. The central subsystem of any satellite is the system with the computer designated as the OBC. In the Suomi100 platform it is known as *NanoMind* and is based on an *Atmel* 32-bit microcontroller [59]. Another vital system to the satellite is naturally the EPS and it is known as *NanoPower* in the platform [60]. The communication system of a satellite is the system responsible for receiving commands from the ground, and is responsible for sending information back to the ground as well. In the platform the communication system is known as *NanoCom* [61].

Besides these essential systems common to all satellites, we have as payload an optical white light wide angle Earth-observing camera and the radio payload measuring *Medium/High frequencies* (MF/HF). The camera came along with the GomSpace platform and is known as *NanoCam* in their catalogue [62]. The most essential subsystems to the topic of this thesis are described in more detail in this section.

On-Board Computer - Nanomind

The Nanomind A3200 On-Board-Computer shown in Figure 12, is based on an *Atmel AT32UC3C* model *Microcontroller unit* (MCU), which is a 32-bit *Reduced Instruction Set Computer* (RISC) with advanced power saving features. This system runs the software that is responsible for the majority of operations of the satellite, and it works as a sort of mediator between subsystems and routes communication between them. The software is explained in more detail in the following subsection.

The MCU has two *Inter-Integrated Circuit* (I2C) buses and one *Controller Area Network* (CAN) bus for communication with other subsystems. It has also 8 *Analog to Digital Conversion* (ADC) pins, which can also be programmed to work as *General Purpose Input-Output* (GPIO) pins. Nanomind contains a *Synchronous Dynamic Random Access Memory* (SDRAM) with 32 MB of capacity for volatile storage as well. For non-volatile storage, the subsystem has a 128 MB NOR Flash. Figure 13 shows a block diagram of the OBC. [59]

Figure 12: Nanomind OBC inside its casing. Courtesy of GomSpace A/S. [59]

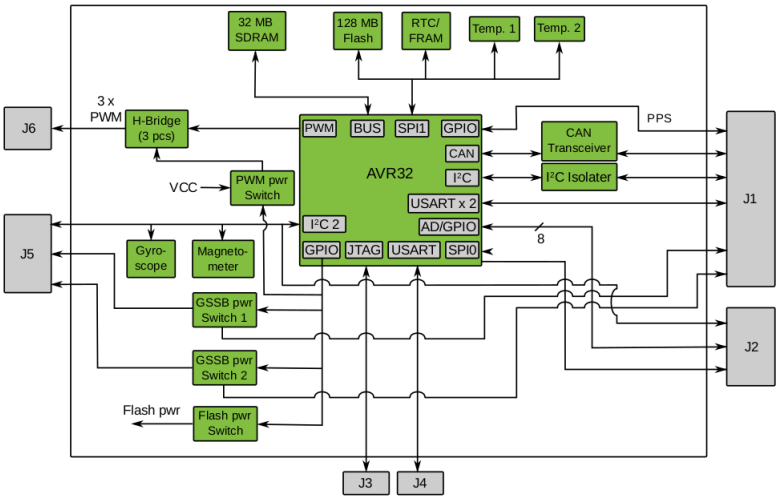

Figure 13: Block diagram of Nanomind. Courtesy of GomSpace A/S. [59]

Electrical Power System - NanoPower

The Nanopower P31 on Suomi100 satellite contains two lithium-ion batteries and has several reliability features. Figure 14 shows a picture of the subsystem. The batteries are charged by the five solar panels aboard the satellite and the batteries provide power to the whole satellite through the stack connector on the PCB of the EPS subsystem. The system has its own microcontroller, which measures the voltages, currents and temperatures of the system. The microcontroller can also be used to control the 5 V and 3.3 V power buses of the EPS, among other features of the MCU. [60]

Figure 14: Nanopower EPS. Courtesy of GomSpace A/S. [60]

Communication subsystem - NanoCom

The NanoCom COM system shown in Figure 15 is a software configurable half-duplex transceiver designed for long-range transmissions. Certain parameters of the system can be reconfigured on-orbit, such as frequency, bitrate, modulation type and filter-bandwidth. Data rates can be between 0.1 - 115.2 kb/s. The subsystem has its

own microcontroller as well as essential radio elements such as *Power Amplifier* (PA) and *Low-noise amplifier* (LNA). [61]

Figure 15: NanoCom communication system. Courtesy of GomSpace A/S. [61]

Camera payload - NanoCam

First of the payloads in Suomi100 satellite is the NanoCam wide-angle white

Figure 16: NanoCam payload camera. Courtesy of GomSpace A/S. [62]

light camera, presented in Figure 16. The subsystem consists of a lens, image acquisition and processing board. The lens is an industrial grade lens and the image acquisition element is an *Aptina MT9T031* 3-megapixel *Complementary Metal Oxide Semiconductor* (CMOS) color sensor. The processing element consists of a PCB with components such as an *Atmel SAMA5D35* processor with a clock rate of 536 MHz, 512 MB of DDR2 memory for image storing and processing, and a 4 GB eMMC flash drive with 2 GB for image storing. [62]

The software for image processing and storing runs on a customized embedded *Linux* (GomSpace Linux) operating system, and there are several features for image acquisition and storing. The images can be stored in either *RAW, BMP* or *JPEG* formats. Several parameters of the camera system can be altered while in orbit, such as exposure time, different gain values, gamma correction and so forth. [62]

Radio payload

The second payload of Suomi100 satellite is the AM radio payload. As noted, this payload was developed by the Suomi100 satellite team, namely by M.Sc Petri Koskimaa, B.Sc Amin Modabberian and B.Sc Arno Alho, based on the concept of the

ground-based lightning detector envisioned by Ph.D. Jakke S. Mäkelä [63, 64, 65, 66]. Figure 17 shows the PCB of this subsystem. Central to the system is the *Silicon Labs Si4740* automotive *Amplitude Modulated/Frequency Modulated* (AM/FM) Radio receiver on an *Integrated Circuit* (IC) [67]. It can receive signals with frequencies from 149 kHz to 23 MHz in 1 kHz steps. The Si4740 can be set to receive AM, AM/SW/LW or FM signals. Several features of the IC can be modified. These include frequency, volume, output format, sample rate, attack rate, release rate and many more. Commands to the Si4740 are sent via the I2C bus and the output of the receiver is read via the SPI bus [68].

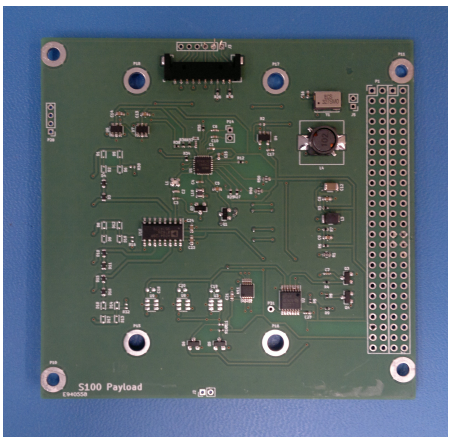

Figure 17: Second payload of Suomi100, AM radio instrument. Courtesy of Aalto University.

Another important element of this subsystem are the antennas and their support structure. The antennas were designed by M.Sc Petri Koskimaa. Design and construction of the antennas are described in his Master's thesis, "Ferrite Rod Antenna in a Nanosatellite Medium and High Frequency Radio" [69]. These antennas are four ferrite rods, with two on either side of the support structure forming one antenna. The first antenna is used when listening to frequencies below 2 MHz, and the second one is for frequencies between 1.0 and 9.3 MHz.

The support structure for the antennas was developed by Ph.D Antti Kestilä and it was made with a 3D printer using *Ultem* plastic, a material that can sustain the extreme environment in space relatively well [70].

3.1.2 Gomspace software

Besides the subsystems for the NanoEye platform, GomSpace also provided software for all these subsystems. The essential core of the software architecture is a delivery protocol known as the *CubeSat Space Protocol* (CSP), which was originally developed in 2008 by a group of students from *Aalborg University* in Denmark. The protocol has further been developed and maintained by GomSpace itself. In practice, the protocol is used for communication between different subsystems as well as with the ground station. Different subsystems are considered as different CSP nodes in the CSP network. [71]

The protocol as well as the software for the subsystems were written in *C* programming language. In addition to the specific software for each subsystem, all the systems share a set of common functionalities. These common functionalities include sending and storing of HK data, parameter tables for adjusting the different functionalities of a given subsystem, logging functions and inter-subsystem communication through CSP. In addition, each subsystem provides a terminal shell known as GomSpace Shell or *GOSH* for control of the subsystem via a PC by using the *Minicom* software. [71]

The software developed by GomSpace for NanoEye additionally includes such general functionalities as the *File Transfer Protocol* (FTP) running over CSP, with which files and data can be uploaded and downloaded from the satellite. Certain basic file handling routines can be handled with the FTP as well. Among the file handling functionalities is the ability to compress or decompress files with the *ZIP* format. Additionally, the software in the satellite can be updated via the FTP by uploading a software image to the satellite and telling the computer to start reading from it after next reboot. In addition to these, the *Flight Planner* is another general feature of the platform and with it commands can be set to execute at certain points in time either once or repeately with some interval. [71]

The operating system running in the NanoMind OBC is a free *Real-time Operating System* (RTOS) known as *FreeRTOS*, which is a lightweight operating system designed for embedded systems that use microcontrollers and small microproserssors [72]. It was developed by *Real Time Engineers Ltd.* in USA. The version of the operating system used in the Suomi100 satellite is 8.0. The operating system is mostly written in C programming language, but certain necessary parts are written with the *Assembly* programming language.

FreeRTOS is a real-time scheduler where different tasks execute in a *Round Robin* fashion, where each task is given some priority value, and tasks with higher priority value are given more processing time. Those with the same priority value take turns in the execution of instructions. Only one task at a time can be in a running state and all the others wait for their turn according to the scheduling policy. In addition to scheduling, the operating system offers functionalities for inter-task communication via *semaphores*, for example. [72]

3.1.3 Satellite control software - CSP Client

The ground station software used to control the satellite is known as the *CSP Client*, which is a simple console program for remotely sending commands to the satellite via CSP, a program written by GomSpace in C programming language. The syntax of the software is almost identical to the Gomspace Shell found in the subsystems manufactured by GomSpace. As the source code is available to us, we were able to add our own commands to control the radio payload among other things. In Figure 18 the CSP client is shown running in *Debian 9* Linux, showing among other commands a command inquiring for housekeeping data from the EPS subsystem.

The software has over a hundred commands if the subcommands related to the main commands are counted. Thus only the main commands used in test automation

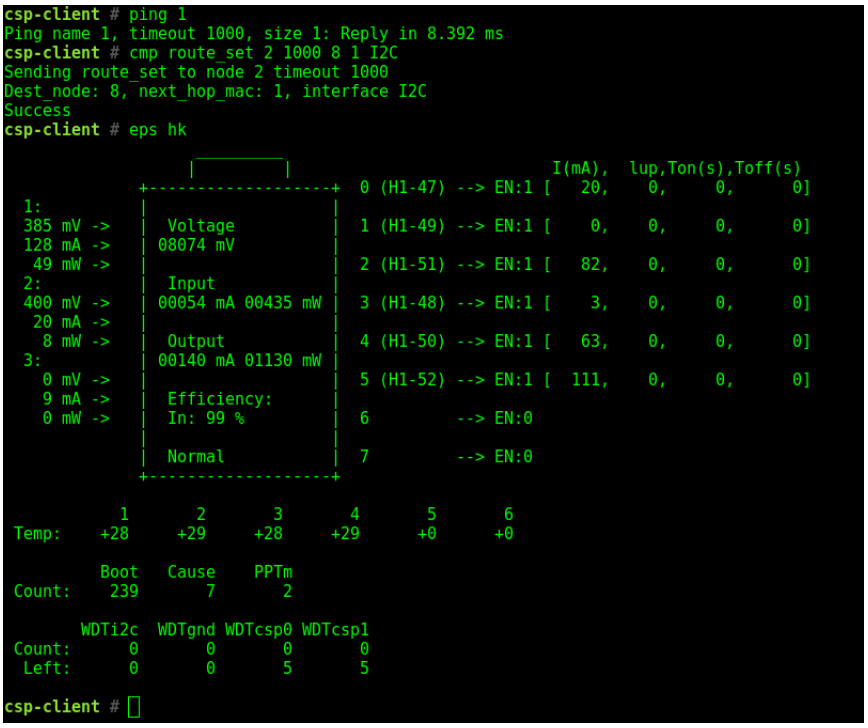

Figure 18: Suomi100 ground station control software running in Linux.

of Suomi100 are presented here:

reboot <CSP node>
Reboots a CSP node.
shutdown <CSP node>
Shutdowns a CSP node.
cmp route_set <node> <timeout> <addr> <mac> <ifstr>
Defines a routing path within the CSP network.
rdpopt <window size> <conn timeout> <packet timeout> <delayed ACKs> <ACK timeout> <ACK delay count>
Configures parameters for CSP packet transmissions over radio link.
hk get <type> <interval> <count> <t0> <path>
Get housekeeping data of certain type. Can be received periodically and the data can be stored in the satellite as well.
eps hk
Get housekeeping data from the EPS.
ping <CSP node> <timeout>
Test reachability of a certain CSP node within the CSP network.
rparam download <CSP node> <mem>
Download configuration parameters from a CSP node.
rparam set <name> <value>
Set configuration parameter value.
rparam get <name>

Get the value of a configuration parameter.
rparam send
Send the defined parameters back to the satellite.
fp server <node> <port>
Sets the flight planner server to a defined CSP node.
fp create <name> [+]<sec> <command> [repeat] [state]
Creates a flight plan with a defined name and command.
cam snap [-a][-d <delay>][-h <color>][-i][-n <count>][-r][-s][-t][-x]
Takes a picture with the camera on the NanoEye platform. Several features can be defined on or off, such as automatic gain, image thumbnail and so forth.

3.1.4 Software for radio payload

The software for controlling the AM radio instrument was developed by B.Sc Juha-Matti Lukkari, the author of this thesis, and by M.Sc Jouni Rynö from the *Finnish Meteorological Institution*. Unlike most of the subsystems in Suomi100, the radio payload has no microcontroller of its own or any other general purpose computer. Therefore, the control software operates as a few FreeRTOS tasks in the NanoMind. In addition, new commands for operating the payload instrument were added to the CSP client as well as to the NanoMind GOSH terminal.

One of the radio payload FreeRTOS tasks receives a command as a CSP packet, which is then parsed as a command to be sent via the I2C bus on NanoMind to the Si4740 IC, for example. Command of the Si4740 is based on the hexadecimal values of the bytes it receives [68]. The first byte received defines which action is being performed and the following bytes define the arguments for that respective action [68]. The IC then gives a response byte, with hexadecimals 80 and 81 implying a succesful command, and ,for example, 40 or 0 implying a failed command [68].

Over 50 different arguments for different commands can be used when controlling the Si4740 [68]. Therefore, when commanding the radio payload to perform a measurement, the different values for different arguments are loaded either from a configuration file or from a GomSpace paramater table. The parameter table and the configuration file were separately added to the NanoMind. All the commands can choose to use either one of these argument sources. In addition, the commands can be used "manually" without loading any external configuration for the command.

Some of the most essential commands for the radio payload operation in CSP client are presented here:

radio on <config> <reg>
Turns on the Si4740 chip, among various other operations needed to setup the payload.
radio operation <config> <config> <mode> <mode arguments>
Runs one of the radio operation modes defined in section 2.5.3.
radio set_property <config> <property> <value>
Sets one property of the Si4740 to a defined value.

radio get_property <property>
Gets the value of a certain property defined in the Si4740.

3.2 Automating testing of Suomi100

One of the *research purpose and goals* pointed in section 1.4 was the aim of using test automation to perform testing for the Suomi100 satellite, and the Robot Framework was defined as the test automation framework for this task. The detailed technical solution for automating the testing of Suomi100 is described in this section.

3.2.1 API and communication layers for CSP Client software

In order to automate the control of the satellite, some form of interface is needed which can communicate between the satellite and the framework used to perform the tests. Fortunately, the Gomspace software already provides a terminal shell program called *GOSH* on each of the subsystems [59]. In addition, all of the subsystems can be controlled from a single shell via a serial to a *USB FTDI* cable connected to *NanoUtil* USB port [73]. As presented in the previous subsection, a separate CSP client software tool exists, which can be used to control the satellite from the ground station via a radio link, and it can also be used to control the satellite via the FTDI cable.

Automating the control of the CSP client software was chosen as the solution on how to automate the control of the satellite. The CSP client was chosen, because by automating control of it, we can perform tests via the radio link as well. The automation was first done by modifying the source code of the *main.c* file of the CSP client, which contains the C-language main function for the program. The modification consists of creating a *POSIX thread* which runs a function that opens and listens to a socket connection on the *localhost* local network address. The localhost is a specific network address referring to the computer itself [74]. When a message is received on the opened socket, the thread then runs the command on the CSP client terminal, as if a user would have written the command on the terminal. Alternative solutions could have been used, for example a separate program could have been written and the CSP client could have communicated with it through some of the inter-process communication methods provided by the Linux operating system. This could have been made through Linux output and input *standard stream* redirection methods such as *pipes* [74]. Using the network connection, however gives the potential to make the automation externally controllable.

Over the course of development of the libraries and test suites, a more direct approach of using the *standard input stream* (stdin), to send commands to the CSP client was also chosen. Furthermore, an even more direct method of simply automating the keypresses of the keyboard was implemented with the aid of a Python library called *pyautogui*. The benefit of having the communication performed with stdin or with automated keypressing through a Linux *kernel* keyboard driver, is that we can automate the use of not just the CSP client but the use of many different

terminal programs. Even programs with source codes that we have no access to, thus omitting the need to write a separate *Application Programming Interface* (API) into them as was done with Suomi100. Nonetheless, using self-tailored process communication APIs which work via e.g. pipes or sockets, have some advantages over these sort of "crude" methods. For example, use of stdin can be reserved to the program in a way that it is not accesible outside the program itself. Sending the commands by automating keypresses can bypass this. However, if a user uses the computer during testing, the keypresses can be received by programs that we did not wish to automate.

Nonetheless, in order to create a generic test automation library, all of these process communication methods are incorporated into the release version of the CubeSat test automation function library, which is explained in more detail in section 4. If we ourselves write the terminal software with our own testing library in mind, all of these communication methods should be valid for automating the testing.

Besides requiring the method of sending commands to the satellite to be performed in an automated fashion, we also must know how the satellite responds to these commands in order to verify the tests as either passed or failed. The CSP client software fortunately receives responds to the commands sent to the satellite, and thus we have some knowledge of how the satellite behaved. It was found that the easiest solution would be to read the *standard output stream* (stdout) of the CSP client program and transfer the responds to the verification functions in the function library.

Another way for the capture of the responses to the commands would be to modify the source code of the CSP client for it to send the received outputs of the executed commands to another port on the socket connection. In this way we could then listen to this port on the function library. Doing the transmission of CSP client output to the test automation libraries this way was experimented by using some Linux output redirection routines such as *dup* [74]. However, there were some difficulties with the implementation, and due to time constraints it was easier to monitor the standard output of the client software. Furthermore as mentioned before, during the development of the test automation libraries, use of stdin for communication was developed as well. In fact, as with using stdin to send commands to the process, reading the stdout of the process allows us to create a generic test verification solution to this as well, provided that the process which we wish to perform automated tests on responds through the stdout stream, which fortunately happens to be the case for most terminal programs [74].

The solution for the communication is illustrated in Figure 19 and the modified *main.c* for the CSP client can be found in Appendix B.

3.2.2 Python libraries

A set of function libraries using Python programming language were written. All of these libraries each consisted of one Python class. The class of the core library, known as *CubeSatAutomation*, has the methods for the communication with the CSP client via any of the three methods, socket, stdin or automated keypressing via

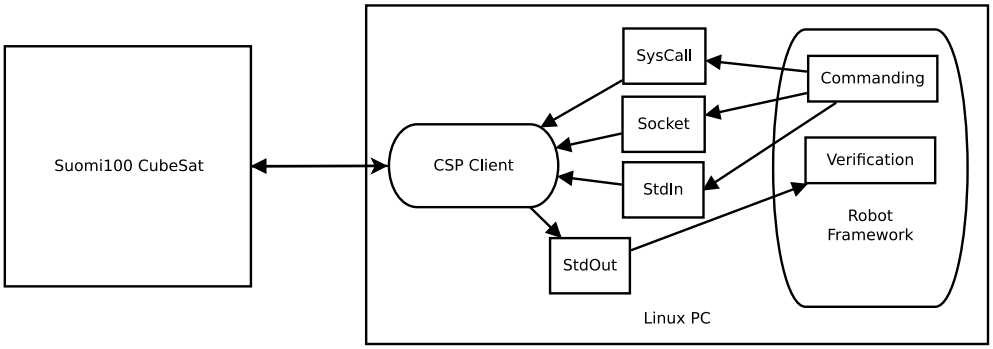

Figure 19: Illustration of the software architecture developed for Suomi100 test automation. Large rectangles represent environments, small ones represent layers and rounded rectangles represent programs.

the pyautogui library. Furthermore, the library includes the methods to read and verify the process replies from stdout. As can be seen in Figure 19, the commands are sent to the CSP client through any of the three communication routes and the output of the program goes to the stdout. The output is then caught and read by the CubeSatAutomation library and test cases and test steps or keywords are failed or passed based on the output read from the CSP client.

It is importation to note the following details about the test automation libraries.

CubeSatAutomation library

CubeSatAutomation library has the crucial functions for sending commands, opening socket connection, opening and closing the opened program and others. Besides being able to open the CSP client program, any program can be automatically opened with the library as the method for opening uses the standard Python *subprocess* library. All the other libraries implemented use the core methods in CubeSatAutomation, for example, to send commands to the CSP client to execute. These other libraries have subsystem-specific functions for the test automation. To create only one open communication route between the CSP client and the Robot Framework and to have only one handle on the CSP client program process, the core library defines these as *class variables*, which are then accessed by the subsystem libraries. In practice this means that we do not open several CSP client programs and several connection routes separately for each subsystem library included in the test suite. Instead, the program and the communication route is opened only once for each test suite.

