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 April 8, 2018

Thesis supervisor:

Prof. Esa Kallio

Thesis advisors:

D.Sc. (Tech.) Antti Kestilä

D.Sc. (Tech.) Juha Itkonen

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: April 8, 2018 Language: English Number of pages: 7+99

Department of ~~Radio Science and Technology~~

Professorship: Space Science and Technology

Supervisor: Prof. Esa Kallio

Advisors: D.Sc. (Tech.) Antti Kestilä, D.Sc. (Tech.) Juha Itkonen

Large portion of launched Cubesats have failed early on their missions. From the statistics of CubeSat missions, inadequate functional testing at system integration level has been suggested as a potential source for satellite failures. In this thesis test automation was used to perform functional system integration testing for Suomi100 CubeSat. Reusable software libraries for test automation were developed to be used in conjunction with Robot Framework. Among other tests performed, "Day in the life of a CubeSat" operational scenario test was executed and ~~we believe~~ that this test can reduce the possibility for infant mortality for any CubeSat. ~~Further, we suggest some form of guidelines for system integration testing to be appended to the CubeSat standard in order to improve overall CubeSat reliability.~~

Keywords: CubeSat 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 integraatio testaus

Päivämäärä: April 8, 2018 Kieli: Englanti Sivumäärä: 7+99

~~Radiotieteen ja -tekniikan laitos~~

Professuuri: Avaruustiede ja -tekniikka

Työn valvoja: Prof. Esa Kallio

Työn ohjaaja: Prof Esa Kallio

~~Suuri osa avaruuteen laukaistuista CubeSat-satelliiteista on epäonnistunut aikaisessa vaiheessa missiota. Lentäneiden CubeSat missioiden statistiikan pohjalta on tehty eräitä johtopäätöksiä että puutteellinen funktionaalinen järjestelmän integraatio testaus on ollut mahdollinen epäonnistumisten lähde. Tässä työssä suoritettiin funktionaalisia järjestelmän integraatio testejä Suomi100 CubeSat satelliitille käyttäen testiautomaatiota. Työssä kehitettiin uudelleenkäytettävä testiautomaatiokirjasto, jota voidaan käyttää Robot Framework testaustyökalun kanssa. Muun muassa "Päivä satelliitin elämässä-testi suoritettiin satelliitille ja uskomme että tämän testin suorittaminen pienentäisiä aikaisen epäonnistumisen todennäköisyyttä mille tahansa CubeSat satelliitille. Lisäksi, jotta CubeSat missioiden luotettavuus paranisi, suosittelemme CubeSat satelliitti~~ konseptiin lisättäväksi ohjeistusta järjestelmän integraatio testauksesta.

Avainsanat: CubeSat epäonnistumiset, Suomi100, Järjestelmän ~~integraatio testaus~~, Testiautomaatio, Päivä satelliitin elämässä

Preface

I want to thank Professor Esa Kallio and my instructors Antti Kestilä and Juha Itkonen for their good guidance.

Otaniemi, April 8, 2018

Juha-Matti Lukkari

Contents

Abbreviations

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

1 Introduction

1.1 Increasing interest in space

Since 1950s, mankind has ever increasingly started to set its foot into space [1]. Nations, industries, businesses, militaries, universities and even private entrepreneurs have sought out to benefit from the opportunities that space offers [1, 2, 3]. Great advancements have been made in technology and science to propel this endeavour even further [1]. Examples of such leaps in techology include sending first human into space in 1957, Moon landings in 1969, sending of probes into other planets in the Solar System like Mars and most recenlty getting the first images of Pluto in a flyby mission in 2013 [1, 4]. 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 space borne technologies and devices [5].

From the end of the 1990s, new inventions have as well brought 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 example by university students. Advancements leading to this can be attributed to the space industry cathcing up with the advancements of electronics as well as cheap launch opportunities coming available. More concretely, development of the *CubeSat* nanosatellite concept in 1999 in CalTech has in recent years brought about hundreds of new satellite developers and hundreds of new space missions based on this nanosatellite concept. [3, 19]

These satellites typically are relatively small and usually use commercial off-the-self (COTS) components, yet are still capable to operate in space around Earth. Cubesats have in recent years emerged as a new viable platform for carrying out space missions. Due to their small size the launch costs are smaller and their use of COTS components makes them relatively fast and relatively cheap to design and manufacture. Several companies have taken interest into the concept and other organizations (like US military) have shown interest into them as well [7].

As an example, Finnish government recently passed a new law regarding space and is pursuing to create new industry of space technology related companies in Finland [6]. The first Cubesats built in Finland in Aalto University have even created new space companies which are building their own satellites to be launched into space, like *Iceye* for example [8]. Nonetheless, usually most of these satellites have been produced by Universities and over 600 Cubesats have already been launched in total since 2000 [9]. 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 [21]. Already in 2014, approximately half of the flown space missions in that year were CubeSats [19].

1.2 Substantial proportion of failures in CubeSat missions

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

Michael Swartwout has done several studies into this subject. His research has suggested that majority of CubeSat failures could be attributed to inadequate testing of the satellite in a flight-equivalent state on ground. He has believed that functional testing of the whole integrated satellite system has been lacking completely or 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. [3, 19, 22, 23]

Though the concept of Cubesats shows promise, from the statistical data it can be seen that there are still some challenges for the satellite concept. Overall reliability of CubeSats is needed to be improved, if they are to become a valid alternative for the traditional time and resources consuming space missions [3].

1.3 Suomi100 Cubesat

The satellite involved in our research is called *Suomi100* which is a one unit Cubesat. The project was conceived in the interest of celebrating Finland's 100 years of independence. Mission of satellite is to take images of Finland from space and to measure different radio signals present in the ionosphere. An artistic presentation can be seen in Figure 1 below.

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 it in a systematic manner. By using software tools for automating the testing, we can achieve a certaing degree of systematicity and rigour. Similarily, verification tests for mechanical stress required by CubeSat standard are done systematically in an automated fashion [7]. So too it would be preferable that the functional system tests could be performed automatically in a systematic manner.

On 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 in *Robot Framework* [48] along with appropriate test suites and test setups.

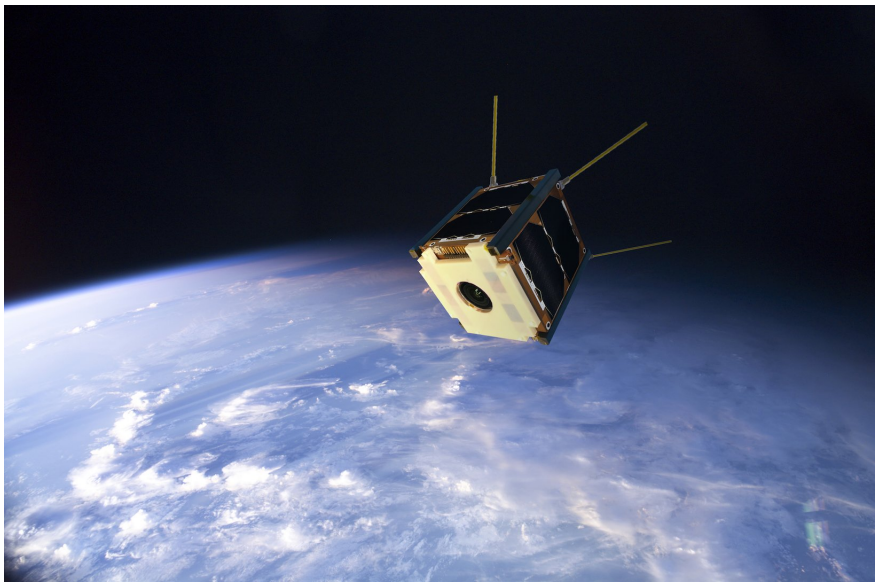


Figure 1: Depiction of Suomi100 satellite in space. Courtesy of Jari Mäkinen. [52]

1.5 Main questions and problems

The main problem is the unreliability of Cubesats, which the thesis tries to partly solve from the standpoint of system integration testing. ~~Could~~ this type of testing detect unrecoverable failures in the satellite operation and system integration and could we in addition verify that the satellite meets out its functional requirements?