Another class variable which defines the scope of the instance of the class in the Robot Framework was defined as well. This was set to define the scope of the library to be on the suite level. By having the scope on the suite level, only one instance of the library class is declared per test suite, thus again having only one handle on the client software and having only one connection route open during the execution of a test suite [55].

The essential core methods used in the CubeSatAutomation Python class are presented here in the form of Robot Framework keywords:

Client Start <config file> <program> <parameters>
Start the program that is to be automated (CSP client). In addition, command line parameters as well as some additional configurations can be defined.
Client Close <socket> <program>
Close the program and the possible socket assiocated with it.
Connect Socket <config file> <server> <port>
Opens a socket connection to a defined server and port address. These values can be read from a configuration file as well.
Send Command <message> <option> <timeout> <read timeout>
Sends a command through the socket connection and reads a reply from the standard output. The reply can be stored temporarily or discarded.
Write Command <message> <option> <timeout> <read timeout>
Writes a command to the standard input and reads a reply from the standard output. The reply can be stored temporarily or discarded.
Type Command <message> <option> <timeout> <read timeout>
Types a command by automating keypresses on the keyboard and reads a reply from the standard output. The reply can be stored temporarily or discarded.
Persistent Command <message> <exception replies> <end reply> <timeout> <read timeout>
Writes a command persistently to the standard output until a defined reply is read or either a specific error reply is read or a timeout value is reached.
Verify Reply Contains <message> <timeout> <read timeout>
Reads several lines from the standard output and tries to find for a defined message from the lines.
Verify Reply Contains Not <message> <timeout> <read timeout>
Reads several lines from the standard output and tries not to find for a defined message from the lines.
Verify Reply Contained <message> <timeout> <read timeout>
Tries to find a defined message from the replies that were stored by an earlier command.
Wait Until Reply Contains <message> <timeout> <read timeout>
Reads from the standard output until a defined message is found or a timeout is reached.

Finally, here are some of the keywords presented that are specific to GomSpace NanoEye.

Set Satellite Parameter <device> <parameter> <value>
Sets value of a configuration parameter of a defined device in the CSP network.
Send Satellite Parameters
Sends the set parameters back to the satellite.

Creation of a skeleton core library is the aim of the final development of the test automation library. This library only has the aforementioned methods to start and communicate with a desired Linux program running on a terminal shell and keywords that are more specific to the Suomi100 or CSP client are omitted. With the aid of this library, satellite software developers wishing to automate testing of their satellite and satellite software, can create their own specific libraries suited to their own needs. The final version of the CubeSat test automation library is described in section 4.

Subsystem libraries
Other libraries developed for test automation of Suomi100 are called *NanoCam.py* and *RadioPayload.py* and these are intended for the automated testing of NanoCam and radio payload subsystems respectively. The classes of both of these create an instance of CubeSatAutomation class, instead of calling for specific functions or methods of that class from the outside. By doing this, the class variables including handle to the automated process, socket address and others are passed to all these other classes as well. The methods of these classes thus use the methods of CubeSatAutomation directly.
The specific methods defined by the NanoCam test automation library are presented below as Robot Framework keywords:

Camera Startup <timeout>
Reboots the camera and downloads parameter Table 1 from the subsystem. Timeout specifies the time that we wait for the camera subsystem to come online in satellite bus.
Camera Take Picture <timeout> <store format> <filename> <autogain>
Sets image format and filename in the camera and takes a picture with the given autogain value (empty at default). Keyword fails if the image is too dark (less 5 % light) or too bright (over 95 % light).
Camera Load Picture <stored file> <loaded file>
Downloads the file stored in NanoCam to the PC running the CSP client program.

The subsystem specific keywords for the radio payload are defined as the following:

Radio Startup <switch input> <switch power> <antenna input>
Sets up and starts the radio payload. The antenna and switch to be used can be additionally defined.
Radio Powerdown
Turns off the radio payload.
Verify Radio Status
Checks for the status of the Si4740 chip.
Run Radio Mode <parameter file> <property file> <mode> <mode arguments
Runs one of the radio operation modes defined in section 2.5.3.

Verify Radio Results <buffer file> <timeout>
After the operation mode is completed, inspects the outputs that the payload sent and fails if certain outputs contained errors tied to the operation of the Si4740 chip.
Radio Load Data <stored file> <loaded file> <timeout>
Downloads a measurement done by the radio payload.
Radio Plot Data <file> <output file> <plot image>
Draws a graph from downloaded radio measurement data.

3.2.3 Robot framework test suites

The test cases follow the keyword-driven approach and the keywords are written to be short and mostly to be non-specific to the test case. The functions and methods written in Python and described in the previous section are used directly as such. This is because, the approach was to make a smaller set of versatile and generic keywords that could be used over many test cases and test suites. This approach was felt to be more efficient as there would be less need to maintain the test suites if they did not have large set of specific, though descriptive keywords, which is common with the Robot Framework. Besides, the Suomi100 satellite project is not a typical software project where stakeholders would go over the test suites and validate them. All people involved in the project have a technical background. In addition, having a set of general keywords that are not entirely tied to Suomi100 is beneficial if some future satellite project wishes to use the testing methods and tools described in this thesis.

Each of the test cases are tied to a particular operation mode. The operation modes are discussed in detail in section 2.5. The purpose of each test case is to verify the functionality of some aspect of a particular operation mode. Each test case is marked with the Robot Framework *[Tags]* marker to identify which operation mode the test case is related to. In addition, each test case begins with a ***Satellite State*** keyword defining the state of the satellite. For example, one such state is when the satellite has restarted itself. This keyword was written in order to make the test cases independent of each other and to have a degree of reproducability for the tests. In some cases, the test cases need to be dependant on each other and in such cases the satellite is not specifically set to a certain state. Such states are called as *Unknown* and *Communicating*, for example.

The test suites are divided firstly based on the four different larger features that are tested with the Suomi100. Namely, separate sets of test suites are written for camera payload, radio payload, NanoEye basic functionality and for the "Day in the life" testing. Each aspect of a feature further divides the test suites into test suites testing different parts of a particular feature.

3.3 Test setups and environment simulation

The different aggregates for testing of the Suomi100 were defined in section 2.4. For simulating the functional environment of the satellite for these different types of tests,

four different environments were set up. Two different environments for testing two different payloads (1 & 2), one for testing basic operational features of the NanoEye platform (3) and one larger environment for the operational scenario testing of the satellite (4).

3.3.1 Camera payload testing

For the testing of the NanoCam and the *imaging operation mode*, we tried to find something facing the camera with similar color and brightness values as what the camera would see while in orbit. The easiest solution would be to simply take the whole integrated satellite outside on a bright day to the balcony on top of the *TUAS* building in Aalto University at Maarintie 8. The satellite stands on top of a stand, and the side with the camera lense is directed towards horizon. A PC with the CSP client and the test automation tools are connected to the satellite via the USB connection on the NanoUtil. In addition, the satellite is loosely enclosed in a plastic container to protect it from particles in the air. Figure 20 below shows the test setup used during the testing.

Figure 20: Suomi100 satellite on a balcony during imaging mode tests.

3.3.2 Radio payload testing

The environment for the testing of the radio payload is set up in the clean room of Aalto University's space laboratory. The satellite is connected via the USB connection to a PC with the CSP client software and the test automation tools. The functional environment is simulated with a radio signal source in order to create some artificial noise in radio frequencies that would mimic the radio signals present in the Ionosphere. The radio noise is generated with a *HackRF One Software Defined*

Figure 21: Radio payload testing with *HackRF One*.

Radio (SDR), which is connected to another PC running *GNURadio* signal processing software. Figure 21 shows the setup for the testing of the radio payload.

The frequencies that are used in the environment simulation are 2 MHz, 5 MHz and 9 MHz. These were chosen based on the requirements of the payload (should operate in range of 1-10 MHz) and the limitations of the hardware, as the HackRF is not able to produce signals with frequencies lower than 2 MHz. Furthermore, the antennas attached to the payload themselves cannot receive signals that are much higher than 9 MHz. The middle frequency was chosen to be 5 MHz based on the research done on the radio signals in the Ionosphere. This frequency would be of most interest to the research conducted by the Suomi100 satellite mission [75]. Figure 22 shows a *Fast Fourier Transform* (FFT) plot of the noise that is generated from the GNURadio, which is then transformed into radio waves by the HackRF.

3.3.3 Satellite basic operations testing

For testing of the basic satellite operational features such as collection of housekeeping and safe rebooting during an error, no external inputs to the satellite are used. The satellite is in the clean room of Aalto University's space laboratory connected to a PC with the CSP client. In Figure 23 we have a picture of this setup.

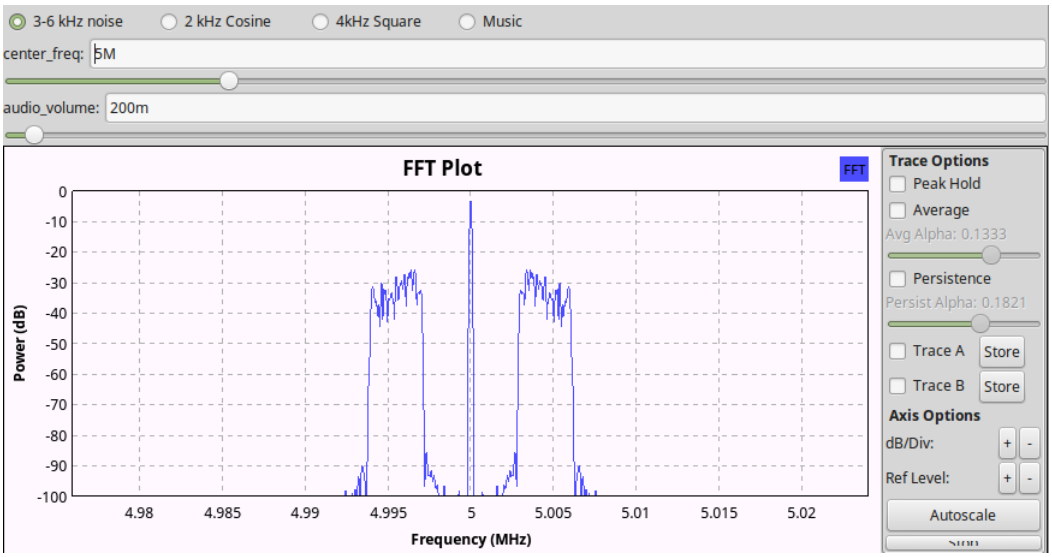

Figure 22: Screenshot from *GNURadio* signal processing software showing FFT plot of radio noise being generated.

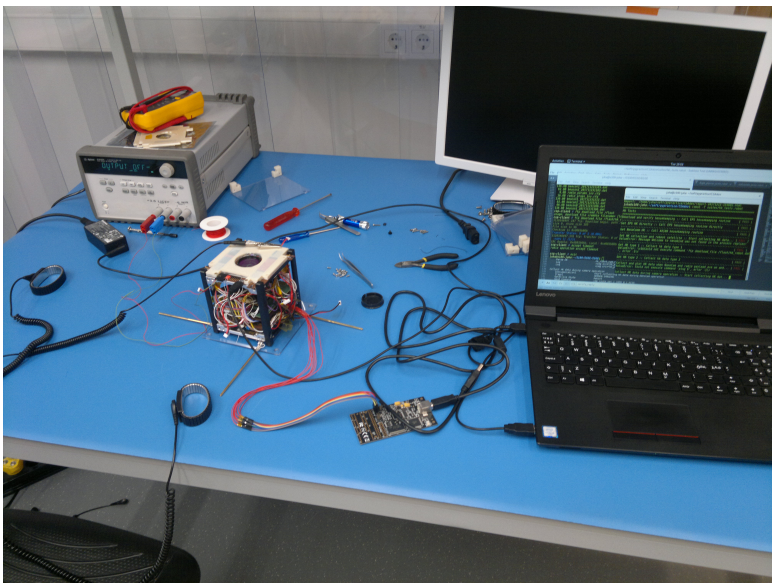

Figure 23: Setup for testing of basic functionalities of the NanoEye platform. Suomi100 is on the left and a PC with CSP client, and a PC with the CSP client and the test automation softwares is on the right.

3.3.4 Operational scenario testing, "Day in the life"

In the "Day in the life" mission operational tests, the Sun is simulated with a 1800 Watt Xenon lamp that is situated approximately 1.5 meters away from the satellite. Two solar panels are connected to the satellite in the manner they are connected during flight. The satellite faces the lamp at an angle so that both panels receive light from the lamp. As the lamp is quite powerful, we can really verify that the solar panels charge the batteries in the satellite. In addition, the lamp can heat

the objects it is faced towards, and this was used as a method to add some thermal features to the test. The idea is to use the lamp for the duration it takes the satellite to heat up to 50 degrees Celcius under the illumination and then let it cool back to room temperature (approx. 27 degrees Celsius in the clean room). The heating and cooling was measured with an *FLIR E6* thermal camera, and the heating duration was measured to be approximately 10 minutes and the cooling down period was measured to last ca. 20 minutes. In Figure 24 this setup with the solar simulator is presented.

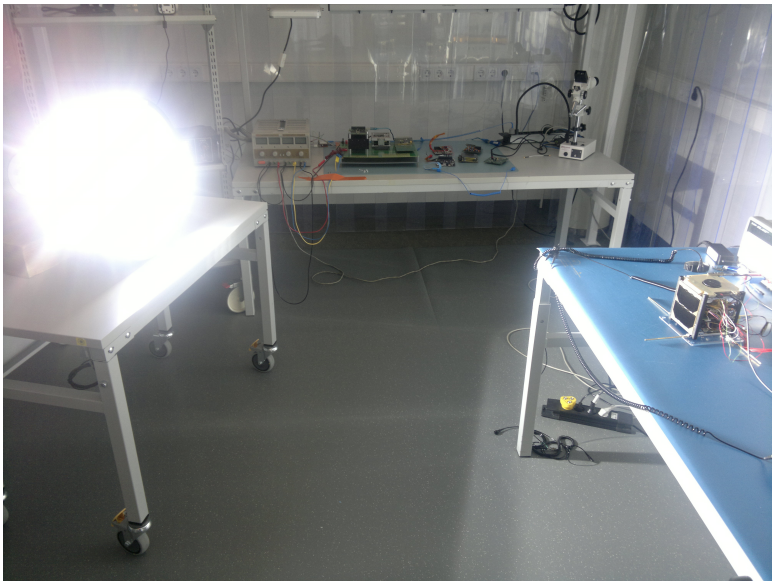

Figure 24: Day in the life setup for the satellite. The Xenon lamp is on the left and Suomi100 CubeSat is on the right in the picture.

For the "Day in the life" testing, these periods form the phases of the operational mission scenarios. When we pretend that the satellite comes from eclipse, we turn on the Xenon lamp and we are in communication with the satellite for 10 minutes. After that, the lamp is turned off and we pretend that the satellite goes out of the reach of our ground stations and stays there for 20 minutes. In addition, the communication window of 10 minutes roughly corresponds to the time that we can be in communication with the satellite during one revolution around Earth by the satellite.

Unlike in the setups described previously, the control of the satellite happens via a radio link. An SDR with the model *Ettus USRP B200* is connected to a PC with the CSP client software and the test automation software tools. This SDR is the one used in the actual ground station and for this testing the SDR and the PC were located in the next room in Aalto University's space laboratory. The actual ground station and all of its hardware is not used because the solar simulator has to be controlled manually and the ground station is situated several floors up from the Aalto University's Space laboratory. Nonetheless, the software and the SDR are identical to that which will be used in the ground station. Figure 25 presents this setup.

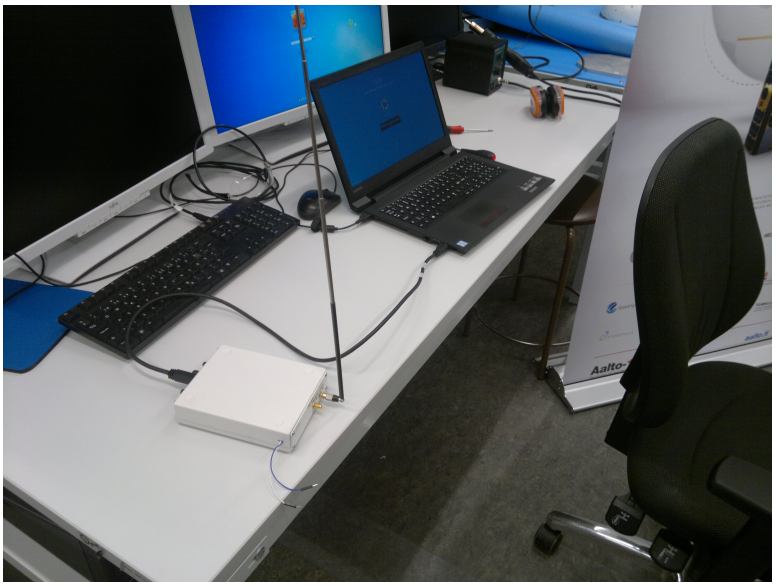

Figure 25: Day in the life setup for a "stripped down" ground station.

4 Results and Discussion

4.1 Executed tests

Different approaches in input selection were followed for the different conglomerations of tests defined in section 2.5. The collections of tests were defined as: radio payload tests, camera payload tests, satellite basic operations tests and the "Day in the life" operational mission scenario tests.

The Robot Framework test suites for the two payloads follow the more lower-level method of testing in the sense of input selection. Each of the test cases for their representative operation mode are identical in steps, but use a different combination of values in the keyword arguments. Since the payloads perform only singular functions, it was felt that using the lower-level approaches in input selection was necessary to have adequate test coverage for these payload operation modes. Yet, each test case was derived from the operation modes defined for the camera and radio payloads.

The test cases and test suites for different basic operations of the satellite are completely different from each other in design, and the tests here begin to resemble different smaller scenarios or use cases. Such cases include telemetry gathering, flight planner commands, software update, etc. In addition, a test suite was written for testing the restarts of the subsystems and of the OBC and of the whole satellite as well. The requirements and use cases for the tests were partly derived from the manuals provided by GomSpace, but mostly these tests were informal in nature as no specifications were made for these features by the Suomi100 satellite mission.

The test suites for the "Day in the life" scenarios follow the higher-level satellite system integration tests where the satellite is tested based on the operational scenarios defined for the mission. Separate test suites were written for each operational mission scenario. The test cases at this level are simply different phases in the mission scenario.

In addition to these test suites, another one was used during the development of the software for the radio payload. This test suite resembles a smoke test and it has a limited set of test cases, testing each of the operation modes. Nonetheless, no proper continous integration chain was done during this time, the smoke tests were only started manually usually once every week.

4.1.1 Camera payload

The test cases for the NanoCam were identical in structure, or in other words, several test cases were made for the same use case of the camera payload. The only difference was that the main parameters, such as gain value and exposure time, differed between test cases. These two are the main parameters according to the NanoCam manual provided by GomSpace [62]. Having the exposure time fixed, a *combinatorial* test input selection method was used to go over different combinations of camera parameters. Three different values of the exposure time were used and for each value, the same parameters other than exposure time were changed in different test cases. The values used for the exposure time were 10, 30 and 90 milliseconds. Figure 26 shows the test case structure used in the testing of the camera payload.

```
*** Test Cases ***
Imaging mode - Exposure 10000 Gain-Target 60
  [Documentation]   The onboard camera is used to take images of the
 ↪  Earth.
  [Tags]        OPMODE-IMAGING
  Satellite State   Idle
  Camera Startup    15
  Verify Startup    Camera
  Verify Device Detected    Camera  5
  Set Satellite Parameter   Camera  exposure-us   10000
  Set Satellite Parameter   Camera  gain-target   60
  Set Satellite Parameter   Camera  gain-global   2048
  Set Satellite Parameter   Camera  jpeg-qual   85
  Set Satellite Parameter   Camera  color-correct true
  Set Satellite Parameter   Camera  gamma-correct true
  Set Satellite Parameter   Camera  white-balance false
  Send Satellite Parameters
  Camera Take Picture   5000  2 def.jpg     -a
  Camera Load Picture   /mnt/data/images/def.jpg  def1.jpg
  Log   <img src="def1.jpg" width="500" height="500"> html=yes
```

Figure 26: Robot Framework test case structure for camera payload testing. The first column represents the command or action performed in CSP client (see section 3.2.2-3.3.3); the second, third and fourth columns define parameter options and values for the commands.

As can be seen from the example case, the test covers such features as NanoCam restart and detection in the satellite bus, the setting of different camera parameters and finally, the taking of images, their storage and transfer from the satellite. All images taken were then added to the Robot Framework log files, which gives a comprehensive view on how the different camera parameter values affected the images taken.

For the environment simulation, the attempt was to find a really sunny day to give some indication of the brightness of the pictures while the satellite is in orbit. This was achieved to some extent, but at the start of the test a few clouds appeared and some pictures turned out very bright and some less so. If the level of light would have stayed the same during the whole test, better baseline information could have been obtained about the affects of the different parameters on the images.

Nonetheless, no test cases failed due to software errors. These tests on the camera suggested that changing different parameters did not stop the software or that any combination of parameters caused no problems in the functioning of the satellite. It was additionally observed that the images did not become distorted in any way. Out of the 39 test cases, 9 failed because the light level was so high that the pictures became almost completely white due to brightness. Yet, the brightness in space around Earth is much higher than what it can be here on the surface even on the

brightest day [76], not to mention the fact that the tests were conducted in Finland during autumn. Thus, increasing the exposure time or gain value while in orbit would make the images to be too bright and therefore, setting the camera parameters to their default values would possibly give the best pictures. One should recall that, the automation of these tests was the first time that the developed test automation libraries along with the Robot Framework were used to properly test the Suomi100 satellite and therefore these tests worked as a technology demonstration as well. Figures 27 and 28 show some pictures taken with the NanoCam camera by the test automation tools.

Figure 27: Picture taken at Maarintie 8, Espoo, with NanoCam integrated on the Suomi100 satellite. Camera parameters were set to the default values provided by the NanoCam manual [62].

Most importantly, the tests demonstrated that the integration of the camera subsystem with the rest of the satellite was succesful. In fact, there was a defect in the integration of the camera with the satellite at the first time the satellite platform arrived to Aalto University. The camera lense was too far away from the cell in the PCB of the camera subsystem, and thus the first test pictures taken with the camera were distorted. The manufacturer of the satellite platform provided a test picture which showed that the camera was working properly, but it seems that they did not test the camera after it was integrated to the satellite. After this problem was found, GomSpace provided a new camera and the integration done by us worked properly. The Robot Framework tests worked thus also as the verification tests for the integration of the NanoCam subsystem to the satellite platform.

Figure 28: Picture taken at Maarintie 8, Espoo, with NanoCam integrated on the Suomi100 satellite. Camera parameters were set to use exposure time of 30 milliseconds and to use no gamma correction.

4.1.2 Radio payload

During the development of the software for the payload radio, a small test suite was used for smoke testing of the software and the payload. Basically, the test cases tested that the general commands are executed without errors and that the radio can output data. This test suite was also used when the payload was integrated into the satellite.

The three tests in this suite were run at least once a week during the development of the software, but a proper *Continuous Integration/Continuous Development* (CI/CD) pipeline was not used where, for example, uploading or *flashing* of a new software to the NanoMind would have caused these test to run automatically.

Besides the aforementioned smoke test, a more comprehensive set of test suites was performed for the radio payload after it was confidently integrated into the satellite platform. As the hardware and software were of designed in Aalto University and the system integration occurred with a platform manufactured by another organization, these tests took the longest time of all the automated tests performed on the Suomi100 satellite. A few sessions were held where these comprehensive test suites were run for the radio payload and each time new defects in the software and in the integration were found. Of all the tasks related to the Suomi100 satellite, the proper integration of the radio payload to the GomSpace 1U NanoEye required the most effort from the satellite team.

The test cases for the radio payload followed the lower-level testing method in a

way that all the the test suites for a given radio operation mode had the same test cases, but the frequency used was different. In other words, singular functionalities were tested with varying input values. In addition, the test suites for different radio operation modes differed as each had slightly different parameters. These test suites then covered some amount of different parameter combinations, and as with the camera payload, the measurement data was downloaded from the satellite and then processed and plotted before the figures were added to the Robot Framework HTML log files. In Figure 29 is an example of the test case structure used in testing of the radio payload.

```
*** Test Cases ***
Lowobs Mode - 5 Mhz Default parameters
  [Documentation]   The payload radio records signals on a single
 ↪  frequency.
  [Tags]        OPMODE-LOWOBS
  Satellite State   Reboot
  Radio Startup   3 0 1
  Verify Startup    Radio
  Verify Device Detected  Radio   5
  Verify Radio Status
  Store Client Responses  Lowobs Mode   80    15
  Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  2
 ↪  0;5000;100;100;100;0;0;
  Sleep     2
  Get HK      30  2  1  1  5  2  /flash/hk_test_lom
  Send Beacon   10  4  1
  Sleep     10
  Verify Radio Results  Lowobs Mode   80
  Radio Power Down
  Radio Load Data   /flash/data/m2_debug.dat  m2_debug1.dat
  Radio Plot Data   m2_debug1.dat         m2_debug1.txt  m2_debug1.png
  Log   <img src="m2_debug1.png" width="640" height="480">  html=yes
```

Figure 29: Robot Framework test case structure for radio payload testing.

The most interesting information about the payload operation was found from the CSP client replies outputted to the log files. What was measured was static and, as such, nothing too much could have been said about the test cases just by looking at the measurement plots. Beyond the fact that the values were not zero or that there was actual variance in the values measured. In Figure 30 is one plot produced from a measurement made with the radio while the satellite was at the laboratory at Aalto University and no external radio signal sources were generated for testing.

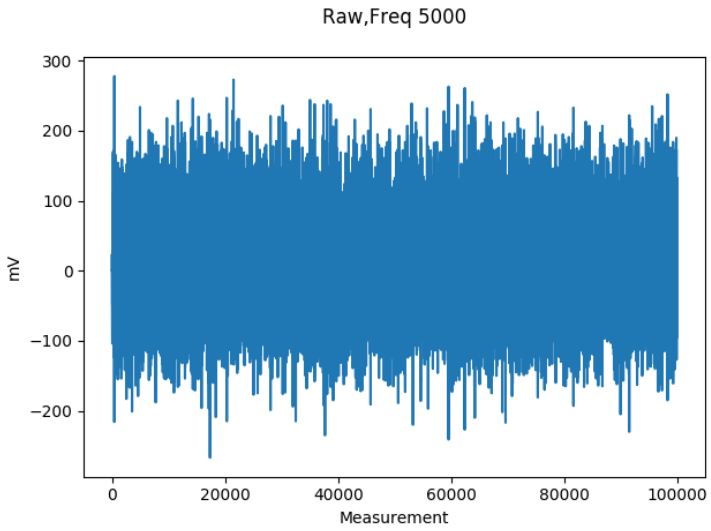

Figure 30: Plot of 100 000 measurements of radio static made by the radio payload at 5 MHz. The vertical axis gives the measured electric field in millivolts.

During the integration and testing of the radio payload, several problems occurred. The outputs of the CSP client shown in the test log files and the plots produced by the test automation tools, gave indications about problems in the operation of the payload. Therefore, a further investigation of the operation of the radio instrument was conducted. The identified problems can partly be attributed to be *emergent problems in integration*. For one, the payload was developed so that a *RaspberryPi 3* computer simulated the OBC. This computer has a Quad Core 1.2 GHz processor from *Broadcom* [77], and the processor in the NanoMind OBC in Suomi100 is a 32 MHz processor with a single processor core [59]. The data from the payload can be read and stored within 31.25 microseconds with the RaspberryPi 3 in order to have a sample rate of 32 kHz at lowest. The reading and storing with NanoMind, however, takes considerably longer, over 60 microseconds approximately.

In addition, the data needs to be read two times before obtaining a completely new measurement value. This is because the payload outputs the data from two audio channels, left and right, and data is read from one channel at a time. Therefore, with the NanoMind OBC, a sample rate of approximately 8.3 kHz can be obtained theoretically. Straightforward calculation for this sample rate can be seen in Equation 1.

$$\frac{1}{\text{data reading and storing time (s)} \times 2} = \frac{1}{60\ \mu s \times 2} \approx 8.3\ \text{kHz} \tag{1}$$

In addition, it was found that the reading and storing even at this rate is not consistent as the FreeRTOS runs other tasks while data is read from the payload. Figure 31 shows a screenshot from a *Tektronix TBS 1072B-EDU* oscilloscope reading the *slave select* pin from the payload. When the value of the slave select is low, a single reading of data from the instrument has occurred. In an attempt to counter the inconsistent reading, the priority of the FreeRTOS task assiocated with the

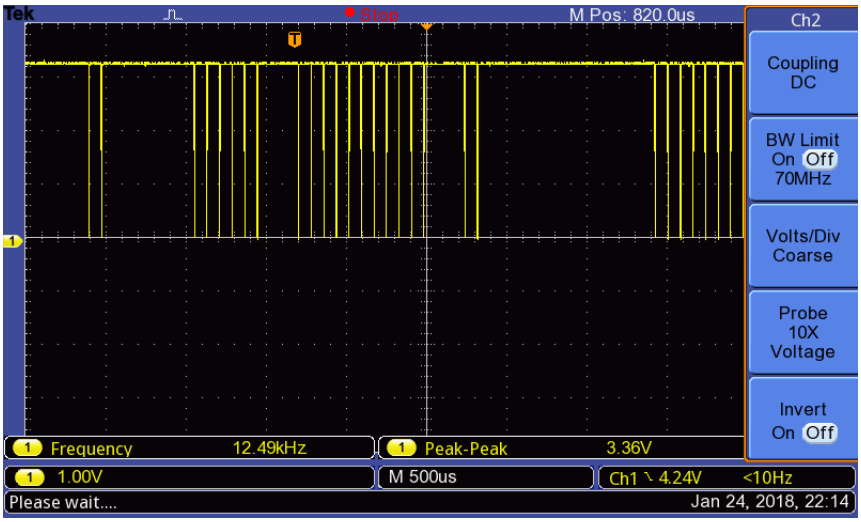

Figure 31: Screenshot from a Tektronix oscilloscope showing inconsistency in data reading rate from the radio payload instrument.

operations of the radio instrument was increased to highest among all of the tasks in NanoMind. This failed to improve the situation though. Thus, the task was made to call for a FreeRTOS *vTaskSuspendAll()* command, which would give all the processing power of the processor for the radio operation task and freeze all the other tasks in the OBC. This caused an unidentified *watchdog* task to reboot the computer, which possibly assumed that the computer was "frozen" and, as a safety feature, restarted the computer.

Therefore, as a compromise the sample rate was dropped down to 1 kHz. This was done by adding a *vTaskDelay(1)* command to the FreeRTOS task of the radio payload operation after every read and store cycle. This was the shortest time that could be added to the radio operation task, a shorter wait time would have possibly given a higher sample rate. This would additionally require modifications to be made to the general definitions of the NanoMind code, which could have caused unforeseen consequences for the operation of the satellite. Figure 32 shows that at certain periods data can be obtained more consistently. Yet, during a longer time period, there still exists longer gaps between data reading events than anticipated. Nonetheless, this data rate was felt to be good enough for our basic scientific needs. The recording of any proper "real-time" human listenable radio signal, such as speech or music, is not feasible with this current setup. With a 1 kHz sample rate the signal data does not have enough resolution for the signal to sound reasonable to human ears.

In addition to the data rate problem, some problems were discovered in the payload and the NanoMind themselves. For the command to tune the frequency in the Si4740 IC, an automatic antenna capacitance calculation was supposed to happen for a given frequency. Yet, from the Robot Framework log files, it was found that the antenna capacitance value was always 1. Thus, a separate routine had to be written to the NanoMind which calculates the capacitance value. In the consequent

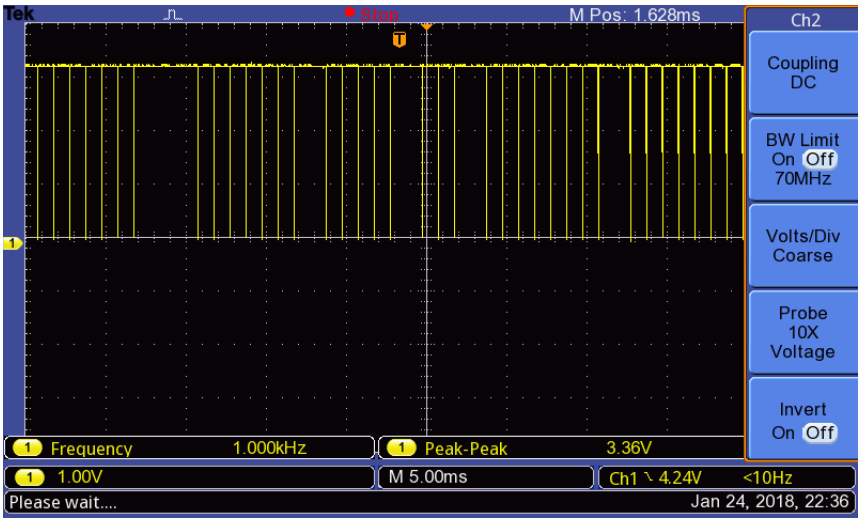

Figure 32: Screenshot from the oscilloscope showing better consistency with lower sample rate in data reading rate from the radio payload instrument.

tests run with the test automation tools after this modification, the values seemed to be correct, and could also be seen from the figures plotted from the measurement data.

Another issue which was encountered involved the GPIO pins in the NanoMind. The radio payload would have needed four pins, yet only three out of six worked. Nevertheless, changing the purpose of these three pins in relation to the radio instrument was sufficient. This in fact made it possible for data to be read from the instrument in the first place.

As a conlusion to the radio payload, the subsystem would have required its own processor and its own flash memory in order to a have higher sample rate with more consistent data reading and storing. This is the case for instance with the NanoCam subsystem, which has a 536 MHz processor and 2 Gigabytes of non-volatile memory for image storage. Nonetheless, even with a very low sample rate, valuable information from the Ionosphere can be gathered and sent back to Earth [75].

4.1.3 Satellite basic operations

Testing of the basic functionalities of the satellite software is necessary in order to obtain a reliable satellite [16, 26, 46]. Therefore, tests were performed for satellite platform features such as safe satellite restarts, different housekeeping commands, flight planner commands and flight software updating. All of these can be seen as the basic functionalities which all of the other operations of the satellite depend upon. As no requirements were defined by us for these features, this testing is, therefore, informal in nature. Nonetheless, execution of these tests was felt to be necessary.

Test suites for these features in regards to the input selection, do not follow the lower-level methods used for the two payload subsystems. Instead, test cases are based on different use cases or different situations which the satellite might face. In addition, the keywords used were less subsystem specific, and more of the test

cases used the generic keywords, such as *Send Command* and *Verify Reply Contained*.

Restart tests

Twelve test cases were written for satellite reboots during different situations. The goal of these tests was to find out if during some operation, e.g. file upload, the NanoMind can become "frozen" so that it is no longer able to restart properly. This test suite additionally has test cases only for shutting down different subsystems and to verify their absence in the satellite bus with the *ping* command. From these test cases, it was found that the NanoCom communication system could not be shutdown completely. It still replied to the ping commands even though a command was sent to shut it down. This functionality naturally is preferable and though this caused one test case of the restart test suite to fail, the test can be seen to have failed positively. As such, eleven test cases passed and one failed positively from this test suite. A few Robot Framework test cases are presented in Figure 33.

```
*** Test Cases ***
EPS reboot
  [Documentation]   Reboot satellite by rebooting EPS
  [Tags]        OPMODE-POWER
  Satellite State   Unknown
  Send Command    reboot 2
  Verify Reply Contained  Welcome to nanomind
    Wait Until Reply Contains   Mount ok

OBC reboot
  [Documentation]   Reboot nanomind OBC
  [Tags]        OPMODE-POWER
  Satellite State   Unknown
  Send Command    reboot 1
  Verify Reply Contained  Welcome to nanomind

Reboot occuring during file upload
    [Documentation]         Reboot satellite during file transfer
    [Tags]                  OPMODE-COM
    Satellite State         Reboot
    Send Command            cmp route_set 1 1000 8 1 KISS
    Send Command            fp server 1 18
    Create Flight Plan      Reboot   reboot 2  30
    Send Command            ftp server 1
    Send Command            ftp upload_file nanomind2.bin
↪   /flash/nanomind_up.bin
    Wait Until Reply Contains  Welcome to nanomind
    Wait Until Reply Contains   Mount ok
    Wait Until Reply Contains   Timeout
```

Figure 33: Robot Framework test case structure for testing satellite restarts.

The other types of test cases in the restart test suite tested system resets during satellite operations, and the restarts were mostly caused by adding reboot commands to the flight planner so that they would occur during some satellite operation, e.g. during radio payload operation. In all of these, the satellite came back safely. Yet, it has been witnessed several times by us that the NanoMind can get stuck and in those situations in order to get the OBC working again, it was necessary to manually restart the EPS or send a command via another subsystem to reboot the NanoMind. The satellite recovered safely from these situations after restart. Fortunately, when using the CSP client over the radio link, rebooting of the system has been possible. Nonetheless, this situation brought up the question whether there can happen something during orbit that causes the satellite to be "frozen" forever.

There are several watchdogs in the satellite that in theory can trigger a reboot after a certain time. Unfortunately, a situation could not have been deliberately made where the NanoMind becomes stuck. Therefore, tests for these watchdog functionalities had to be omitted from the test suite.

Housekeeping tests

Another eleven test cases were performed to test the housekeeping features provided by the GomSpace platform. Test cases were written for different housekeeping commands of different subsystems as well as for housekeeping data storing and transfer from the satellite. Testing of the beacon functionality was as well included in this test suite, as the beacon outputs recent HK information. Two test cases are presented in Figure 34.

Plotting of the beacon data was developed during the realization of these tests by another member of the satellite team, M.Sc Petri Koskimaa, and by the thesis author. These plotting functionalities were later used in the "Day in the life of the satellite" tests as well. All test cases of this test suite passed without errors. This means that, all the commands did what they were supposed to execute, which was obviously the preferred result. In Figure 35 we have an example picture of a plot of the system current provided by the beacon at different timestamps. At certain points in time there are ten datapoints due to the beacon feature used in the way that it takes multiple pulls of the system state when it is called. An average for the milliamperes is presented by the dashed red curve. The time is presented in the vertical axis in *Unix* time format. Implying the time in seconds that has elapsed since 1st of January 1970 [74]. The time is in this format because the satellite tracks time in Unix time [59].

If the presentation of the full Unix time is omitted, at the first beacon data collection at approximately 760 seconds, the satellite has been restarted by power cycling the EPS and housekeeping data collection has been turned on. Right prior to the next data point at 825 seconds the NanoCam has taken a picture and the same operation was performed right before beacon collection at subsequent points approximately at 875 and 925 seconds. In addition, there was a 20 second wait time between each operation of the camera. From the curve presenting the average of the currents collected via the beacon, it can be seen that operating the camera payload increases the overall current in the satellite system.

```
*** Test Cases ***
Download and verify housekeeping
  [Documentation]   Call EPS housekeeping routine
  [Tags]        OPMODE-POWER
  Satellite State   Reboot
  Send Command    cmp route_set 1 1000 8 1 KISS
  Send Command    ftp server 1
  Run Keyword And Ignore Error  Send Command  ftp rm /flash/hk_robot.dat
  Send Command        rparam download 1 19
  Set Satellite Parameter   Nanomind  col_en  1
  Set Satellite Parameter   Nanomind  store_en  1
  Send Satellite Parameters
  Send Command    hk get 0 1 1 0 /flash/hk_robot.dat
  Sleep         5
  Send Command    ftp server 1
  Send Command    ftp download_file /flash/hk_robot.dat hk_robot.dat
  Sleep         5
  Verify Reply Contained      1/1

Get EPS HK directly
  [Documentation]       Call EPS housekeeping routine directly
  [Tags]            OPMODE-POWER
  Satellite State       Unknown
  Send Command        cmp route_set 2 1000 8 1 I2C
  Send Command        eps hk
  Verify Reply Contained    Voltage
  Send Command        eps hksub vi
  Verify Reply Contained    Vbatt
  Verify Reply Contained    Isun
  Verify Reply Contained    Isys
```

Figure 34: Robot Framework test case structure for radio payload testing.

Flight planner tests

For testing of the flight planner feature, seven test cases were performed. The feature was tested with some basic flight planner creation commands as well as with more complicated ones. Out of all these tests involving the basic functionalities of Suomi100, this one had the most failed test cases. Firstly, it was assumed that giving the commands in an uncorrect format (string instead of an integer) would cause the CSP client to indicate an error. Such a thing did not occur, but giving the commands in the wrong format did not cause the software to crash either. In addition, if the command string was too long, an error was indicated and no flight planner command was appended to the flight plan list. This turned out to be the case with one specific command for the radio payload; but this command to run the radio payload in one of the defined operation modes happens to be the single most

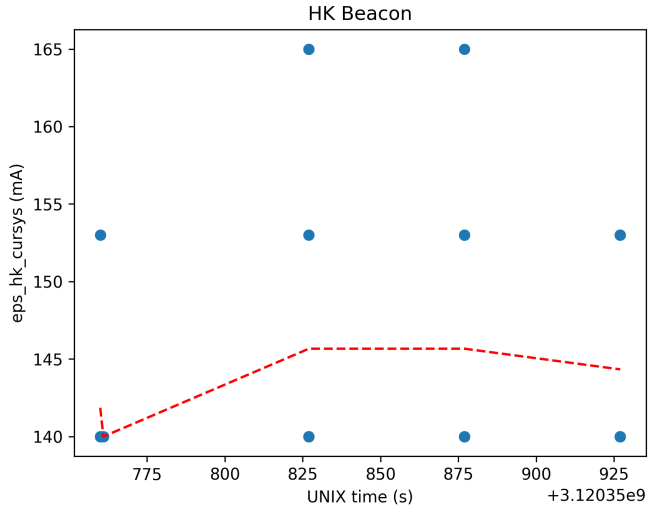

Figure 35: System current from the beacon data during imaging mode operation mode. The vertical axis shows the system current in milliamperes. Time is shown in seconds since beginning of Unix time. Average for the system current is presented by the dashed red curve and blue dots illustrate singular data points.

essential command for the payload. Therefore, in order to make it work, the source code for the flight planner was modified so that it is able to accept longer strings as commands. Otherwise, the radio payload can be used only when the satellite is within the reach of the ground station communication radios. This is not preferrable as the area of the Ionosphere that can be measured would be radically limited.

Software update tests

Three test cases were performed for the software update feature. These tests took the longest time to execute as new software had to be uploaded to the satellite in each test case. These cases tested the basic uploading of a new *software image*, restarting back to the software that was flashed to the NanoMind, and tested the uploading of an invalid file as an image to the new software. The satellite passed all these tests as expected, giving some confirmation that a new software image can actually be safely uploaded to the satellite, and the OBC can be commanded to restart with that software.

In the test case where an invalid file was uploaded as an image, the file itself was just a binary file containing measurement values from the payload radio. Restarting with this file caused the NanoMind to reset with the *EXCEPTION 13* error message. In addition, the NanoMind was set to try to boot with this file three times and it reset each time with the same error. Eventually as the boot counter reached zero, the satellite managed to recover the proper software it was flashed with. This test provided knowledge that the satellite manages to recover itself if a software image happens to be uploaded to the satellite that causes unexpected reboots.

4.1.4 "Day in the life" operational scenarios

Test suites were formed to test different phases of the scenarios that the satellite is likely to encounter while in orbit. A scenario would be, for instance, where the satellite travels in the orbit to the reach of the ground station, commands are sent to the satellite and data is downloaded. Each step in the scenario, such as downlinking of data, formed its own test case within the test suite. Four different operational scenarios were tested. These are described as:

1. scenario: The satellite comes from eclipse and is within reach to form a radio link with the ground station. Housekeeping data is gathered, the satellite takes an image and a measurement is made with the radio payload. Afterwards, the housekeeping data and the measurement are downloaded from the satellite. Finally, the satellite goes beyond the reach of the ground station.

2. scenario: The satellite comes from eclipse and is within reach to form a radio link with the ground station. Housekeeping data is gathered and the flight planner is used to set the camera to take a picture while the satellite is in eclipse and out of the reach of the ground station. After the satellite has orbited the Earth, the satellite comes again from the eclipse and is in the reach of the ground station. Housekeeping data and the taken image is downloaded from the satellite. The satellite goes again out of sight of the ground station.