1.6 Outlining the scope of research

~~We wish to study~~ the use of one industry-proven automated acceptance framework to carry out the automated functional testing on Suomi100 and ~~to~~ investigate the use of certain space-industry proven testing philosophies and methodologies into functional system testing with the satellite.

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, the simulation has to take into account the relatively small funds of an university led project.

2 Background

2.1 Cubesat failures

The CubeSat project was started in 1999 in California Polytechnic State University (Cal Poly) [7]. Over 100 universities and other organisations have since contributed to the project. The purpose of the project is to provide a standard for design of picosatellites in order to decrease development costs and to make accessability to space easier [7]. In fact, number of launched CubeSats has increased quite dramatically over the past few years [19] and it has been estimated that the number of CubeSat missions will increase in the coming years [21]. Yet, large portion of these missions have failed due to various reasons [3, 19, 22]. With only an average of 20 % of missions being able to complete the full mission envisioned for them [19].

2.1.1 CubeSat satellite

The CubeSat project defines a CubeSat to be a picosatellite with dimensions of $10 * 10 * 10cm^3$ and with a mass up to 1.33 kg [7]. Satellite of this size is considered to be 1U CubeSat. These units can be stacked together to form larger CubeSats, with some satellites consisting even of 12 units. Three units seems to statistically be the preferred size for a CubeSat [19].

Another important part of the concept is the *Poly Picosatellite Orbital Deployer* (P-POD), which is a Cal Poly's standardized CubeSat deployment system [7]. This deployment system is integrated to the launch vehicle and the springs in the system release the picosatellites into space [7]. Usually a launch vehicle carries some primary payload, which is much larger than the CubeSats [10]. If an extra weight can be launched along the main payload, then the deployment pods with the CubeSats are integrated into the launch vehicle as secondary payloads [10].

Though these picosatellites are considerably smaller than most of the "traditional" satellites, they nonetheless are able to perform the regular operations of a satellite [12]. As the classification, *picosatellite* 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 [12]. Subsystems containing electronics usually follow *PC/104* standard which defines form factor and computer bus [13]. A single subsystem in a CubeSat can fit into one *Printed Circuit Board* (PCB) following the PC/104 standard [13].

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. The subsystem usually in addition has some power regulation and power distribution features along some features for reliability. [11]

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

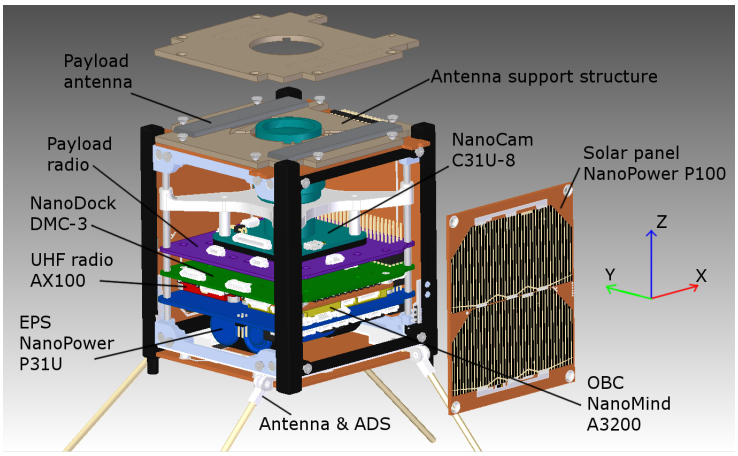


Figure 2: Depiction of Suomi100 satellite subsystems. Courtesy of Aalto University. [52]

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. [11]

The *Communication system* (COMM) 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 [11].

The proper orientation of the satellite is controlled by *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 ground station. [11]

In addition to these subsystems, *Support Structure* forms the subsystem which integrates all of the subsystems into a single mechanical structure [11]. The CubeSat standard defines the dimensions and materials for the support structure [7].

2.1.2 Failure rates of CubeSats

Over 600 Cubesats have been launched as of 2017 [9]. Some studies in recent years have been carried out to investigate the statistics of flown Cubesat missions. 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 contacted from space were also identified. Most active contributor to this topic has been Michael Swartwout and the representation of statistics of CubeSat failures in this thesis is mainly based on his work [3, 19, 22], as not too many papers have yet been published regarding this issue.

A study titled *The First One hundred CubeSats: A Statistical Look* and published in 2013 identified the failure rate for the first 100 flown CubeSats. Out of these first CubeSat missions flown between years 2000-2012, a total of 34 had failed. Out of these failures, third were never contacted after they were released into space (DOA cases). Since then, many CubeSats have flown with varying degree of success [19]. Figure 3 shows mission statistics for all CubeSats flown until 2017 [20]. The

different colours show at which state a CubeSat mission is or to which state a mission ultimately reached.

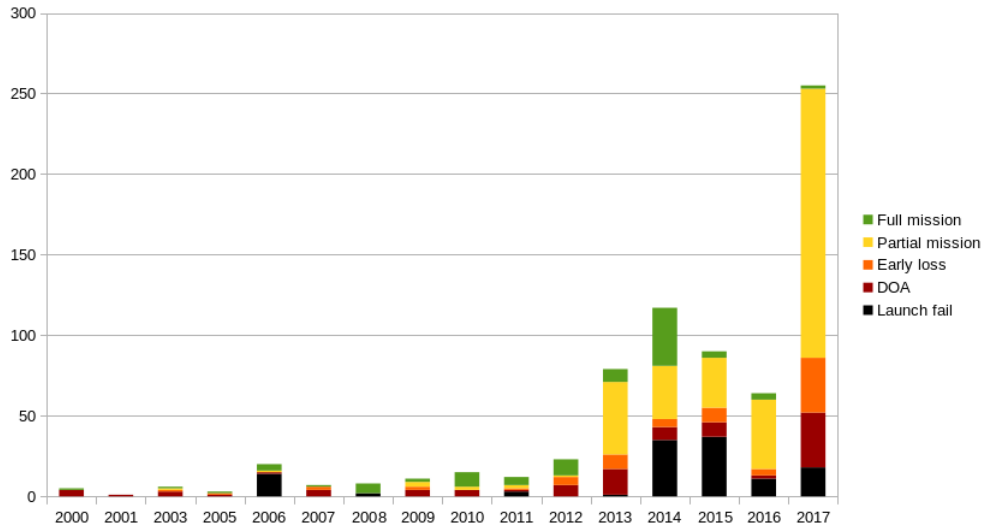


Figure 3: Mission statistics for CubeSats flown until 2017. [20]

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 [19, 22]. A study published in 2016 titled *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 % DOA cases and 9.8 % cases where the spacecraft was lost early in its life, meaning that communication with the satellite was established but no primary operations could have been 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 practices for integration and testing the numbers were: 6.3 % DOA and 6.3 % early loss. On the other hand, when the new University teams educated by their first failure flew their second satellite, the rates for DOA failures halved. [19, 22]

Below Figure 4 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.