3. scenario: The satellite comes from eclipse and within the reach to form a radio link with the ground station. Housekeeping data is gathered and the flight planner is used to set the camera to take pictures continuosly. The flight planner is used to gather housekeeping data continuosly as well. A sudden restart happens and the presence of all subsystems is verfified after restart. The camera is given a command to take a picture to verify the basic operation of the subsystem. In addition, verification for the charging of the batteries via the solar panels is verified. Finally, the housekeeping data is downloaded from the satellite.

4. scenario: The satellite comes from eclipse and is within reach to form a radio link with the ground station. Housekeeping data is gathered and a file is uploaded to the satellite. Finally the satellite goes out of sight of the ground station.

One test suite for each scenario was written. Different test cases in the suites were made to represent different phases in the scenarios. The majority of the keywords used in the test suites were the *Persistent Command* and *Verify Reply Contained* keywords. Persistent commanding was required for the majority of the commands because the radio link was not entirely stable. The test suites were executed several times and occasionally some keywords caused the test cases to fail due to the connection being temporarily lost. A complete test suite for the first scenario can be seen in Appendix C.

Control of the solar simulator was not automated due it being simply a lamp that

one attaches to an electric plug socket to turn it on. Thus in practice the person responsible for the testing had to manually plug and unplug the simulator from the mains current. Therefore, additional keywords were written for these tests which with a sound effect indicate that the lamp has to be turned on, or to be turned off, according to the time periods explained in section 3.3.4. These are the *Wait And Notify*, *Notify After*, *Wait Until Time event* keywords.

One important aspect with these tests was verification that the solar panels were able to charge the batteries. All the test suites had at least one test case for battery charging verification. In all such cases, the solar panels were detected and they did infact charge the batteries. As presented in section 2.1, one of the potential causes for failures with previous CubeSats has been that the solar panels were not properly connected to the power bus [16]. Thus, testing of this was felt to be crucial.

Figure 36 shows how the charge in the batteries changed over the execution of the first scenario. There are ten data points at each point in time where the beacon feature was called. The dashed red curve shows average of the recorded millivolts. The time is presented in the vertical axis in *Unix* time format. If presentation of the full Unix time is omitted, at approximately 160 seconds the first housekeeping data through the beacon has been gathered. Recently prior to this, the OBC has been restarted. Before the consequent points at approximately 220 and 225 seconds, nothing more than housekeeping collection features have been turned on.

At approximately 255 seconds the battery voltage shows an increase, which indicates that the solar panels have begun to recharge the batteries. In addition, prior to this data collection the camera was operated and the taken image was stored to the NanoCam. flash drive. An increase is again shown at the next housekeeping collection at approximately 380 seconds. Before this point in time, the ADCS has been turned off and the radio payload has ran second mode of the radio operation modes. The final collection point was taken after radio measurement data had been downloaded. It can been seen from the dashed red curve that the battery voltage shows a steady increase when the satellite was under illumination, even though the satellite operated both of its payloads during this time.

Concurrently, all the test cases for scenario 1 were executed succesfully. The housekeeping data and radio measurement were downloaded from the satellite successfully via the radio link. Similarily, all test cases for scenario 3 succeeded. A reboot was caused in the satellite and all the subsystems responded to *ping* commands after the reboot. The satellite was able to again take an image and it was verified that the solar panels were again charging the satellite. The test cases for scenario 4 were passed as well and a file was uploaded succesfully to the satellite.

Some problems were identified with the tests for scenario 2. Namely, the download speed was too slow to enable downloading of an image from the satellite during the time defined for the scenario. As presented in section 3.3.4, the time that the satellite within the sight of the ground station is approximately 10 minutes. The image taken by the camera was roughly 300 kilobytes in size and during the time it was downloaded, roughly 53 kilobytes were received.

One reason for the slow download speed was the setting for the CSP client *rdpopt* command, which controls the wait times between each packets received and other

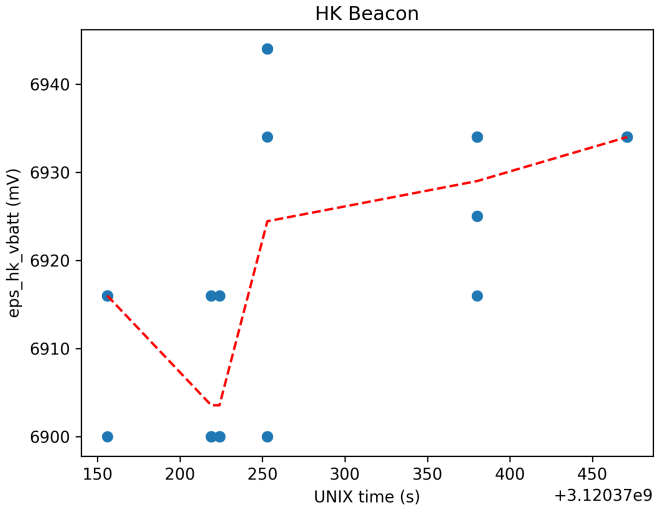

Figure 36: EPS battery charge from the beacon data during the first satellite operational scenario. The vertical axis shows the system voltage in millivolts. The red dashed curve presents average of the battery voltage and blue dots illustrate singular data points. Time is shown in Unix time in the horizontal axis.

commmunication related features. With a short wait time, the time between received packets is shorter, but the connection during downloading can be lost more frequently. With a longer wait time, the speed is lower but the connection is more stable. The theoretical maximum speed for downloading data with the bandwidth allocated for the Suomi100 mission is 0.9 kb/s. From the test suite logs produced by the Robot Framework, it could be seen that at best data could be received at the speed of 0.3 kb/s. Therefore, fine tuning of the *rdpopt* command's parameters would be needed in order to make the download faster in practice.

As a conclusion, it was verified that the solar panels did charge the batteries and the communication with the satellite over the radio link worked. All the commands sent to the satellite worked as they were supposed to according to the GomSpace manuals. Only the download speed was slow with the parameters given to the *rpopt* command in the test. Thus, a full picture could not be downloaded from the satellite. Besides this, all the test cases passed.

4.2 Release version of CubeSatAutomation test library

As was already stated in *Research purpose and goals* in Section 1, the goal was to create a reusable test automation function library. Other satellite teams working with testing of their systems could utilize this funtion library along with the Robot Framework to automize and bring a systematic nature into their testing. Basis for this library was the libraries used in testing of Suomi100 and several features were preserved while others were omitted from the final function library. Notably, all the Suomi100 specific keywords and routines were removed. In addition, some changes to the preserved keywords were made.

One essential change was the completion of the network socket based communication with the system under test. With the core library used in testing of Suomi100, only the stdin was used when replies were read from the CSP client. In the final *CubeSatAutomation* library, replies from a program can be read through the socket connection as well. In addition, several of the keywords try to choose the communication between network socket and stdin/stdout console communication. If a socket object is found to be defined in the class variables of the function library, then socket connection is used for sending commands as well as receiving responses from the SUT. If not, stdin/stdout is used for the communication, with no visible difference in the keywords when they are called in the test cases. The socket communication was tested to work with RaspberryPi 3 through an *Ethernet* cable. For this purpose a program using the source code listed in Appendix B was written to the RaspberryPi as a crude *endpoint* software.

Having two methods of communication with the system under test gives certain benefits. For example, a ground station software or a flight software in a simulated desktop environment can be tested locally by using the console communication. When using the network socket for communication with the SUT, testing can be performed on the target hardware (e.g. a subsystem or an integrated satellite) as well if the hardware has networking capabilities. A separate computer which has networking capabilities could be used for directly controlling the target hardware if the SUT doesn't provide methods for network connection.

To support testing on a remote system, new keywords for remote program startup and termination were developed. For these operations, the keywords use the *Secure Shell* (SSH) communication provided by the Python *Paramiko* library. This communication method is only used for the aforementioned actions, but a future update to the library could additionally include SSH based command sending and response receiving.

Naming of certain keywords was changed as well. The keywords related to program startup and shutdown now have a more generic *Program* as the first word. *Client* was used as the first word in the original version due to the ground station software being named CSP client. In addition, configuration files could be passed to certain keywords in the Suomi100 test automation libraries and that feature is preserved in the release version as well. The keywords related to program and socket setup accept configuration files as keyword arguments. When such a file is given, the content of the file overrule any other arguments passed to the keyword. The source code for the final CubeSat test automation library is listed in Appendix A along with example of a configuration file. In future the library could be written to be more modular with the class methods separated into different Python modules. Listing of such implementation was felt to complicate the source code presentation in the Appendix of this thesis.

However, the source code was added to a *Git* at https://github.com/Juha-MattiLukkari/cubesatautomation. New versions of the test automation library can be obtained from this website. In addition, Robot Framework examples of how the library can be utilized in testing exist in the Git.

In Robot Framework format, the keywords available in the final function library

are the following:

Program Start <program> <parameters> <config file> <wait>
Starts the program that is to be automated. Parameters to the program can be defined and both of the arguments can be read from a configuration file. Wait for *<wait>* seconds to let the program start properly.

Program Close
Closes the program and a possible socket assiocated with it.

Connect Socket <server> <port> <config file> <wait>
Opens a socket connection to a defined server and port address. These values can be read from a configuration file as well. Wait for *<wait>* seconds to let the connection be established fully.

Close Socket
Closes the opened socket connection.

Remote Program Start <program> <server> <port> <username> <password> <config file> <wait>
Starts a program on a remote server using SSH. All the arguments for this keyword can be read from a config file. Wait for *<wait>* seconds to let the program start properly.

Remote Program Close
Closes the SSH connection and terminate the opened program at the remote system.

Send Command <message> <option> <timeout> <read timeout>
Sends a message and read replies through socket connection. The reply can be stored temporarily or discarded. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Write Command <message> <option> <timeout> <read timeout>
Writes a command to the standard input and reads a reply from the standard output. The reply can be stored temporarily or discarded. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Type Command <message> <option> <timeout> <read timeout>
Types a command by automating keypresses on the keyboard and reads a reply from the standard output. The reply can stored temporarily or discarded. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Persistent Command <message> <exception replies> <end reply> <timeout> <read timeout>
Writes a command persistently to the SUT through the communication method defined earlier until a defined reply is read, a specific error reply is read or a timeout value is reached. Try to read replies for *<read timeout>* seconds between each attempt.

Verify Reply Contains <message> <timeout> <read timeout>
Reads several lines through the established communication route and tries to find for a defined message from the lines. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Verify Reply Contains Not <message> <timeout> <read timeout>
Reads several lines through the established communication route and tries not to

find for a defined message from the lines. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Verify Reply Contained <message>

Tries to find a defined message from the replies that were stored by an earlier command.

Verify Reply Contained Not <message>

Tries not to find a defined message from the replies that were stored by an earlier command.

Wait Until Reply Contains <message> <timeout> <read timeout>

Reads replies through the established communication route until a defined message is found or a timeout is reached. Read replies for *<timeout>* seconds, with *<read timeout>* seconds between each attempt to read data.

Clear Messages <option> <read timeout>

Reads messages through socket for *<read timeout>* seconds and discards them. Replies that were stored earlier can be chosen to be removed.

Clear Replies <option> <read timeout>

Flushes the standard output and reads replies for *<read timeout>* seconds and discards them. Replies that were stored earlier can be chosen to be removed.

Clear Stored Messages

Empties all the stored replies.

4.3 Improving CubeSat reliability: "Day in the life of a CubeSat" test

Currently, the CubeSat standard demands the following tests to be performed for a satellite: *Random Vibration*, *Thermal Vacuum Bakeout*, *Shock Testing* and *Visual Inspection* [9]. These are demanded only for the reason of ensuring safe integration of the P-POD deployer and the CubeSat into the launch vehicle. Usually, the specifications for these tests usually are in fact defined by the launch provider [9].

As can be seen, no testing is required for electrical or functional/operational testing of the satellite at the system integration level. In the research data represented in section 2.1, it was found that failure rates from 40 % to 20 % were prevalent in CubeSat missions [16, 14, 17]. In addition, it was suggested that these high failure rates were attributed to poor or nonexistent functional system integration testing. More so, understanding of integration and testing can be something lacking from the university-led CubeSat teams. In comparison, the CubeSat missions that were led by organisations and companies with vast experience in satellite integration and testing had considerably lower failure rates. Therefore, we strongly recommend that at least some form of guidelines for functional system integration testing should be added to the CubeSat project concept. A further detailed study for creation of such guidelines is deemed to be necessary.

In section 2.1.5, the represented research on failures with larger spacrafts highlighted the lack of proper integration and testing as a source of mission failures. In

addition, when the established testing practices used in NASA were "streamlined", consequent missions showed significant increase in failures. When comparing the data from failed CubeSat and traditional space missions, it could further be claimed that at least some guidelines for system integration testing are needed to be included in the CubeSat project.

4.3.1 Test design

"Day in the life" operational mission scenario tests are required for NASA and ESA missions [45]. Besides the mechanical tests mentioned, a test following this principle could be devised as a recommended guideline for functional CubeSat testing. This test could be known as ***"Day in the life of a CubeSat"***, following the methods described in sections 4.1.4 and 3.3.4. A test such as this would test the functionality and proper integration of the satellite. The communication with the ground station could likewise be verified. In theory, this test could decrease the amount of DOA cases for CubeSat missions. As noted in [15, 16] one of the alleged reasons for early CubeSat failures has been improper integration of the solar panels to the satellite, and thus not having enough power to form the radio link with the ground station. A test such as the one discussed here could verify these two aspects of the mission while the satellite is still on the ground.

In fact, a study done in NASA during 2017 came to similar conclusions about the "Day in the life" testing for CubeSats [78]. A recommendation was set forth for a test such as this by operating the satellite via the ground station and performing nominal operations of the satellite. In addition, using a solar illuminator to verify battery charge/discharge was recommended to be included in this test. As noted, these aspects were tested for the Suomi100 in this thesis, and recommended as the "Day in the life of a CubeSat" test in this thesis. However, this work performed with the Suomi100 was done without knowledge of the study conducted by NASA.

One technical solution for the "Day in the life of a CubeSat" test is presented in this thesis. This includes the integrated satellite, a solar simulator and the ground station. Certain basic scenarios for satellite operations were devised and tested. The mechanical stress tests for CubeSats are performed with automated machinery, likewise a method to automate the "Day in the life of a CubeSat" is presented in this thesis. This test automation includes the Robot Framework and the function libraries developed during the course of the thesis. The final version of the function library was made to be a generic testing library, which is able to automate the use of many programs running in a terminal environment. Ground station software can also be automated with this, given that such a program is terminal based. In addition, as CubeSats often lack resources and time for testing, the solution to automate the testing could help in this aspect as well. A setup for the "Day in the life of a CubeSat" test is presented in Figure 37.

4.3.2 Improved requirements and operational specifications

Testing to validate the requirements of the Suomi100 mission was informal to some degree, as the requirements were not specified in more detail. Especially the testing

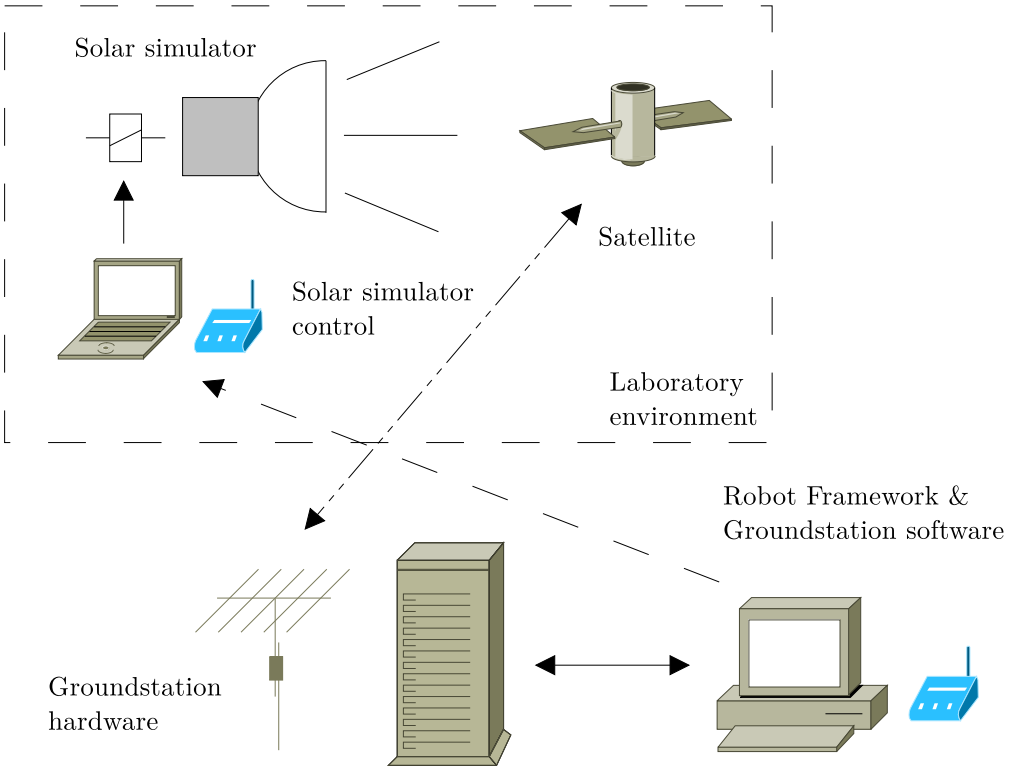

Figure 37: Presentative diagram for the "Day in the life of a CubeSat" test with automated control of ground station and solar simulator.

of the functionalities of the satellite platform was done in an informal manner. No requirements on that level were defined for the satellite by the Suomi100 satellite team. In addition, in the interest of the "Day in the life" testing, no formal documentation about satellite on-orbit operational scenarios was devised.

Therefore, the need for a more detailed requirements and operational documentation was found out as a results of the testing of Suomi100. Yet, the rigour used by the traditional space missions could be avoided. In the vein of the ideology behind CubeSats, an easier and more writable approach towards documentation should also be considered.

Defining different operation modes and assembling different operational scenarios from them could be beneficial for the "Day in the life" testing as well as for the entire satellite project. The modes and different possible scenarios could be outlined at the beginning of a satellite project and these could steer the realization of the satellite mission. Outlining of the CONOPS documents at an early phase of a spacecraft project in fact is the practise used at NASA [48].

Each operation mode could be further described with more detail: *What should the mode do? What shouldn't it do? Which commands are to be used? What happens on failure? Area of operation?* Perhaps presenting each operation mode with a *State diagram*, or with some other modeling method, could be useful. In fact, according to [37], modeling of a system is preferred if not required for mission- and safety-critical applications.

In conclusion, a design for the "Day in the life of a CubeSat" test could consist of the following steps:

1. Design different on-orbit operational scenarios based on the mission definition.
2. Break different scenarios into different operation modes of the satellite. Modes, for example, such as *downlink data*, *use payload instrument*, *stay idle* and so forth. Create state diagrams for the operation modes and derive functional requirements from the diagrams.
3. Run a scenario over a radio link and use a solar illuminator to simulate the Sun.
4. Verify battery charging and proper communication with the satellite over the radio link.
5. Based on the operation mode functional requirements, verify proper functionality of different operation modes in a scenario.
6. Perform steps 3-5 for all operational scenarios.

On the next page an example state diagram of *Imaging mode* operation mode is presented in Figure 38. The operation mode is defined in Section 2.4.2. The different states are presented as circles and the status of each subsystem in the satellite is indicated within the circle. Transitions between states are indicated by arrow lines. *Idle* state is the initial state of the system. When a critical failure is caused in the satellite by e.g. software error, the satellite goes into *System shutdown* state and the satellite power cycles the EPS and restarts the OBC and other subsystems. The satellite finally returns to the Idle state after the reset.

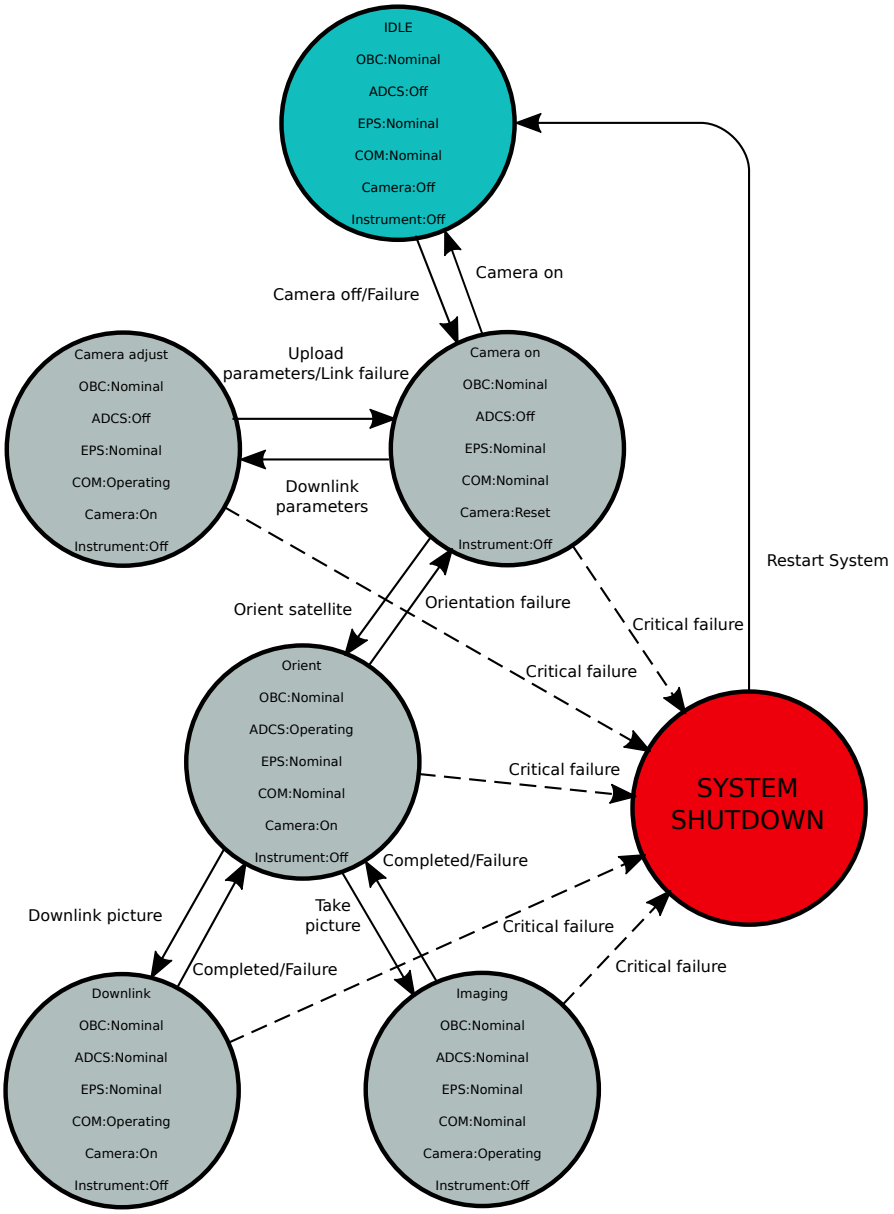

Figure 38: The *Imaging mode* operation mode as a State diagram. Circles represent states of the satellite and arrow lines indicate transitions between each state.

5 Conclusions

In this thesis automated functional system integration tests for the Suomi100 CubeSat were performed with the Robot Framework. The need for testing at this level was identified from surveys conducted for all CubeSat missions flown previously, which showed a high failure rate for missions led by university teams in contrast to low failure rates for missions performed by organisations and companies with established practices in integration and testing. Thus, testing methods used by e.g. NASA at satellite integration level along with industry proven testing practices were applied in the design of testing performed for Suomi100. The testing was automated with the Robot Framework, which is an industry proven open source test automation framework. The framework can be freely obtained through Robot Framework homepage. It was felt that doing the testing with the help of automated computer software would give the testing more rigour and reproducability. The function libraries that were used in automation of the testing were written with Python programming language. A reusable version of the test automation library, *CubeSatAutomation*, can be obtained from https://github.com/Juha-MattiLukkari/cubesatautomation.

The first tests conducted for the satellite were functional tests for the integrated Suomi100 payloads, which were an optical white light camera and an AM radio instrument for Ionospheric measurements. The second test set included testing of the core satellite functions such as housekeeping data collection, safe restart handling, software updates and so forth. The final set consisted of tests for operational mission scenarios or "Day in the life of the satellite" tests.

The thesis described the work which was done to simulate the functional environment for testing of the payloads and the operational scenario tests. For testing of the camera payload, the satellite was taken to a balcony at Aalto University on a sunny day. During the testing of the integrated radio payload, a HackRF software defined radio was used to simulate the noise signals found in the Ionosphere. A larger setup was used for the "Day in the life of the satellite" tests. With this test, not just the proper functionality of the satellite software was tested, but also that the radio link to the satellite worked and that the solar panels were able to charge the satellite. As such, the automated tests were performed via the radio link and a large Xenon lamp was used to simulate the Sun.

With the performed tests, we proved the proper functionality of the camera payload as well as the proper functionality of nearly all of the core satellite functions. The operational scenario tests showed that we can communicate with and send commands to the satellite via radio link, and that the solar panels were able to charge the batteries in the Suomi100. The only downside seemed to be the low downlink speed, yet this was to be expected.

The tests for the radio payload, on the other hand, identified several defects and most importantly emergent problems in the integration with the rest of the satellite platform. The defects were resolved with consequent software updates, but certain problems with integration could not be overcome. The biggest one being the slow speed of reading and storing of data from the payload by the OBC. This demanded a drastic reduction in the sample rate from the possible 32-48 kHz to just 1 kHz for

reliable data measurement.

Performing these tests proved to us that the Robot Framework can work as a testing framework for CubeSats, and that we could carry out automated testing of the Suomi100 CubeSat. The software libraries doing the actual automation were developed into a generic testing library, which potentially could be used for testing and automation of any local or external Linux terminal software, such as a satellite ground station software.

A solution for improving the success rate of CubeSat missions through testing was presented in this thesis as the "Day in the life of a CubeSat" test. Furthermore, features that are to be tested are proposed to be defined through state diagrams. The diagrams would represent different operation modes of the satellite and functional requirements of the system could be derived from the state diagrams. In order to decrease failure rates of CubeSat missions, these two tasks are proposed to be added as part of the CubeSat concept as guidelines for system integration testing.

The "Day in the life of a CubeSat" tests the functionalities which have from statistics been considered as causing the infant mortality of CubeSats. In the test mentioned, the ground station is used to control the satellite over the radio link, and the satellite is situated in a laboratory with a solar illuminator. By executing nominal satellite operations with this setup, the proper on-orbit functionality of the satellite can be verified. As noted, a technical solution for automating this test by automating the control of the ground station software is presented in this thesis.

References

[1] Collins, Martin, *After Sputnik: 50 Years of the Space Age*, Smithsonian Books, HarperCollins, New York, USA, 2007.

[2] Belfore, Michael, *Rocketeers: How a Visionary Band of Business Leaders, Engineers, and Pilots Is Boldly Privatizing Space*, 1st edition, Smithsonian Books, HarperCollins, New York, USA, 2007.

[3] Herrell, Linda M., *Access to Space for Technology Validation Missions: A Practical Guide*, Jet Propulsion Laboratory, 2007 IEEE Aerospace Conference, Montana, USA, 3-10 March 2007.

[4] *http://pluto.jhuapl.edu/*, accessed 8th of March 2018.

[5] Pratt, Timothy, Charles W. Bostian & Jeremy E. Allnutt, *Satellite Communications*, 2nd edition, John Wiley & Sons, New Jersey, USA, 2003.

[6] Bouwmeester J. et al., *Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology*, Delft University of Technology, Acta Astronautica, Vol 67:7-8, 2010.

[7] Poghosyan, Armen et al., *CubeSat evolution: Analyzing CubeSat capabilities for conducting science missions*, Skolkovo Institute of Science and Technology, Progress in Aerospace Sciences, Vol 88, 2017.

[8] DePasquale J. et al., *Analysis of the Earth-to-Orbit Launch Market for Nano and Microsatellites*, American Institute of Aeronautics and Astronautics, AIAA SPACE 2010 Conference & Exposition, California, USA, August 30 - September 2 2010.

[9] The CubeSat Program, *CubeSat Design Specification Rev. 13*, California Polytechnic State University, 2014.

[10] *https://tem.fi/en/spacelaw*, accessed 23rd of February 2018.

[11] *http://markets.businessinsider.com/news/stocks/iceye-successfully-launches-world-s-first-sar-microsatellite-and-establishes-finland-s-first-commercial-satellite-operations-1012996541*, accessed 23rd of February 2018.

[12] Swartwout, Michael, *CubeSats and Mission Success: 2017 Update (with a closer look at the effect of process management on outcome)*, NASA Electronic Parts and Packaging (NEPP) Program 2017 Electronics Technology Workshop, Maryland, USA, 26-29 June 2017.

[13] Doncaster, Bill & Williams Caleb & Shulman Jordan, *2017 Nano/Microsatellite Market Forecast*, SpaceWorks Enterprises, Inc. (SEI), Atlanta, USA, 2017.

[14] Swartwout, Michael, *Secondary Spacecraft in 2016: Why Some Succeed (And Too Many Do Not)*, Saint Louis University , 2016 IEEE Aerospace Conference, Montana, USA, 5-12 March 2016.

[15] Langer, Martin & Bouwmeester Jasper, *Reliability of CubeSats - Statistical Data, Developer's Beliefs and the Way Forward*, 30th Annual AIAA/USU Conference on Small Satellites, 2016.

[16] Swartwout, Michael, *The First One Hundred CubeSats: A Statistical Look*, Journal of small satellites, 2013.

[17] Swartwout, Michael, *Secondary Spacecraft in 2015: Analyzing Success and Failure*, Saint Louis University, 2015 IEEE Aerospace Conference, Montana, USA, 7-14 March 2015.

[18] *http://www.suomi100satelliitti.fi/mika*, accessed 6th of March 2018.

[19] *http://robotframework.org/*, accessed 28th of January 2018.

[20] *https://www.nasa.gov/mission_pages/station/research/benefits/cubesat*, accessed 23rd of February 2018.

[21] Straub, Jeremy et al., *OpenOrbiter: A Low-Cost, Educational Prototype CubeSat Mission Architecture*, Machines 1:1-32, 2013.

[22] Bouwmeester Jasper et al, *Survey on the implementation and reliability of CubeSat electrical bus interfaces*, CEAS Space J, Vol 9:163-173, 2017.

[23] Fortescue, Peter et al., *Spacecraft Systems Engineering*, 4th edition, John Wiley & Sons, USA, 2011.

[24] Silver, H. Ward, *The ARRL Handbook for Radio Communications 2016*, 93rd edition, The American Radio Relay League, USA, 2015.

[25] *https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database*, accessed 19th of March 2018.

[26] *http://spacenews.com/cubesat-reliability-a-growing-issue-as-industry-matures/*, accessed 12th of December, 2017.

[27] *https://www.nasa.gov/smallsat-institute*, accessed 12th of December, 2017.

[28] *http://claudelafleur.qc.ca/Scfam-failures.html*, accessed 3rd of April 2018.

[29] *https://www.space.com/13558-historic-mars-missions.html*, accessed 23rd of February 2018.

[30] Tolker-Nielsen, Toni, *EXOMARS 2016 - Schiaparelli Anomaly Inquiry*, European Space Agency, 2017.

[31] Tafazoli, Mak, *A study of on-orbit spacecraft failures*, Acta Astronautica, 64:195-205, 2009.

[32] Castet, Jean-Francois et al., *Satellite and satellite subsystems reliability: Statistical data analysis and modeling*, Georgia Institute of Technology, Reliability Engineering and System Safety, Vol 94, 2009.

[33] Tosney, William F. et al., *Satellite verification planning: Best practices and pitfalls related to testing*, Proceedings of the 5th International Symposium on Environmental Testing for Space Programmes, Netherlands, 2004.

[34] Jackelen George. et al, *When Standards and Best Practices are Ignored*, Fourth IEEE International Symposium and Forum on Software Engineering Standards, 1999.

[35] Williams, Brian C. et al., *Model-Based Programming of Intelligent Embedded Systems and Robotic Space Explorers*, Proceedings of the IEEE, Vol 91:1, IEEE, 2003.

[36] Laplante, Phillip A. & Ovaska Seppo J., *Real-Time Systems Design and Analysis*, 4th edition, IEEE Press, USA 2012.

[37] Oshana, Robert et al., *Software Engineering for Embedded Systems: Methods, Practical Techniques and Applications*, Newnes, Elsevier, Massachusetts USA, 2013.

[38] Pries, Kim H. & Quigley Jon M., *Testing Complex and Embedded Systems*, Taylor and Francis group, CRC Press, USA 2011.

[39] Myers, Glenford J., *The Art of Software Testing*, 3rd edition, John Wiley & Sons, New Jersey, USA, 2012.

[40] ISO/IEC/IEEE 29119-1:2013(E), *Software and systems engineering — Software testing — Part 1: Concepts and definitions*, IEEE 2013.

[41] Kandl, Susanne et al., *A Formal Approach to System Integration Testing*, European Dependable Computing Conference, United Kingdom, 2014.

[42] Wortman, Kristin, *Management of Independent Software Acceptance Test in the Space Domain: A Practitioner's View*, 2012 IEEE Aerospace Conference, Montana, USA, 3-10 March 2012.

[43] Tomayko, James E., *Computers in Spaceflight: The NASA experience*, National Aeronautics and Space Administration, Wichita State University, USA 1988.

[44] Badaruddin, Kareem S., *System Testbed Use on a Mature Deep Space Mission: Cassini*, 2008 IEEE Aerospace Conference, Montana, USA, 1-8 March 2008.

[45] White, Julia D., *Test Like You Fly: Assessment and Implementation Process*, Aerospace Report NO. TOR-2010(8591)-6, Space and Missile Systems Center Air Force Space Command, El Segundo California, 2010.

[46] European Cooperation For Space Standardization (ECSS) Secretariat, *Space Engineering Verification*, ESA Requirements and Standards Division, ESTEC, Netherlands, 2009.

[47] Williams, W. C., *Lessons from NASA: Design reviews, failure analysis, and redundancy are favored over predictions and life testing for spacecraft success*, IEEE Spectrum, Vol. 18:10, 1981.

[48] National Aeronautics and Space Adminstration, *NASA Systems Engineering Handbook - NASA SP-2016-6105 Rev2: Design Test Integrate Fly*, 2nd revision, CreateSpace Independent Publishing Platform, USA, 2017.

[49] , Tatsumi, Keizo, *Test Automation - Past, Present and Future*, System Test Automation Conference 2013, Tokyo, Japan, 1st of December 2013.

[50] Subramanyan, Rajesh et al., *10th International Workshop on Automation of Software Test (AST 2015)*, 37th IEEE International Conference on Software Engineering, Firenze Italy, 16 May - 24 May 2015.

[51] Kasurinen, Jussi et al., *Software Test Automation in Practice: Empirical Observations*, Lappeenranta University of Technology, Advances in Software Engineering, Vol. 2010, Hindawi Publishing Corporation, 2010.

[52] Cervantes, Alex, *Exploring the Use of a Test Automation Framework*, Jet Propulsion Laboratory, 2009 IEEE Aerospace Conference, Montana, USA, 7-14 March 2009.

[53] Pajunen, Tuomas et al., *Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework*, Tampere University of Technology, 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21-25 March 2011.

[54] Laukkanen, Pekka, *Data-Driven and Keyword-Driven Test Automation Frameworks*, Helsinki University of Technology, 2006.

[55] *http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html*, accessed 6th of March 2018.

[56] *http://www.aalto.fi/fi/current/news/2015-06-24/*, accessed 6th of March 2018.

[57] *https://gomspace.com/example-configurations.aspx*, accessed 6th of March 2018.

[58] *Small Spacecraft Technology State of the Art*, Mission Design Division Ames Research Center, NASA Center for AeroSpace Information, Maryland, USA 2016.

[59] *NanoMind A3200 datasheet*, gs-ds-nanomind-a3200-1.10, GomSpace A/S, 2016.

[60] *NanoPower P31 Datasheet*, gs-ds-nanopower-p31u-17, GomSpace A/S, 2017.

[61] *NanoCom AX100 Datasheet*, gs-ds-nanocom-ax100-3.3.docx3.3, GomSpace A/S, 2016.

[62] *NanoCam C1U Datasheet*, gs-ds-nanocam-c1u-1.5, GomSpace A/S, 2017.

[63] Mäkelä, J.S. et al., *HF radiation emitted by chaotic leader processes*, Journal of Atmospheric and Solar-Terrestrial Physics, Vol 69, 2007.

[64] Mäkelä, J.S. et al., *Using full-flash narrowband energy for ranging of lightning ground strokes*, Journal of Atmospheric and Solar-Terrestrial Physics, Vol 70, 2008.

[65] Mäkelä, Jakke S. et al., *Properties of preliminary breakdown processes in Scandinavian lightning*, Journal of Atmospheric and Solar-Terrestrial Physics, Vol 70, 2008.

[66] Mäkelä, Jakke S. et al., *Single-station narrowband ranging of active storm cells without lightning-type discrimination*, Journal of Atmospheric and Solar-Terrestrial Physics, Vol 71, 2009.

[67] *Si4740/41/42/43/44/45-C10 Automotive AM/FM Radio Receiver*, Silicon Laboratories, 2009.

[68] *Si47xx Programming Guide*, Rev. 1.0 9/14, Silicon Laboratories, 2014.

[69] Koskimaa, Petri, *Ferrite Rod Antenna in a Nanosatellite Medium and High Frequency Radio*, Aalto-University School of Electrical Engineering, 2016.

[70] *https://www.stratasysdirect.com/resources/case-studies/3d-printed-satellite-exterior-nasa-jet-propulsion-laboratory*, accessed 20th of April.

[71] *A3200 SDK*, gs-man-nanomind-a3200-sdk-v1.2, GomSpace A/S, 2017.

[72] *The FreeRTOS Reference Manual*, version 10.0.0 issue 1, Amazon Web Services, 2017.

[73] *AVR32 Tool Chain*, gs-man-avr32-toolchain-1.4, GomSpace A/S, 2016.

[74] Kerrisk, Michael *The Linux Programming Interface*, No Starch Press Inc., San Francisco, USA, 2010.

[75] Kallio, Esa et al., *Feasibility study for a nanosatellite-based instrument for in-situ measurements of radio noise*, 1st URSI Atlantic Radio Science Conference (URSI AT-RASC), Las Palmas, Spain, 16-24 May 2015.

[76] Günther, Matthias *Advanced CSP Teaching Materials: Chapter 2 Solar Radiation*, Deutsches Zentrum für Luft- und Raumfahrt.

[77] *https://www.raspberrypi.org/products/raspberry-pi-3-model-b/*, accessed 3rd of April 2018.

[78] Venturini, Catherine C. *Improving Mission Success of CubeSats*, Aerospace Report NO. TOR-2017-01689, Space and Missile Systems Center Air Force Space Command, El Segundo California, 12th of June 2017.