For contrast, same statistics for CubeSats built by traditional contractors is shown in figure 5 below. A clear difference in DOA and early loss missions is visible for these two different teams.

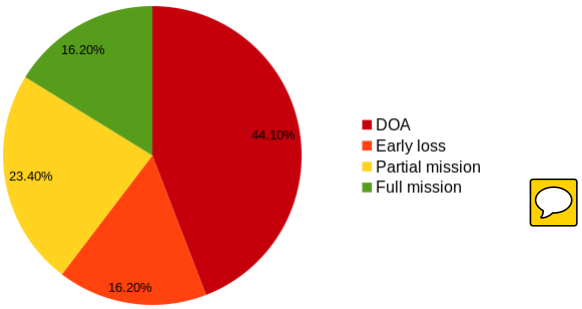


Figure 4: Statistics for CubeSats flown between 2000-2015 that were constructed by University teams without prior experience for satellite construction. [19]

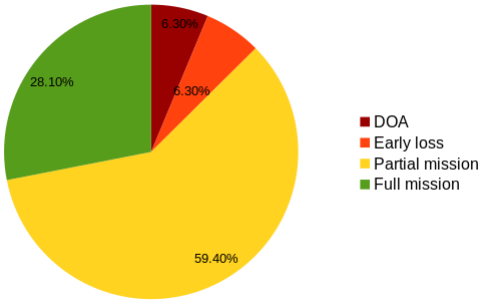


Figure 5: Statistics for CubeSats flown between 2000-2015 that were constructed by traditional contractors with extensive experience of satellite construction. [19]

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* [3]. The subsystems that were thought of been 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 COMM, 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 doesn't reboot anymore or gets unrecoverable stuck in some way. For EPS the error can mean for example that power is not being transferred to the satellite from the solar panels and for the failure in COMM subsystem it can mean for example that there is insufficient power for the antennas to close the link with the ground station. [3]

Based on the satellite developers believes about causes of failures, another study was carried out by Langer et al. in 2014 [23] to investigate in more detail the contribution of different subsystems in CubeSat failures. This study by Langer also used statistical data of CubeSat failures obtained from CubeSat Failure Database (CFD), at that point comprising data about 178 CubeSat missions. With this data a reliability estimate for different subsystems was calculated using Kaplan-Meier estimator for nonparametric and parametric analysis. In addition, a parametric

model for total CubeSat satellite reliability was devised. Figure 6 below depicts the subsystem contributions to satellite failures for the first 178 CubeSat missions. Three main subsystems causing failures were identified to be in order: EPS, OBC and communication systems, in accordance with Swartwout research, but with different percentages as EPS being the main contributor to failures [3, 23].

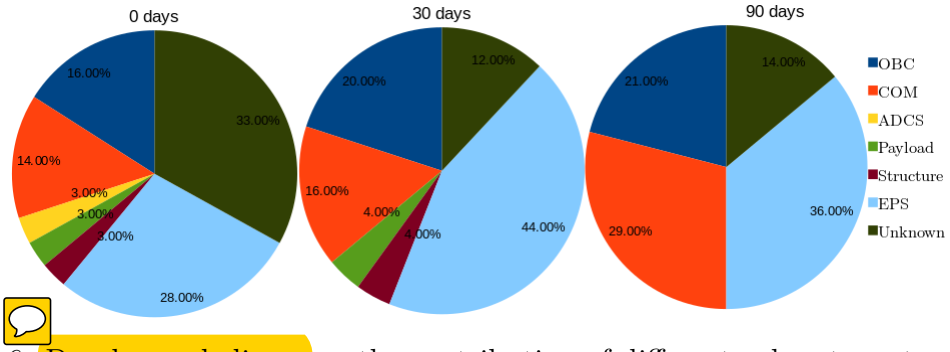


Figure 6: Developers believes on the contribution of different subsystems to satellite failure. From left to right the charts present failure contribution data for 0, 30 and 90 days after launch. [23]

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 % change that the satellite fails. Yet, the developer's belief seems to be too optimistic when compared to the data gathered from CFD. Nonetheless, from largest to smallest in probability to cause system failure, the main subsystems were identified in the order COMM, EPS and OBC.[23]

2.1.4 Needs for the system level functional testing

The aforementioned studies made some anecdotal guesses to what could have contributed to the failures in the satellites so that the missions failed either partially or completely. Though the current data doesn't clearly prove these guesses, it has been believed by Swartwout and others that system integration level functional testing of the satellites has been lacking completely or has been inadequate [3, 19, 22, 23].

Based on his study in 2013 [3], Swartwout came to a strong belief that the critical failures in the subsystems were caused by poor functional 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 this paper 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 of the satellite to test. [3]

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 [19, 22]. As an example, the ORS-3 mission flown in 2013 consisted of 28 secondary payloads and out of these 13 were assembled by new university teams flying their first satellite and 15 were constructed by traditional contractors [19]. 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 quite possibly missing from the university built CubeSats.

In addition, David Voss of the Air Force Research Lab speaking more recently in the 31st Annual Conference on Small Satellites held at 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 [24]. 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 [24, 25]. He noted though that the goal is not apply the same rigorous assurance procedures used earlier in larger and more expensive spacecrafts, but to design new procedures and keep some of the familiar methods that can be useful.

2.1.5 Comparison of CubeSat failures to failures with larger spacecrafts

Besides Cubesats, failures have happened to the more traditional spacecrafts as well. In fact, the history of space industry as a whole is filled with examples of failed missions [26]. As an example, in recent years several Mars landers have failed during landing phases of the mission [27]. For instance, Mars lander *Schiaparelli* crashed on the Martian surface in 2016 when a sensor used to measure distance to the ground read a negative value and shut off its descent thrusters [28].

Earlier research about the on-orbit failures was carried out in 2005 [29]. It investigated failures of 129 different spacecrafts 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. [29]

A research conducted in 2008 analyzed the contributions of different subsystems in failures of 1584 satellites that were launched between 1990 and 2008 [30]. 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 time (i.e. years), the main subsystems contributing to failures were

identified to be the ADCS and COMM. Some similarity to the subsystem failures with CubeSats can be drawn from here, with the communication subsystem and the solar panels playing a crucial role in infant mortality of CubeSats.

Further prove 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"* [31]. 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 [31]. 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 [32].

In conclusion, based on the experiences found by 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 spacecrafts can be classified as *embedded systems* [33]. 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"* [14]. Systems like these can contain many parts and consist not just of software but of hardware elements as well [34].

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 [34]. For example, mechanical stress and thermal vacuum tests are required by the CubeSat standard to be performed for CubeSats before launch [7]. 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 like this is in practice testing of the software in the final hardware environment [34]. 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, *who 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. [16]

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 [14, 15]. 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 [15].

Methods of testing

Various different methods for software testing exist. So called box approach is one common method for testing. Testing can either be done automatically by some computer run script or manually by a tester who follows a specified test plan. [14]

Below are some of the testing methods explained 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 outputs we see how did the system perform. If the outputs were what we expected, the system passed the test. Black box testing is usually used when we are interested in the functionality of our system under test. Figure 7 below illustrates black box testing method. [15]

White box testing
White box testing is performed with interest about the internal structure of the SUT. Testing of the internal functions rather than the 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 7 below gives an illustration about white box testing. [15]

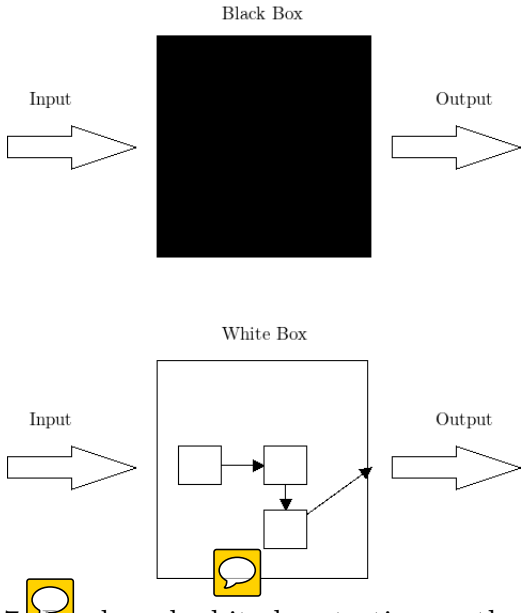


Figure 7: Black and white box testing methods. [15]

Input selection
There exists some philosophy in choosing the right 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 some logical set of combinations and not all possible ones. [15]

Other approach in choosing the right inputs is to look for the requirements and specifications defined for the software [15]. 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 [15]. A single *use case* describes how to use the system in order to reach a particular goal [14]. In other words, a use case is a description of an operational scenario of the system.

Test case
A set of inputs along with 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. [17]

A collection of test cases with the focus on testing a specific area of a system can be called to be 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 is done by starting from smaller parts and gradually going into larger components of the system.[14, 16]

One can define different levels of testing as follows [14, 15, 17]:

Unit testing
Unit testing is the most basic level of testing. On this level, individual components of a software 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 using some of the combinatorial methods.

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 bigger component itself works properly. Both black box and white box method can be applied to carry out the tests. Inputs for test cases are commonly derived using some of the combinatorial methods.

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.

Acceptance testing

At this level testing is usually performed by some outside team that wasn't involved in the development of the software. This team could mean final users or there could be a separate testing team to ensure that the final product conforms to the original requirements defined by the final user. 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. 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. [18, 34]

Types of testing

Besides the method and level of testing chosen, there exists multiple different types of testing that can be performed. [14, 16, 34]

The most common ones are the following [14]:

Functional testing

Functional testing is performed when we are interested in knowing what the system under test does based on our input 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, like e.g. performance testing or security testing. 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 stabile 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 tests can be executed once every change has been made to the software. [14]

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, 34]. From simulating the target hardware on a computer to running tests on a integrated satellite, different methodologies exist to account for different environments [37]. In addition, satellites consist of several subsystems (as described in section 2.1.1) and each have their own softwares. Each of these softwares are tested individually and finally together on the integrated satellite [14, 37].

In NASA, on different levels of system development different environments and different teams are used for testing [37]. During the course of the Apollo program, NASA adopted the four level software testing practice [36]. At the lowest level, unit 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 [37, 38]. This environment contains assembled subsystems, interface emulators and ground and flight softwares. The goal is to verify proper required software functionality. As an example, a system testbed was used to test operation of singular and several subsystems of the Cassini-Huygens space probe [38]. 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 [37, 39]. For example, downlink procedures, maneuvres, payload operations and so forth are tested on this level. This test is usually performed by a separate integration test team [37]. Below is Figure 8 describing 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 [40]. ESA also states that a system verification by testing shall consist of testing of system performance and functions under representative simulated environments [40]. As can be seen, testing is done in the same vein as what is done in 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 [41]. A mantra commonly used in space industry has been *"Test like you fly"* (TLYF), meaning for one that spacecrafts should be tested on ground in the same way as they would be operated in-orbit [31, 39]. On general, the TLYF philosophy provides a basis for acquiring and verifying a system and gives a mission-centric focus on space system validation and verification [39]. As such, same software and same hardware should be used in testing as will be used when the spacecraft is launched to orbit [31]. One test on system integration level using this philosophy is commonly referred as *"day in the life"* or operational scenario testing [31, 39].

In "day in the life" testing, tests are derived from mission operations requirements

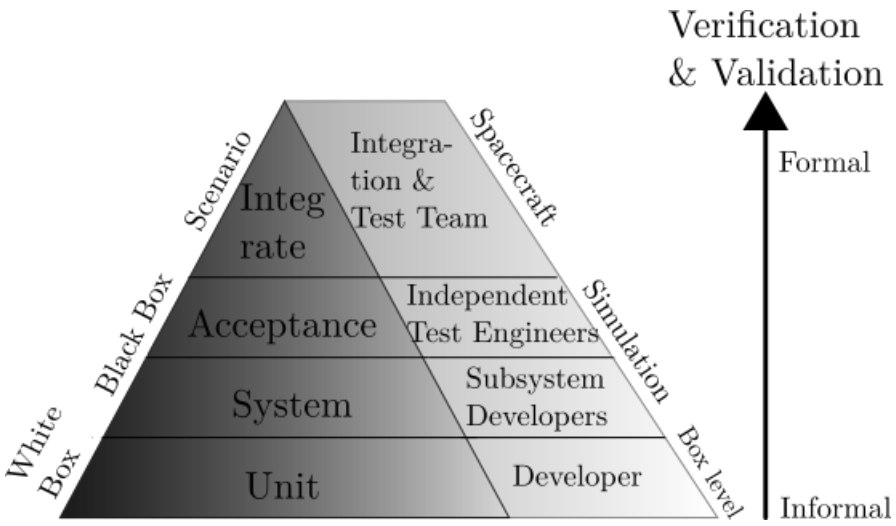


Figure 8: Illustration of spacecraft testing on different levels of system development. [37, 39]

documents [39]. A document called *operations concept document* or CONOPS is commonly used for this type of documentation [39]. 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 ground being flown in a flight-like manner to the extent feasible. In addition, controlling and communicating with the spacecraft from the ground station in the way that has been envisioned in the mission operations requirements document [39]. This type of testing has been deemed necessary as many failed space missions had been succesfully tested to meet all their requirements, but were not tested to verify succesful completion of mission objectives [39]. This test is required by NASA GSFC and ESA to be performed before a spacecraft can be verified for mission readiness [39, 40].

It has to be further noted, space as an environment itself provides extra challenges for maintaining proper reliability of spacecrafts. Variations in temperature as well as particles carried by the solar wind put unique demands for reliable design. These devices operating practically out of our physical reach impose further demands for system reliability. If in a satellite orbiting at an altitude of 500 km happens 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, testing of spacecrafts has to be thorough and systematic. Higher demands are usually set for testing of spacecrafts than for terrestial systems. [11]

2.3 Test automation

Software test automation has among software projects been a topic of interest for the past few decades [43]. It has been proclaimed as a solution for decreasing costs related to testing and enabling release of human resources for other tasks [44, 45]. 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 [45]. Several softwares and frameworks for test automation have been developed over the brief history of test automation [43, 45].