A CubeSatAutomation function library

The source code for the release version of the CubeSatAutomation test automation library is presented in this section. New versions of the library can be obtained from https://github.com/Juha-MattiLukkari/cubesatautomation. The library requires the following libraries to be installed: *paramiko*, *psutil* and *pyautogui* is imported at run time by the *type_ command* method.

```
1  import socket
2  import sys
3  import os
4  import signal
5  import subprocess
6  import thread
7  import time
8  from fcntl import fcntl, F_GETFL, F_SETFL
9  from os import O_NONBLOCK, read
10 from ConfigParser import SafeConfigParser
11 import psutil
12 import robot
13 import paramiko
14
15 class CubeSatAutomation(object):
16   ''' Function library for CubeSat test automation
17     Version 1.0. Written by Juha-Matti Lukkari 2017-2018.
18     Provides low level methods to automate testing of both local terminal
   ↪  based programs
19     and remote systems with networking capabilities.
20     A local program would be e.g. groundstation software, a remote system
   ↪  could e.g.
21     be a Hardware-in-the-loop, such as a satellite subsystem in a
   ↪  testbed.
22
23     stdin/stdout of a local program is used for commanding and receiving
   ↪  responses
24     from the program.
25     Socket connection is used for commanding and receiving responses from
   ↪  a remote
26     program.
27     SSH is used to start and close programs for testing in a remote
   ↪  system.
28
29   '''
30
31   ROBOT_LIBRARY_SCOPE = 'TEST_SUITE'
32   proc = None
33   server = None
34   port = 0
```

```
35    sock = None
36    ssh = None
37    reply_buffer = ""
38
39    def __init__(self):
40      self.parser = SafeConfigParser()
41
42    def connect_socket(self, server, port, config_file=None, wait_time=2):
43      '''  Connect to a network socket
44        Server is either the hostname or the IP address of the host.
45        Server and port defined in a config file override the given
      ↪  settings.
46      '''
47      if config_file:
48        print "Config file"
49        self.parser.read(str(config_file))
50        server = str(self.parser.get('SOCKET', 'server'))
51        port = int(self.parser.get('SOCKET', 'port'))
52
53      CubeSatAutomation.server = str(server)
54      CubeSatAutomation.port = int(port)
55      print "Opening socket connection.."
56      CubeSatAutomation.sock = socket.socket(socket.AF_INET,
        ↪  socket.SOCK_STREAM)
57      server_address = (CubeSatAutomation.server, CubeSatAutomation.port)
58      CubeSatAutomation.sock.connect(server_address)
59      CubeSatAutomation.sock.setblocking(0)     # For non-blocking network
        ↪  communication
60      print "Connected to %s port %s" % server_address
61      time.sleep(int(wait_time))
62
63    def close_socket(self):
64      ''' Close the network socket
65      '''
66      if CubeSatAutomation.sock:
67        print "Closing socket connection"
68        CubeSatAutomation.sock.shutdown(socket.SHUT_RDWR)
69        CubeSatAutomation.sock.close()
70        CubeSatAutomation.sock = None
71        CubeSatAutomation.server = None
72        CubeSatAutomation.port = 0
73      else:
74        print "No socket connection initialized!"
75
76    def program_start(self, prog, params=None, config_file=None,
      ↪  wait_time=5):
77      ''' Start the program for automation
```

```
78        Program and params defined in a config file override the given
    ↪   settings.
79        '''
80        if config_file:
81          self.parser.read(str(config_file))
82          prog = self.parser.get('PROGRAM', 'path')
83          params = self.parser.get('PROGRAM', 'params')
84
85        print "Opening program %s for automated control.." % str(prog)
86        CubeSatAutomation.proc = subprocess.Popen([str(prog) + " " +
           ↪   str(params)], stdin=subprocess.PIPE,
87          stdout=subprocess.PIPE, stderr=subprocess.STDOUT, shell=True)
88        flags = fcntl(CubeSatAutomation.proc.stdout, F_GETFL)      # get
           ↪   current process stdout flags
89        fcntl(CubeSatAutomation.proc.stdout, F_SETFL, flags | O_NONBLOCK)  #
           ↪   For non-blocking stdout communication
90        print "Started program " + str(prog) + " with parameters " +
           ↪   str(params)
91        time.sleep(int(wait_time))
92
93      def program_close(self):
94        ''' Close the program we were automating
95          Close any existing socket connections as well.
96          First tries to close the program in a neat way, if that fails
97          then executes the 'kill' command from terminal.
98
99          Any program that stays alive and doesn't exit after tests have
    ↪   finished
100         is a problem for the subsequent tests against the same program.
101       '''
102       if CubeSatAutomation.sock:
103         self.close_socket()
104       if CubeSatAutomation.proc:
105         CubeSatAutomation.proc.terminate()    # Doesn't close the program
           ↪   properly in some cases!
106         if os.getpgid(CubeSatAutomation.proc.pid):
107           print "Clean termination of the program wasn't successful."
108           print "Attempting to terminate from OS.."
109           pid = os.getpgid(CubeSatAutomation.proc.pid)
110           kill_command = "kill -15 " + "-" + str(pid)
111           subprocess.Popen([str(kill_command)], shell=True)
112
113         CubeSatAutomation.proc = None
114
115     def remote_program_start(self, prog, server, port=22,
116               user=None, passw=None, config_file=None, wait_time=5):
117       ''' Start a program for testing at a remote location through SSH
118         Using a config file for setup is preferred.
```

```
119         Parameters for the program are not defined separately, but
120         should be included to the prog argument.
121       '''
122       ssh = paramiko.SSHClient()
123       ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
124       if config_file:
125         self.parser.read(str(config_file))
126         prog = str(self.parser.get('REMOTE', 'prog'))
127         server = str(self.parser.get('REMOTE', 'server'))
128         port = int(self.parser.get('REMOTE', 'port'))
129         user = str(self.parser.get('REMOTE', 'username'))
130         passw = str(self.parser.get('REMOTE', 'password'))
131
132       ssh.connect(hostname=str(server), port=int(port), username=user,
          ↪  password=passw)
133       stdin, stdout, stderr = ssh.exec_command(str(prog), get_pty=True)
134       print "Started program %s on remote server %s" % (str(prog),
          ↪  str(server))
135       CubeSatAutomation.ssh = ssh
136       time.sleep(int(wait_time))
137
138     def remote_program_close(self):
139       ''' Close a remotely started program through SSH
140         Simply closes the socket and as get_pty was used, the program
      ↪  should
141         terminate on the remote system.
142       '''
143       if CubeSatAutomation.ssh:
144         CubeSatAutomation.ssh.close()
145         CubeSatAutomation.ssh = None
146
147     def _send_socket(self, message):
148       ''' Send message through socket connection
149       '''
150       print "Sending command '%s' through socket connection" % str(message)
151       command = str(message) + "\r"
152       CubeSatAutomation.sock.sendall(command)
153
154     def _send_console(self, message):
155       ''' Send message through standard input
156       '''
157       print "Sending command '%s' through standard input" % str(message)
158       command = str(message) + '\r'
159       CubeSatAutomation.proc.stdin.write(command)
160
161     def _communicate(self, message):
162       ''' Choose the communication route for sending commands
163       '''
```

```
164       if CubeSatAutomation.sock:
165         self._send_socket(str(message))
166       else:
167         self._send_console(str(message))
168
169     def _receive(self, timeout, read_timeout):
170       ''' Choose the communication route for receiving replies
171       '''
172       if CubeSatAutomation.sock:
173         console_lines = self._read_socket(int(timeout), int(read_timeout))
174       else:
175         console_lines = self._read_console(int(timeout), int(read_timeout))
176       return console_lines
177
178     def send_command(self, message, option="Store", timeout=2,
      ↪  read_timeout=2):
179       '''  Send commands to the program via the socket connection
180         Replies from the socket are read concurrently.
181       '''
182       self._send_socket(str(message))
183       console_lines = self._read_socket(int(timeout), int(read_timeout))
184       console_lines = str(console_lines).split("\\n")
185       if "Store" in str(option):
186         CubeSatAutomation.reply_buffer = console_lines
187
188     def write_command(self, message, option="Store", timeout=2,
      ↪  read_timeout=2):
189       '''  Send commands to the program via standard input
190         Replies from standard output are read concurrently.
191       '''
192       self._send_console(str(message))
193       console_lines = self._read_console(int(timeout), int(read_timeout))
194       console_lines = str(console_lines).split("\\n")
195       if "Store" in str(option):
196         CubeSatAutomation.reply_buffer = console_lines
197
198     def type_command(self, message, option="Store", timeout=2,
      ↪  read_timeout=2):
199       '''  Send commands to the program by simulating typing on a keyboard
200         Uses pyautogui library to perform the simulated typing.
201         Replies from standard output are read concurrently.
202
203         Use this keyword with caution! When using this keyword, the
    ↪  computer
204         shouldn't be used for anything else than performing testing.
205       '''
206       import pyautogui
207       pyautogui.typewrite(str(message))
```

```
208       pyautogui.press('enter')
209       console_lines = self._read_console(int(timeout), int(read_timeout))
210       console_lines = str(console_lines).split("\\n")
211       if "Store" in str(option):
212         CubeSatAutomation.reply_buffer = console_lines
213
214     def _read_socket(self, timeout=5, read_timeout=5):
215       ''' Read messages through the socket
216       '''
217       print "Reading messages from socket connection"
218       socket_lines = []
219       time_count = 0
220       while time_count < int(timeout):
221         time.sleep(1)           # Wait for data to be 'cooked'
222         time_count = time_count + 1
223         try:
224           line = CubeSatAutomation.sock.recv(1024)
225         except socket.error:      # No data to be read, wait if more comes
226           socket_lines.append("Waiting for more data from socket..\n")
227           read_timecount = 0
228           while read_timecount < int(read_timeout):
229             try:
230               line = CubeSatAutomation.sock.recv(1024)
231             except socket.error:
232               time.sleep(1)
233               read_timecount = read_timecount + 1
234               time_count = time_count + 1
235               continue
236             else:
237               break
238           if read_timecount >= int(read_timeout):
239             socket_lines.append("Process data read timeout!\n")
240             break
241         print "sock:" + line.rstrip()
242         if line != '':
243           socket_lines.append(line)
244       return socket_lines
245
246     def _read_console(self, timeout, read_timeout=10):
247       '''  Read messages through standard output
248       '''
249       console_lines = []
250       time_count = 0
251       while time_count < int(timeout):
252         time.sleep(1)           # Wait for data to be 'cooked'
253         time_count = time_count + 1
254         try:
255           line = read(CubeSatAutomation.proc.stdout.fileno(), 1024)
```

```
256         except OSError:           # No data to be read, wait if more comes
257           console_lines.append("Waiting for more data from process..\n")
258           read_timecount = 0
259           while read_timecount < int(read_timeout):
260             try:
261               line = read(CubeSatAutomation.proc.stdout.fileno(), 1024)
262             except OSError:
263               time.sleep(1)
264               read_timecount = read_timecount + 1
265               time_count = time_count + 1
266               continue
267             else:
268               break
269           if read_timecount >= int(read_timeout):
270             console_lines.append("Process data read timeout!\n")
271             break
272         print "term:" + line.rstrip()
273         if line != '':
274           console_lines.append(line)
275       return console_lines
276
277     def clear_messages(self, option="Stored", read_timeout=5):
278       ''' Empty messages that have come through socket connection
279       '''
280       if "Stored" in str(option):
281         CubeSatAutomation.reply_buffer = ""
282       try:
283         CubeSatAutomation.sock.recv(1024)
284       except socket.error:          # No data to be read, wait if more comes
285         read_timecount = 0
286         while read_timecount < int(read_timeout):
287           try:
288             CubeSatAutomation.sock.recv(1024)
289           except socket.error:
290             time.sleep(1)
291             read_timecount = read_timecount + 1
292             continue
293           else:
294             break
295
296     def clear_replies(self, option="Stored", read_timeout=5):
297       ''' Clear process replies
298         Flush the stdout and read & discard messages during read_timeout.
299         Additionally, empty the reply_buffer
300       '''
301       CubeSatAutomation.proc.stdout.flush()
302       if "Stored" in str(option):
303         CubeSatAutomation.reply_buffer = ""
```

```
304       try:
305         read(CubeSatAutomation.proc.stdout.fileno(), 1024)
306       except OSError:             # No data to be read, wait if more comes
307         read_timecount = 0
308         while read_timecount < int(read_timeout):
309           try:
310             read(CubeSatAutomation.proc.stdout.fileno(), 1024)
311           except OSError:
312             time.sleep(1)
313             read_timecount = read_timecount + 1
314             continue
315           else:
316             break
317
318     def clear_stored_messages(self):
319       '''  Empty the reply_buffer of messages received from the program
320       '''
321       CubeSatAutomation.reply_buffer = ""
322
323     def verify_reply_contains(self, message, timeout=5, read_timeout=10):
324       ''' Read messages from the standard output/socket
325         Verify that the specified message is received.
326       '''
327       console_lines = self._receive(int(timeout), int(read_timeout))
328       console_lines = str(console_lines).split("\\n")
329       found = False
330       for line in console_lines:
331         if str(message) in line:
332           found = True
333           break
334       if not found:
335         print console_lines
336         raise ValueError ("Message %s was not found in the process
          ↪  replies!\n" % str(message))
337
338     def verify_reply_contains_not(self, message, timeout=5,
       ↪  read_timeout=10):
339       ''' Read messages from the standard output/socket
340         Verify that the specified message isn't received.
341       '''
342       console_lines = self._receive(int(timeout), int(read_timeout))
343       console_lines = str(console_lines).split("\\n")
344       found = False
345       for line in console_lines:
346         if str(message) in line:
347           found = True
348           break
349       if found:
```

```
350         print console_lines
351         raise ValueError ("Message %s was not supposed to be found in the
             ↪  process replies!\n" % str(message))
352 
353     def verify_reply_contained(self, message):
354       '''  Verify if a specified message is contained in the reply_buffer
           ↪  class variable
355       '''
356       console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
357       found = False
358       for line in console_lines:
359         if str(message) in line:
360           found = True
361           break
362       if not found:
363         print console_lines
364         raise ValueError ("Message %s was not found in the recent process
             ↪  replies!\n" % str(message))
365 
366     def verify_reply_contained_not(self, message):
367       '''  Verify that a specified message is not contained in the
           ↪  reply_buffer class variable
368       '''
369       console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
370       found = False
371       for line in console_lines:
372         if str(message) in line:
373           found = True
374           break
375       if found:
376         print console_lines
377         raise ValueError ("Message %s was not supposed to be found in the
             ↪  recent process replies!\n" % str(message))
378 
379     def wait_until_reply_contains(self, message, timeout=20,
         ↪  read_timeout=5):
380       '''  Wait until a specified reply is received
381       '''
382       completed = False
383       found = False
384       time_count = 0
385       for line in CubeSatAutomation.reply_buffer:
386         if str(message) in str(line):
387           completed = True
388           found = True
389       while not completed:
390         console_lines = self._receive(1, int(read_timeout))
391         console_lines = str(console_lines).split("\\n")
```

```
392         time_count = time_count + 1
393         time.sleep(1)
394         if time_count > int(timeout):
395           completed = True
396         for line in console_lines:
397           if str(message) in line:
398             found = True
399             completed = True
400             break
401       if not found:
402         print console_lines
403         raise ValueError ("Message %s was not found in the process
            ↪  replies!\n" % str(message))
404
405     def persistent_command(self, message, exception_replies,
406                 end_reply="None", timeout=5, read_timeout=2):
407       ''' Sends a command persistently until either time runs out or a
          ↪  certain reply is received
408         The exception replies are such replies which are considered to be
        ↪  failures in the
409         commanding of the system. When such a reply is encountered, the
        ↪  command is re-sent.
410         An end reply indicates a reply which tells us to stop sending
        ↪  resending the command.
411         '''
412       time_count = 0
413       completed = False
414       found = False
415       error_found = False
416       command = str(message) + '\r'
417       self._communicate(str(message))
418       exception_replies = str(exception_replies)
419       exception_replies = exception_replies.split(';')
420       while not completed:
421         if time_count >= int(timeout):
422           completed = True
423           break
424         console_lines = self._receive(int(timeout), int(read_timeout))
425         console_lines = str(console_lines).split("\\n")
426         CubeSatAutomation.reply_buffer = console_lines
427         for line in console_lines:
428           for exception_reply in exception_replies:
429             if str(exception_reply) in str(line):
430               print "Exception %s found, retrying to send command" %
                  ↪  str(exception_reply)
431               self._communicate(str(message))
432               break
433           if str(end_reply) in str(line):
```

```
434             completed = True
435             found = True
436             break
437           if "None" in str(end_reply):
438             completed = True
439             found = True
440             break
441         time_count = time_count + 1
442         time.sleep(1)
443       if len(end_reply) > 0:
444         if found:
445           print "Desired reply %s was found in process replies" %
              ↪  str(end_reply)
446         else:
447           if str(end_reply) == "Timeout":
448             pass
449           else:
450             raise ValueError ("Desired reply %s was not found in process
                ↪  replies" % str(end_reply))
451       if "None" in str(end_reply) in str(end_reply):
452         console_lines = self._receive(1, int(read_timeout))
453         console_lines = str(console_lines).split("\\n")
454         CubeSatAutomation.reply_buffer = console_lines
455         for line in console_lines:
456             if str(exception_reply) in str(line):
457               raise ValueError ("Exception %s still found after timeout" %
                  ↪  str(exception_reply))
```

A configuration file which can be used with certain keywords in the CubeSatAutomation library is presented below. **Note:** The format of the configuration files should follow the one presented here.

```
1  [PROGRAM]
2  path:       /home/juha/S100/EGSE/EGSE/csp-client-v1.1/build/csp-client
3  params:     -a 8 -d /dev/ttyUSB0 -b 500000
4  [SOCKET]
5  server:     s100-juha
6  port:       5000
7  [REMOTE]
8  prog:       /home/pi/SUT/Socket/satprog
9  server:     169.254.57.130
10 port:       22
11 username:   pi
12 password:   raspberry
```

B API for CSP client

The C language code for socket connection API appended to the GomSpace CSP client is presented below.

```
#ifdef linux
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#endif

void *api_server(void)
{
    // Here listen for connections
    // and use command_run to execute the commands
    int sockfd, newsockfd;
    char buffer[256];
    struct sockaddr_in serv_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
            printf("ERROR opening socket\n");
    else
        printf("Socket started\n");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    int portno = atoi("5000");
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
             sizeof(serv_addr)) < 0)
             printf("ERROR on binding\n");
    listen(sockfd, 5);
    newsockfd = accept(sockfd,
                (struct sockaddr *) NULL,
                NULL);
     if (newsockfd < 0)
        printf("ERROR on accept\n");
    while(1)
    {
        bzero(buffer, 256);
        int n = read(newsockfd, buffer, 255);  //Read messages coming
         ↪  through socket
        if(n>0)
        {
            printf("SOCKET: Received %d bytes:%s\n", n, buffer);
            command_run(buffer);
```

```
43              write(newsockfd, reply_buffer, strlen(reply_buffer));
44              //dup2(old, newsockfd);
45          }
46      }
47      close(newsockfd);
48      printf("Closed socket");
49      close(sockfd);
50  }
51
52  int main(int argc, char * argv[]) {
53
54      /* API */
55      static pthread_t handle_api;
56      pthread_create(&handle_api, NULL, api_server, NULL);
57
58      /* Wait here for console to end */
59      pthread_join(handle_api, NULL);
60      return 0;
61  }
```

C Robot Framework test suites

Some of the Robot Framework test suites that were executed in the testing of Suomi100 CubeSat are presented here. In total more than 20 test suites were executed. Given that some test suites are identical in structure to others, only the most essential test suites are listed here.

First the Robot Framework script file common to all test suites is presented. This file contains some additional keywords such as the *Satellite State* keyword.

```
#--------------s100_keywords.robot-------------

*** Keywords ***
Start Suite
  Client Start
  Sleep     5
  Connect Socket

End Suite
  Send Message    exit_client
  Close Connection
  Client Close

Satellite State
  [Arguments] ${state}
  ${state}= Convert To Lowercase  ${state}
  Run Keyword If  '${state}' == 'idle'
  ... Idle
  Run Keyword If  '${state}' == 'reboot'
  ... Reboot

Idle
  Send Message    fp delete telem
  Send Message    fp delete beacon
  Send Message    radio opmode_thread_terminate
  Verify Startup    Satellite

Reboot
  Send Message    fp delete telem
  Send Message    fp delete beacon
  Send Message    radio opmode_thread_terminate
  Send Message    reboot 2
  Sleep       30
  Verify Startup    Satellite
```

Next are certain test cases for the functional test of the camera presented.

```
#--------------camera_tests.robot-------------

*** Settings ***
```

```
4  Library   String
5  Library   ../CubeSatAutomation.py
6  Library   ../NanoCam.py
7  Resource  ../s100_keywords.robot
8  Suite Setup   Start Suite
9  Suite Teardown  Client Close
10
11 *** Test Cases ***
12
13 Imaging mode - Default parameters
14   [Documentation]   The onboard camera is used to take images of the
   ↪  earth.
15   [Tags]        OPMODE-IMAGING
16   Satellite State   Reboot
17   Camera Startup    15
18   Verify Startup    Camera
19   Verify Device Detected    Camera  5
20   Set Satellite Parameter   Camera  exposure-us   10000
21   Set Satellite Parameter   Camera  gain-target   30
22   Set Satellite Parameter   Camera  gain-global   2048
23   Set Satellite Parameter   Camera  jpeg-qual   85
24   Set Satellite Parameter   Camera  color-correct true
25   Set Satellite Parameter   Camera  gamma-correct true
26   Set Satellite Parameter   Camera  white-balance false
27   Send Satellite Parameters
28   Camera Take Picture   5000  2 def.jpg   -a
29   Camera Load Picture   /mnt/data/images/def.jpg  def.jpg
30   Log   <img src="def.jpg" width="500" height="500">  html=yes
31
32 Imaging mode - Exposure 10000 Gain-Target 60
33   [Documentation]   The onboard camera is used to take images of the
   ↪  earth.
34   [Tags]        OPMODE-IMAGING
35   Satellite State   Idle
36   Camera Startup    15
37   Verify Startup    Camera
38   Verify Device Detected    Camera  5
39   Set Satellite Parameter   Camera  exposure-us   10000
40   Set Satellite Parameter   Camera  gain-target   60
41   Set Satellite Parameter   Camera  gain-global   2048
42   Set Satellite Parameter   Camera  jpeg-qual   85
43   Set Satellite Parameter   Camera  color-correct true
44   Set Satellite Parameter   Camera  gamma-correct true
45   Set Satellite Parameter   Camera  white-balance false
46   Send Satellite Parameters
47   Camera Take Picture   5000  2 def.jpg     -a
48   Camera Load Picture   /mnt/data/images/def.jpg  def1.jpg
49   Log   <img src="def1.jpg" width="500" height="500"> html=yes
```

```
50
51  Imaging mode - Exposure 10000 Gain-Target 90
52    [Documentation]    The onboard camera is used to take images of the
    ↪  earth.
53    [Tags]         OPMODE-IMAGING
54    Satellite State    Idle
55    Camera Startup     15
56    Verify Startup     Camera
57    Verify Device Detected     Camera  5
58    Set Satellite Parameter    Camera  exposure-us    10000
59    Set Satellite Parameter    Camera  gain-target    90
60    Set Satellite Parameter    Camera  gain-global    2048
61    Set Satellite Parameter    Camera  jpeg-qual    85
62    Set Satellite Parameter    Camera  color-correct true
63    Set Satellite Parameter    Camera  gamma-correct true
64    Set Satellite Parameter    Camera  white-balance false
65    Send Satellite Parameters
66    Camera Take Picture    5000  2 def.jpg     -a
67    Camera Load Picture    /mnt/data/images/def.jpg  def2.jpg
68    Log    <img src="def2.jpg" width="500" height="500"> html=yes
69
70  Imaging mode - Exposure 10000, Jpeg quality 20
71    [Documentation]    The onboard camera is used to take images of the
    ↪  earth.
72    [Tags]         OPMODE-IMAGING
73    Satellite State    Idle
74    Camera Startup     15
75    Verify Startup     Camera
76    Verify Device Detected     Camera  5
77    Set Satellite Parameter    Camera  exposure-us    10000
78    Set Satellite Parameter    Camera  gain-target    20
79    Set Satellite Parameter    Camera  gain-global    2048
80    Set Satellite Parameter    Camera  jpeg-qual    20
81    Set Satellite Parameter    Camera  color-correct true
82    Set Satellite Parameter    Camera  gamma-correct true
83    Set Satellite Parameter    Camera  white-balance false
84    Send Satellite Parameters
85    Camera Take Picture    5000  2 def.jpg
86    Camera Load Picture    /mnt/data/images/def.jpg     def5.jpg
87    Log    <img src="def5.jpg" width="500" height="500"> html=yes
88
89  Imaging mode - Exposure 10000, Gamma correct false
90    [Documentation]    The onboard camera is used to take images of the
    ↪  earth.
91    [Tags]         OPMODE-IMAGING
92    Satellite State    Idle
93    Camera Startup     15
94    Verify Startup     Camera
```

```
95    Verify Device Detected    Camera  5
96    Set Satellite Parameter   Camera  exposure-us   10000
97    Set Satellite Parameter   Camera  gain-target   20
98    Set Satellite Parameter   Camera  gain-global   2048
99    Set Satellite Parameter   Camera  jpeg-qual   100
100   Set Satellite Parameter   Camera  color-correct true
101   Set Satellite Parameter   Camera  gamma-correct false
102   Set Satellite Parameter   Camera  white-balance false
103   Send Satellite Parameters
104   Camera Take Picture   5000  2 def.jpg
105   Camera Load Picture   /mnt/data/images/def.jpg  def10.jpg
106   Log   <img src="def10.jpg" width="500" height="500">  html=yes
107
108 Imaging mode - Exposure 30000, Default parameters
109   [Documentation]   The onboard camera is used to take images of the
      ↪  earth.
110   [Tags]        OPMODE-IMAGING
111   Satellite State   Idle
112   Camera Startup    15
113   Verify Startup    Camera
114   Verify Device Detected    Camera  5
115   Set Satellite Parameter   Camera  exposure-us   30000
116   Set Satellite Parameter   Camera  gain-target   30
117   Set Satellite Parameter   Camera  gain-global   2048
118   Set Satellite Parameter   Camera  jpeg-qual   85
119   Set Satellite Parameter   Camera  color-correct true
120   Set Satellite Parameter   Camera  gamma-correct true
121   Set Satellite Parameter   Camera  white-balance false
122   Send Satellite Parameters
123   Camera Take Picture   5000  2 def.jpg     -a
124   Camera Load Picture   /mnt/data/images/def.jpg  def13.jpg
125   Log   <img src="def13.jpg" width="500" height="500">  html=yes
126
127 Imaging mode - Exposure 30000 Gain-Target 60
128   [Documentation]   The onboard camera is used to take images of the
      ↪  earth.
129   [Tags]        OPMODE-IMAGING
130   Satellite State   Idle
131   Camera Startup    15
132   Verify Startup    Camera
133   Verify Device Detected    Camera  5
134   Set Satellite Parameter   Camera  exposure-us   30000
135   Set Satellite Parameter   Camera  gain-target   60
136   Set Satellite Parameter   Camera  gain-global   2048
137   Set Satellite Parameter   Camera  jpeg-qual   85
138   Set Satellite Parameter   Camera  color-correct true
139   Set Satellite Parameter   Camera  gamma-correct true
140   Set Satellite Parameter   Camera  white-balance false
```

```
141     Send Satellite Parameters
142     Camera Take Picture    5000  2 def.jpg      -a
143     Camera Load Picture    /mnt/data/images/def.jpg  def14.jpg
144     Log    <img src="def14.jpg" width="500" height="500">  html=yes
```