2.3.1 Test automation frameworks

A test automation framework is an integrated system that sets the rules of automation of 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 are needed to be modified. [46]

A common practice is that test cases are written into separate scripts with a scripting language specific to the framework [46, 48]. Function libraries are written into their own source code files with some of the more common programming languages like Python, Java, C++ and so forth [48, 47]. The test scripts then call for the functions in the function libraries to perform the actual automation [47]. 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 [48]. The framework emerged from a Master's thesis written by Pekka Klärck for one Finnish software testing consultancy company known as *Qentinel Oy*. Title of his thesis was "Data-Driven and Keyword-Driven Test Automation Frameworks" and it was written in 2005 [48, 49]. In turn, the writer of this thesis at hand has also been working at Qentinel and thus has become quite familiar with Robot Framework. This is one of the reasons why Robot Framework was chosen for test automation of Suomi100 satellite.

Robot Framework is in addition open-source under 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. The modularity and flexibility of the framework has made it possible to use the framework to do test automation on various different projects. Some companies like ABB, Nokia, Kone, Metso, Axon, Zilogic and others have used Robot Framework in testing of embedded systems. While other companies like Finnair have been using it to test their web based applications. Some companies like ABB, Metso and others are performing their testing with Robot Framework in many different areas. U.S. Naval research laboratory has also been using the framework with their SAGE multi-agent system. [48]

Based on how many companies have been confidently using the framework [48] and that it has been used in many different areas (embedded systems, web applications, etc.), we feel confident to develop test automation for Suomi100 satellite with Robot Framework. In addition, the framework being open-source makes it even more appealing for this task [48]. Future CubeSat projects could use Robot Framework as well and possibly with the generic libraries that are created in this thesis.

Robot Framework uses *keyword-driven* testing technique and the test scripts have a tabular data syntax. With keyword-driven testing, the test cases consist of keywords that perform 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 to be simple function calls (performing one action). Below in Figure 9 a Robot Framework test suite script is shown. [50]

```
*** Settings ***
Library         String
Library         AutomationLibrary  1
Suite Setup     Start Suite
Suite Teardown  Close Suite        2

*** Test Cases ***

Set satellite orientation  3
        Verify Connection
    4   Set Orbital Elements    10      20
        Set ADCS                1       2       3


Measure particles with payload
        Verify Connection                           5
        Startup payload         10
        Measure particles       HIGH    100000
        Store Measurment        /flash/data/measurment.dat
        Downlink Data           /flash/data/measurement.dat
```

Figure 9: Example of a Robot Framework test suite with two test cases.

In Figure 9, the number (1) on the script shows how function libraries are included to test execution. These libraries can simply be Python files or Python classes. In that case, a simple Python file can be included to test execution by defining a relative path and the filename. Python modules which are included in operating system *PATH* variable can be included directly.

Number (2) on the script illustrates how test suite setup and teardown scripts are included. These calls to start and close the test execution can as well be Python functions included in the function libraries. Call for *Suite Setup* defines what operations are peformed at the start of the test execution, like opening of the software under test, initiating network connection to the software and so forth [50]. Calling for *Suite Teardown* defines what actions are performed when test execution ends [50]. Like closing of the software which we are testing and so forth.

Number (3) shows in what way the test cases are defined. The names of the test cases are arbitrary. Though they should be named differently from each other and the naming should represent the activity of the test case [50].

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 [50]. Spaces and letter cases don't matter in the keyword when calling for a Python function from Robot Framework.

Under number (5) the parameters for the representive keywords are defined. Parameters are separated from the keywords and from each other by arbitrary number of tabular spaces [50].

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 test execution log files are generated in *Hypertext Markup Language* (HTML) by Robot Framework. Several Robot Framework log files can be seen in Appendix D. [50]

2.4 Suomi100 satellite mission

Suomi100 satellite is the system under test in this thesis. The mission was conceived in 2015 in the interest of celebrating Finland's 100 years of independence [51]. The original design called for a 2U CubeSat, but was later changed to a 1U CubeSat. The mission demands to have two payloads on board the satellite. First payload is a white ligth camera in the interest of taking images of Finland from space and the second one is a science instrument which measures the radio static in the ionosphere [51, 52].

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 by northern lights.~~"

Table 1 on the next page shows requirements derived from this requirement.
The second requirement defines the operational requirement as:
"Suomi100 is a CubeSat."

This requirement defines on general that Suomi100 must meet requirements defined for CubeSat standard and other operational requirements like power consumption and downlink speed. For the interest of this thesis, going through them in detail is unnecessary. Only to note that Suomi100 meets the requirements set for CubeSat standard and 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.

Table 1: Suomi100 functional mission requirements derived from the requirement to take images of Finland and ~~measure~~ *Radiofrequency* (RF) radiation ~~caused by northern lights~~.

1st Derivation	2nd Derivation	3rd Derivation
Take one image of Finland per day	Capable of pointing camera towards Finland Must compress images for faster downlinking Image resolution shall be adequate to discern geographical features Must take images at both day and night time	Camera points with 5 degree accuracy RAW, BMP and JPEG output formats ~~60~~ m / pixel resolution 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 Adequate resolution for scientific measurements Must compress data for faster downlinking	1-10MHz region measured in 6 kHz ~~strips.~~ Sampling frequency 48 kHz. 50 samples from each 6 kHz band. Radio resolution 16 bits. AGC resolution 5 bits. 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. Software includes scheduler.	Ground station uses Cubesat space protocol

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 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 atleast 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 atleast 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 atleast 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 atleast 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 start immediately when the satellite is deployed. The correct commands for the thermal knifes 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, few macro modes containing both of the payloads are defined as well.

First mode for the radio instrument is tied to the *Low observation* operation mode. With this instrument mode, we use a single frequency to measure signals in the ionosphere. Table 2 shows 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

Like 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.

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 like in mode two. But the frequency what 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 what arguments 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, image quality and other parameters. In addition, this instrument mode is part of the *Imaging mode* operation mode. Table 5 describes these parameters in more detail.

By compining some of the instrument modes for the radio instrument and for the camera, several different macro modes can be constructed. For example, first

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

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

performing the first mode for the radio instrument and then using the camera with its instrument mode and finally performing another measurement with radio instrument mode 3.

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

2.4.4 Automated functional system integration testing

The testing of 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 Suomi100 satellite. *Automated* refers to the fact that we are using one automated testing framework for test execution. *Functional* comes from the reason that we test what functions the satellite performs based on our inputs and commands to it. *System integration level* refers to the fact that we are performing testing on the whole integrated satellite. Testing is carried out this way because, as mentioned earlier in section 2.1, this kind of testing has possibly been lacking in the previous CubeSat missions and carrying out these tests could possibly mitigate failures that occur in the early life of a satellite. In addition, we attempt to verify that the satellite functions according to the requirements we have 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 thought of causing failures with CubeSats or that testing of them has been inadequate. In addition, 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 like housekeeping, safe reboots, software updates and so forth. Tests for the (4) "day in

the life" operational scenario testing will be performed likewise.

Approach for 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 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 radio payload will use externally generated radio signals as input. The tests for "day in the life" scenarios will use a solar simulator and the satellite will be commanded over 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 in a test case leads to the test case to fail. Test steps are failed based on the responses of the satellite control software.

3 Methods

3.1 Suomi100 satellite

Suomi100 satellite is assembled together with a 1U Cubesat manufactured and designed by Gomspace company from Denmark. This 1U Cubesat, which is known as *NanoEye* in Gomspace product catalogue, forms the platform and main systems of the satellite [53]. This part of the satellite is referred as *platform* in this thesis henceforth. Picture of the platform is shown in Figure 10. On top of the platform, we added another payload, which consists of an *Amplitude Modulated* (AM) radio on a PC-104 type PCB and two ferrite antennas and a support structure. All designed an assembled in Aalto University by members of the Suomi100 satellite team. This part of the satellite is referred as *radio payload* in this thesis.