The following listing presents certain test cases from different test suites written for testing of the radio payload.

```
#--------------payload_tests_rawmode_5.robot-------------

*** Settings ***
Library   String
Library   ../libraries/CubeSatAutomation.py
Library   ../libraries/RadioPayload.py
Resource  ../resources/s100_keywords.robot
Suite Setup   Start Suite
Suite Teardown  Client Close

*** Test Cases ***

Raw Mode - 5 Mhz Default parameters
  [Documentation]   The payload radio performs several sweeps over the entire frequency range.
  [Tags]        OPMODE-RAW
  Satellite State   Reboot
  Radio Startup   3 0 1
  Verify Startup    Radio
  Verify Device Detected  Radio   5
  Verify Radio Status
  Store Client Responses  Raw Mode    200   15
  Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  1  0;5000;100;100000;
  Sleep     2
  Get HK      30  2  1  1  5  2  /flash/hk_test_lom
  Send Beacon   10  4  1
  Sleep     10
  Verify Radio Results  Raw Mode    200
  Radio Power Down
  Radio Load Data   /flash/data/m1_debug.dat  m1_debug1.dat
  Radio Plot Data   m1_debug1.dat         m1_debug1.txt  m1_debug1.png
  Log   <img src="m1_debug1.png" width="640" height="480">  html=yes

Raw Mode - 5 Mhz 1000000 times
  [Documentation]   The payload radio performs several sweeps over the entire frequency range.
  [Tags]        OPMODE-RAW
  Satellite State   Reboot
  Radio Startup   3 0 1
  Verify Startup    Radio
  Verify Device Detected  Radio   5
  Verify Radio Status
  Store Client Responses  Raw Mode    820   25
  Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  1  0;5000;100;1000000;
```

```
43    Sleep      2
44    Get HK       30   2   1   1   5   2   /flash/hk_test_lom
45    Send Beacon    10   4   1
46    Sleep      10
47    Verify Radio Results   Raw Mode   820
48    Radio Power Down
49    Sleep      30
50    Radio Load Data    /flash/data/m1_debug.dat   m1_debug7.dat      260
51    Radio Plot Data    m1_debug7.dat    m1_debug7.txt m1_debug7.png
52    Log    <img src="m1_debug7.png" width="640" height="480">   html=yes
53
54  #--------------payload_tests_lowobsmode_5.robot-------------
55
56  *** Settings ***
57  Library    String
58  Library    ../libraries/CubeSatAutomation.py
59  Library    ../libraries/RadioPayload.py
60  Resource   ../resources/s100_keywords.robot
61  Suite Setup    Start Suite
62  Suite Teardown   Client Close
63
64  *** Test Cases ***
65
66  Lowobs Mode - 5 Mhz Default parameters, output_type 3
67    [Documentation]    The payload radio performs several sweeps over the
    ↪  entire frequency range.
68    [Tags]         OPMODE-LOWOBS
69    Satellite State    Idle
70    Radio Startup    3 0 1
71    Verify Startup     Radio
72    Verify Device Detected   Radio    5
73    Verify Radio Status
74    Store Client Responses   Lowobs Mode    80     15
75    Run Radio Mode       /flash/radio_params.cfg   /flash/radio_props.cfg   2
    ↪  0;5000;100;100;100;3;0;
76    Sleep      2
77    Get HK       30   2   1   1   5   2   /flash/hk_test_lom
78    Send Beacon    10   4   1
79    Sleep      10
80    Verify Radio Results   Lowobs Mode 80
81    Radio Power Down
82    Radio Load Data    /flash/data/m2_debug.dat   m2_debug5.dat
83    Radio Plot Data    m2_debug5.dat    m2_debug5.txt    m2_debug5.png
84    Log    <img src="m2_debug5.png" width="640" height="480">   html=yes
85
86  Lowobs Mode - 5 Mhz 100 times, 10 points in average
87    [Documentation]    The payload radio performs several sweeps over the
    ↪  entire frequency range.
```

```
88    [Tags]        OPMODE-LOWOBS
89    Satellite State   Idle
90    Radio Startup   3 0 1
91    Verify Startup    Radio
92    Verify Device Detected  Radio   5
93    Verify Radio Status
94    Store Client Responses  Lowobs Mode   80    15
95    Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  2
      ↪  0;5000;100;10;100;3;0;
96    Sleep     2
97    Get HK      30  2  1  1  5  2  /flash/hk_test_lom
98    Send Beacon   10  4  1
99    Sleep     10
100   Verify Radio Results  Lowobs Mode 80
101   Radio Power Down
102   Radio Load Data   /flash/data/m2_debug.dat  m2_debug8.dat
103   Radio Plot Data   m2_debug8.dat   m2_debug8.txt m2_debug8.png
104   Log   <img src="m2_debug8.png" width="640" height="480">  html=yes
105
106 #--------------payload_tests_targetmode.robot-------------
107
108 *** Settings ***
109 Library   String
110 Library   ../libraries/CubeSatAutomation.py
111 Library   ../libraries/RadioPayload.py
112 Resource  ../resources/s100_keywords.robot
113 Suite Setup   Start Suite
114 Suite Teardown  Client Close
115
116 *** Test Cases ***
117
118 Target Mode - Other antenna
119   [Documentation]   The payload radio performs several sweeps over the
      ↪  entire frequency range.
120   [Tags]        OPMODE-TARGET
121   Satellite State   Idle
122   Radio Startup   3 0 0
123   Verify Startup    Radio
124   Verify Device Detected  Radio   5
125   Verify Radio Status
126   Store Client Responses  Target Mode   1420    15
127   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  3
      ↪  0;1000;10000;10;10;1000;100;0;0;
128   Sleep     2
129   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
130   Send Beacon   10  4  1
131   Sleep     10
132   Verify Radio Results  Target Mode   1420
```

```
133   Sleep         20
134   Radio Power Down
135   Radio Load Data   /flash/data/m3_debug.dat  m3_debug4.dat
136   Radio Plot Data   m3_debug4.dat         m3_debug4.txt  m3_debug4.png
137   Log   <img src="m3_debug4.png" width="640" height="480">  html=yes
138
139 Target Mode - N_ave 1000
140   [Documentation]   The payload radio performs several sweeps over the
      ↪  entire frequency range.
141   [Tags]        OPMODE-TARGET
142   Satellite State   Idle
143   Radio Startup   3 0 1
144   Verify Startup    Radio
145   Verify Device Detected  Radio   5
146   Verify Radio Status
147   Store Client Responses  Target Mode   1920    15
148   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  3
      ↪  0;1000;10000;10;10;1000;1000;3;0;
149   Sleep     2
150   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
151   Send Beacon   10  4  1
152   Sleep     10
153   Verify Radio Results  Target Mode   1920
154   Sleep         20
155   Radio Power Down
156   Radio Load Data   /flash/data/m3_debug.dat  m3_debug6.dat    260
157   Radio Plot Data   m3_debug6.dat         m3_debug6.txt  m3_debug6.png
158   Log   <img src="m3_debug6.png" width="640" height="480">  html=yes
```

Certain test cases for NanoEye core features are presented next.

```
#--------------flight_planner_tests.robot-------------

*** Settings ***
Library   String
Library   ../libraries/CubeSatAutomation.py
Library   ../libraries/RadioPayload.py
Library   ../libraries/NanoCam.py
Resource  ../resources/s100_keywords.robot
Suite Setup   Start Suite
Suite Teardown  Client Close

*** Test Cases ***

Simple Flight Planner
  [Documentation]   Create flight planner command
  [Tags]        OPMODE-COM
  Satellite State   Reboot
  Create Flight Plan  Ping1   ping 1  10
  Verify Reply Message  Reply in  10  10

Invalid Flight Planner
  [Documentation]   Create invalid flight planner command
  [Tags]        OPMODE-COM
  Satellite State   Idle
  Run Keyword And Ignore Error  Send Command  fp delete Ping1
  Create Flight Plan    Ping1   ping 1  abc   def
  Verify Reply Contained  error

Delete Flight Planner
  [Documentation]   Delete flight planner command
  [Tags]        OPMODE-COM
  Satellite State   Idle
  Run Keyword And Ignore Error  Send Command  fp delete Ping1
  Create Flight Plan    Ping1   ping 1  10
  Send Command      fp delete Ping1
  Send Command      fp list
  Verify Reply Contained Not  Ping1

Create Larger Flight Planner
  [Documentation]   Create flight planner with beacon and a payload
 ↪  command
  [Tags]        OPMODE-COM
  Satellite State   Idle
  Run Keyword And Ignore Error  Send Command  fp delete Ping1
  Run Keyword And Ignore Error  Send Command  hk server 1
  Send Command      hk server 1 21
```

```
46    Create Flight Plan    Beacon    hk get 0 1 1 0  5 5
47    Create Flight Plan    Picture   cam snap -a   30
48    Sleep         30
49    Verify Reply Message  All
50
51  #--------------hk_tests.robot-------------
52
53  *** Settings ***
54  Library   String
55  Library   ../libraries/CubeSatAutomation.py
56  Library   ../libraries/RadioPayload.py
57  Library   ../libraries/NanoCam.py
58  Resource  ../resources/s100_keywords.robot
59  Suite Setup   Start Suite
60  Suite Teardown  Client Close
61
62  *** Test Cases ***
63
64  Download and verify housekeeping
65    [Documentation]   Call EPS housekeeping routine
66    [Tags]        OPMODE-POWER
67    Satellite State   Reboot
68    Send Command    cmp route_set 1 1000 8 1 KISS
69    Send Command    ftp server 1
70    Run Keyword And Ignore Error  Send Command  ftp rm /flash/hk_robot.dat
71    Send Command        rparam download 1 19
72    Set Satellite Parameter   Nanomind  col_en  1
73    Set Satellite Parameter   Nanomind  store_en  1
74    Send Satellite Parameters
75    Send Command    hk get 0 1 1 0 /flash/hk_robot.dat
76    Sleep         5
77    Send Command    ftp server 1
78    Send Command    ftp download_file /flash/hk_robot.dat hk_robot.dat
79    Sleep         5
80    Verify Reply Contained      1/1
81
82  Get NanoComm HK
83    [Documentation]   Call AX100 housekeeping routine
84    [Tags]        OPMODE-POWER
85    Satellite State   Unknown
86    Send Command    cmp route_set 5 1000 8 1 CAN
87    Send Command      ax100 hk
88    Verify Reply Contained  last_contact
89    Verify Reply Contained  tot_tx_count
90    Verify Reply Contained  tot_rx_count
91    Verify Reply Contained  temp_brd
92
93  Get Telemetries
```

```
94    [Documentation]   Get Telemetries for subsystems
95    [Tags]        OPMODE-POWER
96      Satellite State     Reboot
97      Send Command        rparam download 1 18
98      Set Satellite Parameter     Nanomind    col_obc     10
99      Set Satellite Parameter     Nanomind    col_eps     10
100     Set Satellite Parameter     Nanomind    col_com     10
101     Set Satellite Parameter     Nanomind    col_cam     10
102     Set Satellite Parameter     Nanomind    bcn_interval     10 10 10
103     Send Satellite Parameters
104     Send Command                rparam download 1 19
105     Set Satellite Parameter     Nanomind    col_en  1
106     Set Satellite Parameter     Nanomind    store_en    1
107     Send Satellite Parameters
108     Sleep           30
109   Send Command  ftp server 1
110   Send Command  ftp download_file /flash/hk/tbl-021.bin tbl-021.bin
111     Wait Until Reply Contains   100.0%
112     Send Command    ftp download_file /flash/hk/tbl-022.bin tbl-022.bin
113     Wait Until Reply Contains   100.0%
114     Send Command    ftp download_file /flash/hk/tbl-025.bin tbl-025.bin
115     Wait Until Reply Contains   100.0%
116     Send Command    ftp download_file /flash/hk/tbl-026.bin tbl-026.bin
117     Wait Until Reply Contains   100.0%
118   Parse HK       tbl-021.bin
119     Parse HK       tbl-022.bin
120     Parse HK       tbl-025.bin
121     Parse HK       tbl-026.bin
122
123 #--------------reboot_tests.robot-------------
124
125 *** Settings ***
126 Library   String
127 Library   ../libraries/CubeSatAutomation.py
128 Library   ../libraries/RadioPayload.py
129 Library   ../libraries/NanoCam.py
130 Resource  ../resources/s100_keywords.robot
131 Suite Setup   Start Suite
132 Suite Teardown  Client Close
133
134 *** Test Cases ***
135
136 EPS reboot
137   [Documentation]   Reboot satellite by rebooting EPS
138   [Tags]        OPMODE-POWER
139   Satellite State   Unknown
140   Send Command    reboot 2
141   Verify Reply Contained  Welcome to nanomind
```

```
142     Wait Until Reply Contains   Mount ok
143
144 OBC reboot
145   [Documentation]   Reboot nanomind OBC
146   [Tags]        OPMODE-POWER
147   Satellite State   Unknown
148   Send Command    reboot 1
149   Verify Reply Contained  Welcome to nanomind
150
151 Shutdown systems and verify their absence
152   [Documentation]     Shutdown subsystems
153   [Tags]          OPMODE-POWER
154   Satellite State     Reboot
155   Send Command      cmp route_set 6 1000 8 1 CAN
156   Verify Device Detected  Camera  10
157   Send Command      shutdown 6
158   Run Keyword And Ignore Error  Send Command      ping 6
159   Verify Reply Contained  Timeout after
160   Send Command      cmp route_set 5 1000 8 1 CAN
161   Verify Device Detected  Comm  10
162   Send Command      shutdown 5
163   Run Keyword And Ignore Error  Send Command      ping 5
164   Verify Reply Contained  Timeout after
165   Send Command      reboot 2
166   Wait Until Reply Contains   Mount ok
167
168 Reboot occuring during radio payload operation
169   [Documentation]   Reboot EPS during payload measurement
170   [Tags]        OPMODE-LOWOBS
171   Satellite State   Unknown
172   Radio Startup   3 0 1
173   Verify Startup    Radio
174   Verify Device Detected  Radio   5
175   Verify Radio Status
176   Run Radio Mode    /flash/radio_params.cfg  /flash/radio_props.cfg  2
    ↪  0;5000;100;100;100;0;0;
177   Send Command    reboot 2
178   Verify Reply Contained  Welcome to nanomind
179     Wait Until Reply Contains   Mount ok
180   Verify Device Detected  Radio   5
181
182 Reboot occuring during file download
183   [Documentation]     Reboot satellite during file transfer
184   [Tags]          OPMODE-COM
185   Satellite State     Reboot
186     Send Command            cmp route_set 1 1000 8 1 KISS
187     Send Command            fp server 1 18
188     Create Flight Plan      Reboot   reboot 2  30
```

```
189   Send Command      ftp server 1
190   Send Command      ftp download_file /flash/nanomind.bin
      ↪  nanomind_down.bin
191   Wait Until Reply Contains   Welcome to nanomind
192     Wait Until Reply Contains   Mount ok
193     Wait Until Reply Contains   Timeout
194
195 #--------------softupdate_tests.robot-------------
196
197 *** Settings ***
198 Library   String
199 Library   ../libraries/CubeSatAutomation.py
200 Library   ../libraries/RadioPayload.py
201 Library   ../libraries/NanoCam.py
202 Resource  ../resources/s100_keywords.robot
203 Suite Setup   Start Suite
204 Suite Teardown  Client Close
205
206 *** Test Cases ***
207
208 Upload new software and reboot using it
209   [Documentation]   Upload new software to Nanomind
210   [Tags]        OPMODE-SOFTUPDATE
211   Satellite State   Reboot
212   Send Command    cmp route_set 1 1000 8 1 KISS
213   Send Command    ftp server 1
214   Run Keyword And Ignore Error  Send Command  ftp rm /flash/nanomind2.bin
215   Sleep         5
216   Send Command    ftp upload_file nanomind2.bin /flash/nanomind2.bin
217   Wait Until Reply Contains   100.0%    45
218   Send Command    rparam download 1 0
219   Set Satellite Parameter   Nanomind  swload_image
      ↪  \"/flash/nanomind2.bin"\
220   Set Satellite Parameter   Nanomind  swload_count  10
221   Send Satellite Parameters
222   Send Command    reboot 1
223   Wait Until Reply Contains   Ram image   45  20
224   Sleep         80
225   Send Command    radio on 0 1
226   Verify Reply Contained    radio reply size 1
227
228 Upload invalid software image and see that nanomind returns to the default
      ↪  software
229   [Documentation]   Upload invalid file for software update
230   [Tags]        OPMODE-SOFTUPDATE
231   Satellite State   Reboot
232   Send Command    cmp route_set 1 1000 8 1 KISS
233   Send Command    ftp server 1
```

```
234   Run Keyword And Ignore Error  Send Command    ftp rm /flash/m1_debug.bin
235   Sleep         5
236   Send Command    ftp upload_file m1_debug1.dat /flash/m1_debug.bin
237   Wait Until Reply Contains   100.0%    45
238   Send Command        rparam download 1 0
239   Set Satellite Parameter   Nanomind  swload_image
      ↪  \"/flash/m1_debug.bin"\
240   Set Satellite Parameter   Nanomind  swload_count  3
241   Send Satellite Parameters
242   Send Command        reboot 1
243   Wait Until Reply Contains   Booting image in 10 seconds   45  30
244   Verify Reply Message    EXCEPTION
245   Wait Until Reply Contains   1 times left  180   30
246   Wait Until Reply Contains   Welcome to nanomind   45    30
247   Clear Replies         All
248   Send Command        ping 1
249   Verify Reply Contained    Reply in
```

Here is presented one test suite for the "Day in the life of a satellite".

```
#--------------dayinthelife1_mod.robot-------------

*** Settings ***
Library   String
Library   ../libraries/CubeSatAutomation.py
Library   ../libraries/RadioPayload.py
Library   ../libraries/NanoCam.py
Resource  ../resources/s100_keywords.robot
Suite Setup   Client Start   None
 ↪  /home/petri/s100/EGSE/csp-client-v1.1/build/csp-client  -a 10 -z
 ↪  localhost
Suite Teardown  Client Close   None


*** Test Cases ***

Come from eclipse - Verify charging
    [Documentation]     Day in the life operations
    [Tags]              OPMODE-POWER
    Wait and Notify     Coming from eclipse     5   /resources/notify.wav
    Notify After        Going to eclipse        600
 ↪  /resources/notify2.wav
    Satellite State     Unknown
    Clear Replies       All
    Persistent Command  reboot 1   error
    Sleep               10
    Persistent Command  rdpopt 5 30000 16000 1 2000 3     error   Setting
    Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat
 ↪  error
    Persistent Command  gssbcsp addr 6     error
    Persistent Command  gssbcsp interstage sensors  error   Panel
    Verify Reply Contained Not  Coarse Sunsensor: 0
    Persistent Command          gssbcsp addr 7     error
    Persistent Command          gssbcsp interstage sensors  error   Panel
    Verify Reply Contained Not  Coarse Sunsensor: 0
    Persistent Command          eps hk     error   Voltage
    Persistent Command          eps hksub vi    error   Vbatt
    Verify Reply Contained      Vbatt
    Verify Reply Contained      Isun
    Verify Reply Contained      Isys
    Verify Reply Contained Not  boost[1] 0mV
    Verify Reply Contained Not  boost[2] 0mV
    Persistent Command      ftp server 1       error
    Run Keyword And Ignore error    Persistent Command    ftp rm
 ↪  /flash/hk_robot.dat    error   No such file
    Persistent Command      rparam download 1 19    error   Wrote
```

```
42      Persistent Command      rparam set col_en 1     error   Result
43      Persistent Command      rparam set store_en 1   error   Result
44      Persistent Command      rparam send     error   REP
45      Persistent Command      rparam download 1 18    error  Wrote
46      Persistent Command      rparam set bcn_interval 10 10 10    error
    ↪   Result
47      Persistent Command      rparam send     error   REP
48      Verify Reply Contained Not  error
49      Persistent Command      hk get 0 10 10 0 /flash/hk_robot.dat    error
50
51  Come from eclipse - Take image
52      [Documentation]     Day in the life operations
53      [Tags]              OPMODE-IMAGING
54      Satellite State     Communicating
55      Clear Replies       All
56      Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat   error
57      Persistent Command  rparam download 1 19    error   Wrote
58      Persistent Command  rparam set col_en 0     error   Result
59      Persistent Command  rparam set store_en 0   error   Result
60      Persistent Command  rparam send             error   REP
61      Persistent Command  rparam download 1 18    error   Wrote
62      Persistent Command  rparam set bcn_interval 0 0 0   error   Result
63      Persistent Command  rparam send             error   REP
64      Persistent Command  adcs server 1 20        error
65      Persistent Command  adcs ephem tle new      error
66      Persistent Command  adcs run start          error
67      Persistent Command  adcs set nadir          error
68      Persistent Command  cmp route_set 6 1000 8 1 CAN    error   Success
69      Persistent Command  cam snap -a   Snap error        All
70      Persistent Command  cam store test.jpg      error   Result
71      Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat  error
72
73  Come from eclipse - Record radio signals
74      [Documentation]     Day in the life operations
75      [Tags]              OPMODE-LOWOBS
76      Satellite State     Communicating
77      Clear Replies       All
78      Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat  error
79      Persistent Command  adcs run fullstop       error
80      Persistent Command  rparam download 1 19    error   Wrote
81      Persistent Command  rparam set col_en 1     error   Result
82      Persistent Command  rparam set store_en 1   error   Result
83      Persistent Command  rparam send             error   REP
84      Persistent Command  rparam download 1 18    error   Wrote
85      Persistent Command  rparam set bcn_interval 10 10 10    error   Result
86      Persistent Command  rparam send             error   REP
87      Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat    error
```

```
88      Write Command       radio operation /flash/radio_params.cfg
    ↪   /flash/radio_props.cfg 2 0;0;0;0;0;0;0;
89      Sleep               120
90      Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat  error
91
92  Come from eclipse - Downlink data
93      [Documentation]     Day in the life operations
94      [Tags]              OPMODE-COM
95      Satellite State     Communicating
96      Clear Replies       All
97      Persistent Command  rdpopt 6 30000 16000 1 2000 3   error   Setting
98      Persistent Command  rparam download 1 19    error   Wrote
99      Persistent Command  rparam set col_en 0     error   Result
100     Persistent Command  rparam set store_en 0   error   Result
101     Persistent Command  rparam send             error   REP
102     Persistent Command  rparam download 1 18    error   Wrote
103     Persistent Command  rparam set bcn_interval 0 0 0   error   Result
104     Persistent Command  rparam send             error   REP
105     Persistent Command  ftp server 1            error
106     Persistent Command  ftp download_file /flash/data/m2_debug.dat
    ↪   m2_debug.dat     error   100.0%      45    10
107     Persistent Command  hk get 0 10 10 0 /flash/hk_robot.dat    error
108     Persistent Command  ftp server 1            error
109     Persistent Command  ftp download_file /flash/hk_robot.dat hk_robot.dat
    ↪   error   100.0%  45  10
110     Wait Until Time Event   Going to eclipse    600
111     Parse HK            hk_robot.dat  None      True    hk_plot1.png
    ↪   timestamps  eps_hk_vbatt
112     Parse HK            hk_robot.dat  None      True    hk_plot2.png
    ↪   timestamps  eps_hk_cursys
113     Log                 <img src="hk_plot1.png" width="500" height="500">
    ↪   html=yes
114     Log                 <img src="hk_plot2.png" width="500" height="500">
    ↪   html=yes
```

D Robot Framework Test Results

The log files for the majority of the executed test suites are presented in this section. Keywords of the test cases are not shown, only the titles and Pass/Fail status of the test cases are presented. For demonstration, below in Figure D is an example of how the test result logs appear when the test cases and the keywords are expanded in the HTML file. Responses from the CSP client are visible under the two keywords expanded in the example.