The subsystems and the satellite platform have flown in space aboard other missions successfully [54]. The platform forms a relatively well tested system with which we can investigate the development of automated functional tests for Cubesats [54]. In addition, the radio payload and its control software integrated to the platform give us 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.

~~The mission~~ of the Suomi100 satellite is to take pictures of the northern hemisphere, especially of Finland. The satellite flies in the 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 devised.

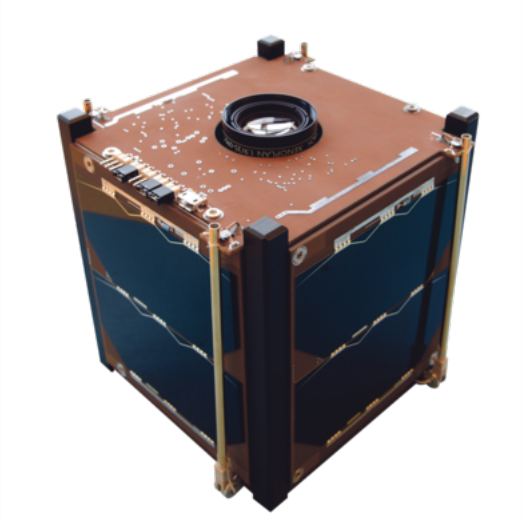


Figure 10: GomSpace NanoEye 1U. Courtesy of GomSpace A/S. [53]

3.1.1 Subsystems

The satellite consists of several subsystems. The central subsystem of any satellite is the system with the computer designated as OBC. In our platform it is known *NanoMind* and it is based on a Atmel 32-bit microcontroller [55]. Another vital system to the satellite is naturally the EPS and it is known as *NanoPower* in our platform [56]. Communication system of a satellite is the system responsible for receiving commands from the ground and responsible for sending information back to the ground as well. In our platform the communication system is known as *NanoCom* [57]. 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 ~~AM~~ frequencies. The camera came along with the GomSpace platform and is known as *NanoCam* in their catalogue [58]. Below the most essential subsystems to the topic of this thesis are desribed in more detail.

On-Board Computer - Nanomind

Figure 11: Nanomind OBC inside its casing. Courtesy of GomSpace A/S. [55]

The Nanomind A3200 On-Board-Computer shown in Figure 11, is based on Atmel AT32UC3C *Microcontroller unit* (MCU), which is a 32-bit *Reduced Instruction Set Computer* (RISC) microcontroller 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 sort of a mediator between subsystems and routes communication between them correctly. 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 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. Below in Figure 12 is a block diagram of the OBC. [55]

Electrical Power System - NanoPower

The Nanopower P31 on our satellite contains two lithium-ion batteries and has several reliability features. Figure 13 shows a picture of the subsystem. The bat-

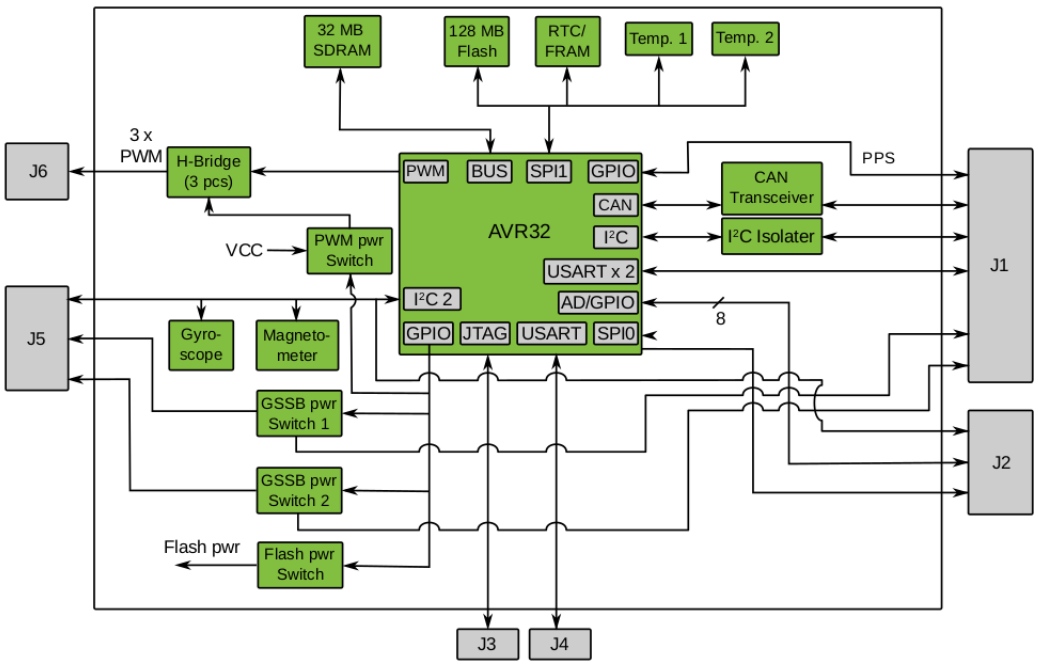


Figure 12: Block diagram of Nanomind. Courtesy of GomSpace A/S. [55]

Figure 13: Nanopower EPS. Courtesy of GomSpace A/S. [56]

teries 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. [56]

Communication subsystem - NanoCom

The NanoCom COMM system shown in 14 is a software configurable half-duplex transceiver designed for long-range transmissions. Certain parameters of the system can 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

Figure 14: NanoCom communication system. Courtesy of GomSpace A/S. [57]

Low-noise amplifier (LNA). [57]

Camera payload - NanoCam

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

Figure 15: NanoCam payload camera. Courtesy of GomSpace A/S. [58]

camera, presented in Figure 15. 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 clock rate of 536 MHz, 512 MB of DDR2 memory for image storing and processing and of a 4 GB eMMC flash drive with 2 GB for image storing. [58]

The software for image processing and storing runs on a customized embedded *Linux* (GomSpace Linux) 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 on-orbit, such as exposure time, different gain values, gamma correction and so forth. [58]

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. Figure 16 shows the PCB of this subsystem. Central to the system is the PCB which has *Silicon Labs Si4740* automotive *Amplitude Modulated/Frequency Modulated* (AM/FM) Radio receiver on

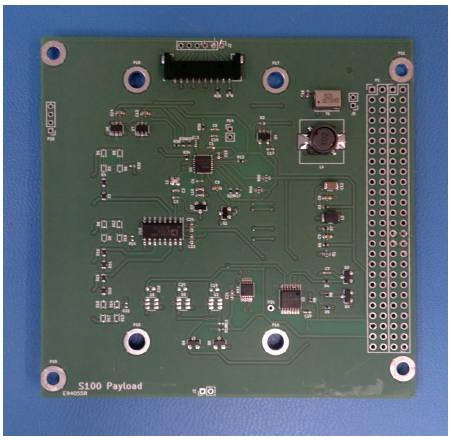


Figure 16: Second payload of Suomi100, AM radio instrument. Courtesy of Aalto University.

an *Integrated Circuit* (IC) [59]. 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. Including 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 [60].

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, titled "Ferrite Rod Antenna in a Nanosatellite Medium and High Frequency Radio" [61]. 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 material that can sustain the environment in space.

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 *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. [62]

The protocol as well as the software for 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 *housekeeping* (HK) data, parameter tables for adjusting 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 *Minicom* software. [62]

The software developed by GomSpace for NanoEye additionally includes such general functionalities as *File Transfer Protocol* (FTP) running over ~~CSP. With~~ which files and data can uploaded and downloaded from the satellite. Some 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. The software in the satellite can in addition be updated via the FTP by uloading a software image to the satellite and telling the computer to start reading from it after next reboot. In addition to these, *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. [62]

The operating system running in the NanoMind OBC is a free *Real-time Operating System* (RTOS) known as *FreeRTOS*, which is a light weight operating system designed for embedded systems that use microcontrollers and small microproserssors [63]. It was developed by *Real Time Engineers Ltd.* in USA. The version of the operating system used in our satellite is 8.0 at least. The operating system is mostly written in C programming language, but certain necessary parts are written with *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 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. [63]

3.1.3 Satellite control software - CSP Client

The ground station software used to control the satellite is known as *CSP Client*, which is a simple console program for remotely sending commands to the satellite via CSP. The program is 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 our radio payload among other things. In Figure 17 the CSP client is shown running in *Debian 9* Linux, showing among other commands a command inquiring for housekeeping data from 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 of Suomi100 are presented here:

reboot <CSP node>

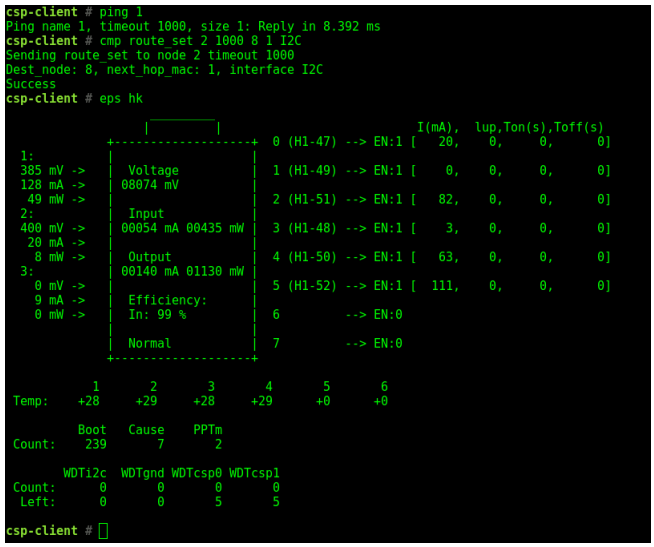


Figure 17: Suomi100 ground station control software running in Linux.

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 to 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.

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 *Finnish Meteorological Institution*. Unlike in most of the subsystems in Suomi100, the radio payload doesn't have its own microcontroller or any other general purpose computer. Therefore, the control software operates as 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 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. Commanding of the Si4740 is based on the hexadecimal values of the bytes it receives [60]. The first byte received defines which action is being performed and the following bytes define the arguments for that respective action [60]. 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 [60].

Over 50 different arguments for different commands can be used when controlling the Si4740 [60]. Therefore, when commanding the radio payload to perform a measurement, the different values for different values are loaded either from a configuration file or from a GomSpace paramater table. The parameter table and the configuration file were added to NanoMind by us. 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>
Get the value of a certain property defined in the Si4740.

3.2 Automating testing of Suomi100

3.2.1 API and communication layers for CSP Client software

In order to automate the control of the satellite, we need some form of interface 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 [55]. In addition, all of the subsystems can be controlled from a single shell via a serial to USB FTDI cable connected to NanoUtil USB port [64]. As presented in previous subsection, a separate CSP client software exists, which can be used to control the satellite from the groundstation via a radio link and it can also be used to control 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 do 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 creation of a *POSIX thread* (pthread) which runs a function that opens and listens to a socket connection on the *localhost* local network address. 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 Linux operating system. This could have been made through Linux output and input *standard stream* redirection methods like *pipes* [65]. Using the network connection has potential of making the automation externally controllable though.

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 Linux *kernel* keyboard driver, is that we can automate the use of not just our CSP client but the use 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 in our case. 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. But if a user uses the computer during testing, the keypresses can be received by programs that we didn't wish to automate.

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

Besides requiring the method of how to send commands to the satellite in an automated fashion, we also require to 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 did the satellite behave. 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.

Other way to the capture responds 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 our function library. Doing the transmission of CSP client output to the test automation libraries this way was experimented by the use of some Linux output redirection routines like *dup*. 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 generate a generic test verification solution to this as well. Provided that the process which we wish to do automated tests with responds through the Stdout stream, which fortunately happens to be the case for most terminal programs [65].

The solution for the communication is illustrated in Figure 18 and the modified main.c for the CSP client can be found in the Appendix B.

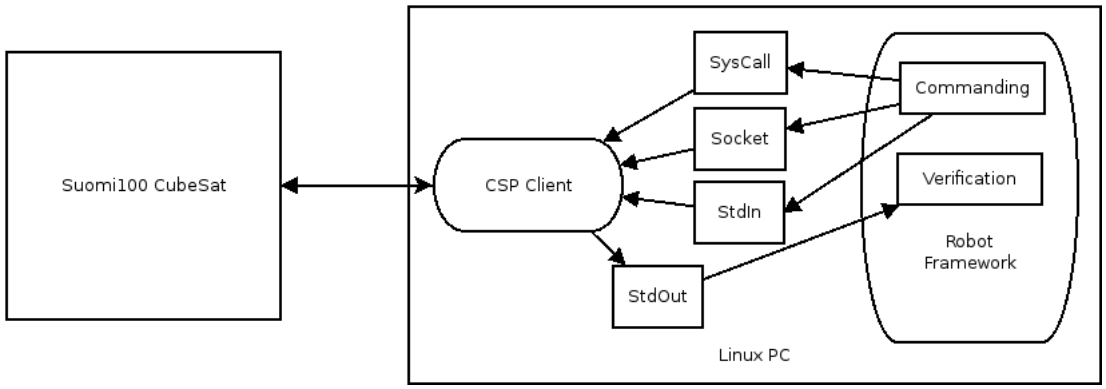


Figure 18: Illustration of the software architecture developed for Suomi100 test automation. Large rectangles represent environments, small ones represent layers and rounded rectangles represent programs.

3.2.2 Python libraries

A set of function libraries using Python programming language were written. All of these libraries consisted of one Python class each. 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 pyautogui library. Furthermore the library includes the methods to read and verify the process replies from Stdout. As can be seen in Figure 18, the commands are sent to 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 CSP client and 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 don't 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 [50].

The essential core methods used in the CubeSatAutomation Python class are presented below 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 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 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 stored temporarily or discarded.

Persistent Command <message> <exception replies> <end reply> <time-

out> <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 for 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 our test automation library. This library only has the aforementioned methods to start and communicate with a desired Linux program running on terminal shell (or bash shell) and keywords that are more specific to 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 desribed 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. 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> <auto-gain>
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 CSP client program.

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

Radio Startup <switch input> <switch power> <antenna input>
Setups 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 a 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 previous section are used directly as such. Because, the approach was to make 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 didn't have large set of specific, though descriptive keywords, which is common with Robot Framework. Besides, our 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 our satellite project 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. Purpose of each test case is to verify functionality of some aspect of the 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 are needed to be dependant on each other and in such cases the satellite is not specifically set to a certain state.