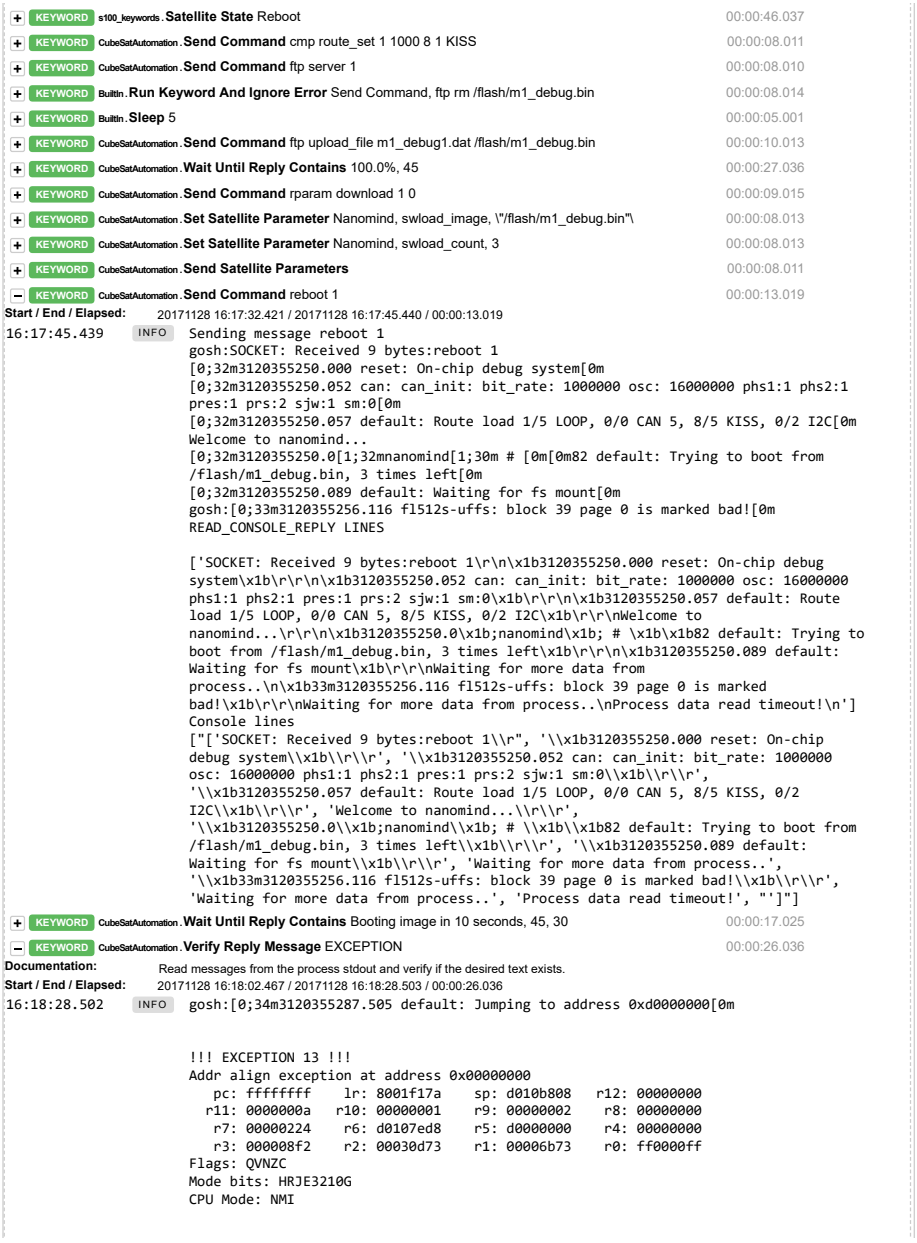

Figure D: Expanded Robot Framework test log file.

Camera Tests Test Log

Generated
20171013 17:57:46 GMT+03:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	39	30	9	02:15:28	
All Tests	39	30	9	02:15:28	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-IMAGING	39	30	9	02:15:28	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Camera Tests	39	30	9	02:15:33	

Test Execution Log

SUITE **Camera Tests** 02:15:33.095

Full Name: Camera Tests
Source: /home/juha/soft/pypractice/CSAdev/camera_tests.robot
Start / End / Elapsed: 20171013 15:42:13.522 / 20171013 17:57:46.617 / 02:15:33.095
Status: 39 critical test, 30 passed, 9 failed
39 test total, 30 passed, 9 failed

SETUP s100_keywords . **Start Suite** 00:00:05.012
TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.004
TEST **Imaging mode - Default parameters** 00:03:35.311
TEST **Imaging mode - Exposure 10000 Gain-Target 60** 00:03:02.276
TEST **Imaging mode - Exposure 10000 Gain-Target 90** 00:02:59.273
TEST **Imaging mode - Exposure 10000 Gain-Target 120** 00:02:24.241
TEST **Imaging mode - Exposure 10000 Gain-Target 150** 00:02:18.221
TEST **Imaging mode - Exposure 10000, Jpeg quality 20** 00:02:47.265
TEST **Imaging mode - Exposure 10000, Jpeg quality 60** 00:02:47.261
TEST **Imaging mode - Exposure 10000, Jpeg quality 80** 00:02:52.277
TEST **Imaging mode - Exposure 10000, Jpeg quality 100** 00:04:04.392
TEST **Imaging mode - Exposure 10000, Color correct false** 00:03:51.372
TEST **Imaging mode - Exposure 10000, Gamma correct false** 00:04:25.422
TEST **Imaging mode - Exposure 10000, White balance true** 00:04:18.417
TEST **Imaging mode - Exposure 10000, Color correct & gamma correct false** 00:05:17.503
TEST **Imaging mode - Exposure 30000, Default parameters** 00:02:52.282
TEST **Imaging mode - Exposure 30000 Gain-Target 60** 00:03:17.315
TEST **Imaging mode - Exposure 30000 Gain-Target 90** 00:02:58.285
TEST **Imaging mode - Exposure 30000 Gain-Target 120** 00:02:18.225
TEST **Imaging mode - Exposure 30000 Gain-Target 150** 00:02:32.231
TEST **Imaging mode - Exposure 30000, Jpeg quality 20** 00:02:31.246

TEST Imaging mode - Exposure 30000, Jpeg quality 60	00:03:01.296
TEST Imaging mode - Exposure 30000, Jpeg quality 80	00:02:46.278
TEST Imaging mode - Exposure 30000, Jpeg quality 100	00:04:28.453
TEST Imaging mode - Exposure 30000, Color correct false	00:04:11.406
TEST Imaging mode - Exposure 30000, Gamma correct false	00:05:04.467
TEST Imaging mode - Exposure 30000, White balance true	00:04:29.427
TEST Imaging mode - Exposure 30000, Color correct & gamma correct false	00:05:13.472
TEST Imaging mode - Exposure 90000, Default parameters	00:03:20.334
TEST Imaging mode - Exposure 90000 Gain-Target 60	00:03:19.321
TEST Imaging mode - Exposure 90000 Gain-Target 90	00:02:56.294
TEST Imaging mode - Exposure 90000 Gain-Target 120	00:02:29.245
TEST Imaging mode - Exposure 90000 Gain-Target 150	00:02:20.233
TEST Imaging mode - Exposure 90000, Jpeg quality 20	00:02:22.235
TEST Imaging mode - Exposure 90000, Jpeg quality 60	00:02:20.231
TEST Imaging mode - Exposure 90000, Jpeg quality 80	00:02:27.237
TEST Imaging mode - Exposure 90000, Jpeg quality 100	00:03:32.336
TEST Imaging mode - Exposure 90000, Color correct false	00:03:57.416
TEST Imaging mode - Exposure 90000, Gamma correct false	00:05:04.473
TEST Imaging mode - Exposure 90000, White balance true	00:05:23.523
TEST Imaging mode - Exposure 90000, Color correct & gamma correct false	00:05:27.526

Payload Tests Rawmode 5 Test Log

Generated
20180102 13:37:03 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:42:16	
All Tests	4	4	0	00:42:16	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-RAW	4	4	0	00:42:16	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Payload Tests Rawmode 5	4	4	0	00:42:21	

Test Execution Log

SUITE **Payload Tests Rawmode 5** 00:42:21.473

Full Name: Payload Tests Rawmode 5
Source: /home/juha/soft/pypractice/CSAdev/suites/payload_tests_rawmode_5.robot
Start / End / Elapsed: 20180102 12:54:41.751 / 20180102 13:37:03.224 / 00:42:21.473
Status: 4 critical test, 4 passed, 0 failed
4 test total, 4 passed, 0 failed

SETUP s100_keywords . **Start Suite** 00:00:05.011

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.011

TEST **Raw Mode - 5 Mhz Default parameters** 00:04:10.208

TEST **Raw Mode - 5 Mhz Zeros as parameters** 00:04:07.947

TEST **Raw Mode - 5 Mhz Default parameters, sample rate 48k** 00:04:13.567

TEST **Raw Mode - 5 Mhz 1000000 times** 00:29:44.471

Payload Tests Lowobsmode 5 Test Log

Generated
20180102 15:22:42 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	9	9	0	00:32:11	
All Tests	9	9	0	00:32:11	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-LOWOBS	9	9	0	00:32:11	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Payload Tests Lowobsmode 5	9	9	0	00:32:16	

Test Execution Log

SUITE **Payload Tests Lowobsmode 5** 00:32:16.318

Full Name: Payload Tests Lowobsmode 5
Source: /home/juha/soft/pypractice/CSAdev/suites/payload_tests_lowobsmode_5.robot
Start / End / Elapsed: 20180102 14:50:26.482 / 20180102 15:22:42.800 / 00:32:16.318
Status: 9 critical test, 9 passed, 0 failed
9 test total, 9 passed, 0 failed

SETUP s100_keywords . **Start Suite** 00:00:05.011

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.008

TEST **Lowobs Mode - 5 Mhz Default parameters** 00:04:04.547

TEST **Lowobs Mode - 5 Mhz Zeros as parameters** 00:03:06.495

TEST **Lowobs Mode - 5 Mhz Default parameters, output_type 1** 00:03:02.048

TEST **Lowobs Mode - 5 Mhz Default parameters, output_type 2** 00:03:02.564

TEST **Lowobs Mode - 5 Mhz Default parameters, output_type 3** 00:03:00.627

TEST **Lowobs Mode - 5 Mhz Default parameters, sample rate 48k** 00:03:01.630

TEST **Lowobs Mode - 5 Mhz 10000 times** 00:06:18.653

TEST **Lowobs Mode - 5 Mhz 100 times, 10 points in average** 00:03:00.719

TEST **Lowobs Mode - 5 Mhz 100 times, 1000 points in average** 00:03:33.747

Payload Tests Targetmode Test Log

Generated
20180103 12:48:31 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	6	6	0	02:10:32	
All Tests	6	6	0	02:10:32	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-TARGET	6	6	0	02:10:32	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Payload Tests Targetmode	6	6	0	02:10:38	

Test Execution Log

SUITE **Payload Tests Targetmode** 02:10:37.556

Full Name: Payload Tests Targetmode
Source: /home/juha/soft/pypractice/CSAdev/suites/payload_tests_targetmode.robot
Start / End / Elapsed: 20180103 10:37:53.369 / 20180103 12:48:30.925 / 02:10:37.556
Status: 6 critical test, 6 passed, 0 failed
6 test total, 6 passed, 0 failed

SETUP s100_keywords . **Start Suite** 00:00:05.010

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.007

TEST **Target Mode - Default parameters** 00:22:25.392

TEST **Target Mode - Zeros as parameters** 00:20:18.144

TEST **Target Mode - output_type 3** 00:21:21.401

TEST **Target Mode - Other antenna** 00:21:55.296

TEST **Target Mode - N_ave 1000** 00:22:38.615

TEST **Target Mode - Default parameters, sample rate 48k** 00:21:53.307

Flight Planner Tests Test Log

Generated
20171127 11:23:37 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	7	4	3	00:09:00	
All Tests	7	4	3	00:09:00	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-COMM	5	4	1	00:06:07	
OPMODE-RAW	2	0	2	00:02:52	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Flight Planner Tests	7	4	3	00:09:05	

Test Execution Log

SUITE **Flight Planner Tests** 00:09:05.025

Full Name: Flight Planner Tests
Source: /home/juha/soft/pypractice/CSAdev/suites/flight_planner_tests.robot
Start / End / Elapsed: 20171127 11:14:31.973 / 20171127 11:23:36.998 / 00:09:05.025
Status: 7 critical test, 4 passed, 3 failed
7 test total, 4 passed, 3 failed

SETUP s100_keywords . **Start Suite** 00:00:05.006

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.004

TEST **Simple Flight Planner** 00:01:09.055

TEST **Invalid Flight Planner** 00:00:33.052

TEST **Delete Flight Planner** 00:00:49.089

TEST **Simple Repeating Flight Planner** 00:01:29.094

TEST **Create Larger Flight Planner** 00:02:07.172

TEST **Create Flight Planner for Low Observation Mode** 00:01:22.140

TEST **Create Flight Planner for Low Observation and Imaging Modes** 00:01:30.158

Hk Tests Test Log

Generated
20171218 11:10:43 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	11	11	0	00:36:21	
All Tests	11	11	0	00:36:21	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-IMAGING	1	1	0	00:05:51	
OPMODE-LOWOBS	1	1	0	00:07:30	
OPMODE-POWER	9	9	0	00:23:00	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Hk Tests	11	11	0	00:36:26	

Test Execution Log

SUITE **Hk Tests** 00:36:26.392

Full Name: Hk Tests
Source: /home/juha/soft/pypractice/CSAdev/suites/hk_tests.robot
Start / End / Elapsed: 20171218 10:34:16.608 / 20171218 11:10:43.000 / 00:36:26.392
Status: 11 critical test, 11 passed, 0 failed
11 test total, 11 passed, 0 failed

SETUP s100_keywords . **Start Suite** 00:00:05.012

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.009

TEST **Download and verify housekeeping** 00:02:21.185

TEST **Get EPS HK directly** 00:00:25.060

TEST **Get NanoComm HK** 00:00:19.046

TEST **Set HK collection and reboot satellite** 00:02:58.244

TEST **Reload previous HK collection parameters** 00:03:55.340

TEST **Get HK type 1** 00:02:11.188

TEST **Get HK type 2** 00:02:20.194

TEST **Get Telemetries** 00:03:47.339

TEST **Collect and plot HK data when NanoCam and radio payload are on and off** 00:04:42.676

TEST **Collect an plot HK data during camera operation** 00:05:50.847

TEST **Collect and plot HK data during radio payload operation** 00:07:29.988

Reboot Tests Test Log

Generated
20171218 16:01:46 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	12	11	1	00:16:20	
All Tests	12	11	1	00:16:20	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-COMM	5	5	0	00:06:40	
OPMODE-IDLE	1	1	0	00:01:53	
OPMODE-IMAGING	1	1	0	00:02:24	
OPMODE-LOWOBS	1	1	0	00:01:50	
OPMODE-POWER	4	3	1	00:03:33	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Reboot Tests	12	11	1	00:16:26	

Test Execution Log

SUITE **Reboot Tests** 00:16:25.747

Full Name: Reboot Tests
Source: /home/juha/soft/pypractice/CSAdev/suites/reboot_tests.robot
Start / End / Elapsed: 20171218 15:45:20.817 / 20171218 16:01:46.564 / 00:16:25.747
Status: 12 critical test, 11 passed, **1 failed**
12 test total, 11 passed, **1 failed**

SETUP s100_keywords . **Start Suite** 00:00:05.011
TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.005

TEST **EPS reboot** 00:00:29.050
TEST **OBC reboot** 00:00:13.032
TEST **Camera payload reboot** 00:00:51.080
TEST **Comm system reboot** 00:00:47.073
TEST **Shutdown systems and verify their absence** 00:02:00.174
TEST **Shutdown all systems except EPS and reboot system** 00:01:53.169
TEST **Reboot occuring during radio payload operation** 00:01:50.167
TEST **Reboot occuring during picture download** 00:02:24.198
TEST **Reboot occuring during file download** 00:02:04.176
TEST **Reboot occuring during file upload** 00:02:08.179
TEST **Reboot occuring during param list update** 00:01:01.101
TEST **Reboot occuring during flight planner creation** 00:00:39.065

Softupdate Tests Test Log

Generated
20171128 16:19:40 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	3	3	0	00:12:33	
All Tests	3	3	0	00:12:33	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-SOFTUPDATE	3	3	0	00:12:33	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Softupdate Tests	3	3	0	00:12:38	

Test Execution Log

SUITE **Softupdate Tests** 00:12:38.042

Full Name: Softupdate Tests
Source: /home/juha/soft/pypractice/CSAdev/suites/softupdate_tests.robot
Start / End / Elapsed: 20171128 16:07:02.573 / 20171128 16:19:40.615 / 00:12:38.042
Status: 3 critical test, 3 passed, 0 failed
3 test total, 3 passed, 0 failed

SETUP s100_keywords . **Start Suite** 00:00:05.006

TEARDOWN CubeSatAutomation . **Client Close** 00:00:00.004

TEST **Upload new software and reboot using it** 00:05:24.282

TEST **Reboot to default software** 00:02:35.124

TEST **Upload invalid software image and see that nanomind returns to the default software** 00:04:33.371

Dayinthelife1 Mod Test Log

Generated
20180112 15:45:21 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	4	4	0	00:10:40	
All Tests	4	4	0	00:10:40	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-COMM	1	1	0	00:05:04	
OPMODE-IMAGING	1	1	0	00:01:12	
OPMODE-LOWOBS	1	1	0	00:02:38	
OPMODE-POWER	1	1	0	00:01:46	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Dayinthelife1 Mod	4	4	0	00:10:40	

Test Execution Log

SUITE **Dayinthelife1 Mod** 00:10:40.417

Full Name: Dayinthelife1 Mod
Source: /home/petri/s100/Testautomation/suites/dayinthelife1_mod.robot
Start / End / Elapsed: 20180112 15:34:40.870 / 20180112 15:45:21.287 / 00:10:40.417
Status: 4 critical test, 4 passed, 0 failed
4 test total, 4 passed, 0 failed

SETUP CubeSatAutomation . **Client Start** None, /home/petri/s100/EGSE/csp-client-v1.1/build/csp-client, -a 10 -z localhost 00:00:00.003

TEARDOWN CubeSatAutomation . **Client Close** None 00:00:00.005

TEST **Come from eclipse - Verify charging** 00:01:45.857

TEST **Come from eclipse - Take image** 00:01:12.159

TEST **Come from eclipse - Record radio signals** 00:02:38.110

TEST **Come from eclipse - Downlink data** 00:05:04.025

Dayinthelife2 Mod Test Log

Generated
20180209 13:22:47 GMT+02:00

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	5	4	1	00:50:03	
All Tests	5	4	1	00:50:03	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-COMM	3	2	1	00:44:31	
OPMODE-POWER	2	2	0	00:05:32	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Dayinthelife2 Mod	5	4	1	00:50:04	

Test Execution Log

SUITE **Dayinthelife2 Mod** 00:50:03.778

Full Name: Dayinthelife2 Mod
Source: /home/petri/s100/Testautomation/suites/dayinthelife2_mod.robot
Start / End / Elapsed: 20180209 12:32:44.067 / 20180209 13:22:47.845 / 00:50:03.778
Status: 5 critical test, 4 passed, **1 failed**
5 test total, 4 passed, **1 failed**

SETUP CubeSatAutomation . **Client Start** None, /home/petri/s100/EGSE/csp-client-v1.1/build/csp-client, -a 10 -z localhost 00:00:00.003

TEARDOWN CubeSatAutomation . **Client Close** None 00:00:00.008

TEST **Come from eclipse - Verify charging** 00:02:43.865

TEST **Come from eclipse - Set flight planner commands** 00:07:55.496

TEST **Go to eclipse - Wait during the time that the satellite is out reach** 00:20:15.702

TEST **Come from eclipse - Verify charging again** 00:02:48.304

TEST **Come from eclipse - Downlink data** 00:16:20.121