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

3.3 Test setups and environment simulation

As defined in section 2.4, different aggregates for testing were identified from Suomi100 satellite. 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 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 colour and brightness values as what the camera would see while in orbit. The easiest solution is to simply take the whole integrated satellite outside on a bright day to the balcony on top of our department in Aalto University. The satellite is standing on top of a stand and the side with the camera lense is directed towards horizon. A PC with CSP client and the test automation tools are connected to the satellite via the USB connection on NanoUtil. In addition, the satellite is loosely enclosed in a plastic container to protect it from particles in the air.

Below is Figure 19 of the test setup used during the testing.

3.3.2 Radio payload testing

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 was 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.

Figure 19: Suomi100 satellite on a balcony during imaging mode tests.

The radio noise is generated with a *HackRF One* Software Defined Radio (SDR) which is connected to another PC running *GNURadio* signal processing software.

Figure 20 shows the setup for the testing of the radio payload.

Figure 20: Radio payload testing with *HackRF One*.

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 can't produce signals with lower frequencies 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 as based on the research done on the radio signals in the ionosphere, this frequency would be of most interest to us [66].

Figure 21 shows *Fast Fourier Transform* (FFT) plot of the noise that is generated from the GNURadio, which is then transformed into radio waves by the HackRF.

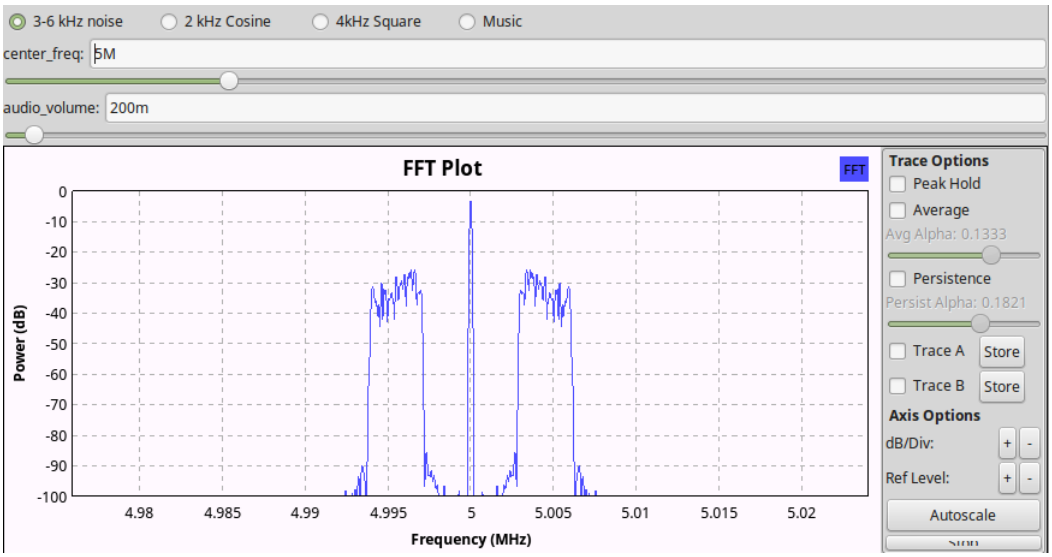


Figure 21: Screenshot from *GNURadio* signal processing software showing FFT plot of a radio noise being generated.

3.3.3 Satellite basic operations testing

For testing of the basic satellite operational features like 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 22 we have a picture of this setup.

3.3.4 Operational scenario testing, "Day in the life"

In "day in the life" operational tests, the Sun is simulated with a 1800 Watt Xenon lamp that is situationed 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 in 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

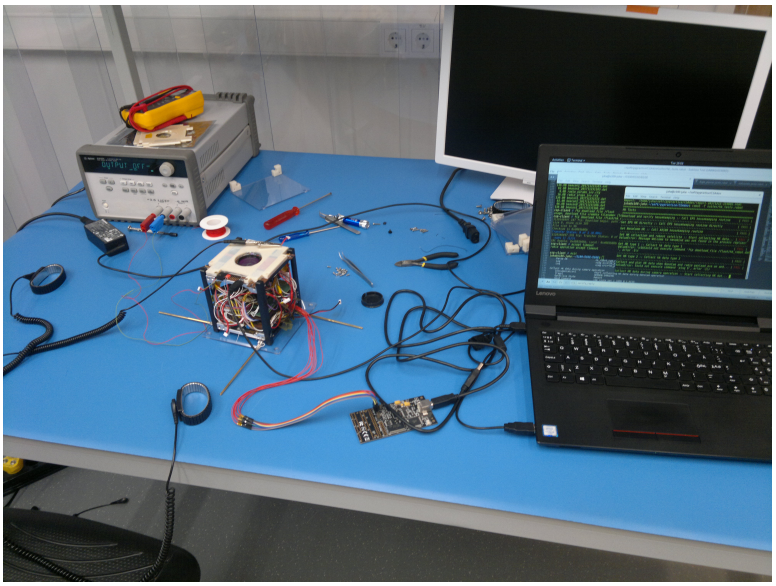


Figure 22: Setup for testing of basic functionalities of the NanoEye platform. Suomi100 is on the left and a PC with CSP client and test automation softwares is on the right.

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 Celcius in our clean room). The heating and cooling was measured with *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 23 this setup with the solar simulator is presented.

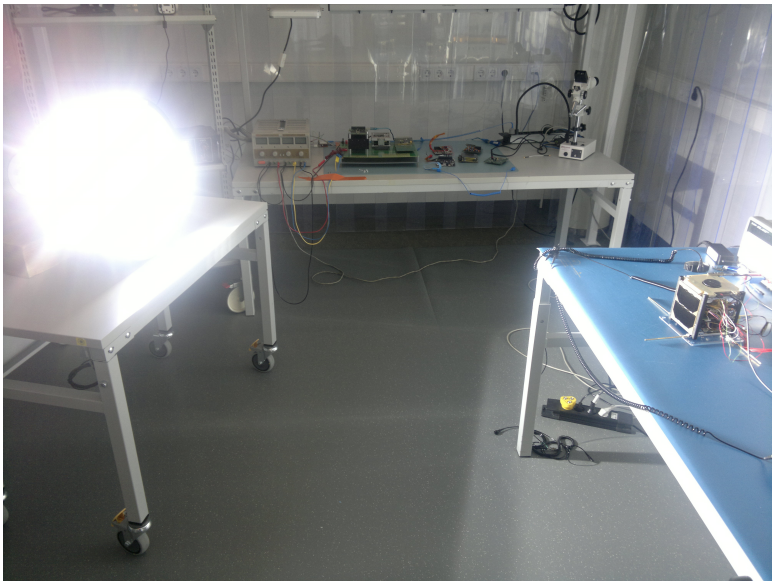


Figure 23: Day in the life setup for the satellite. The Xenon lamp is on the left and Suomi100 CubeSat is on the right in the picture.

For the "day in the life" testing, these periods form the phases of the operational 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 and 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 10 minutes communication window 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 radio link. A SDR with model *Ettus USRP B200* is connected to a PC with the CSP client software and the test automation softwares. 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 groundstation and all of its hardware is not used because the solar simulator has to be controlled manually and the groundstation is situated several floors up from the Aalto University's Space laboratory. Nonetheless, the same software and the same SDR are used as what will be used in the groundstation.

Figure 24 presents this setup.

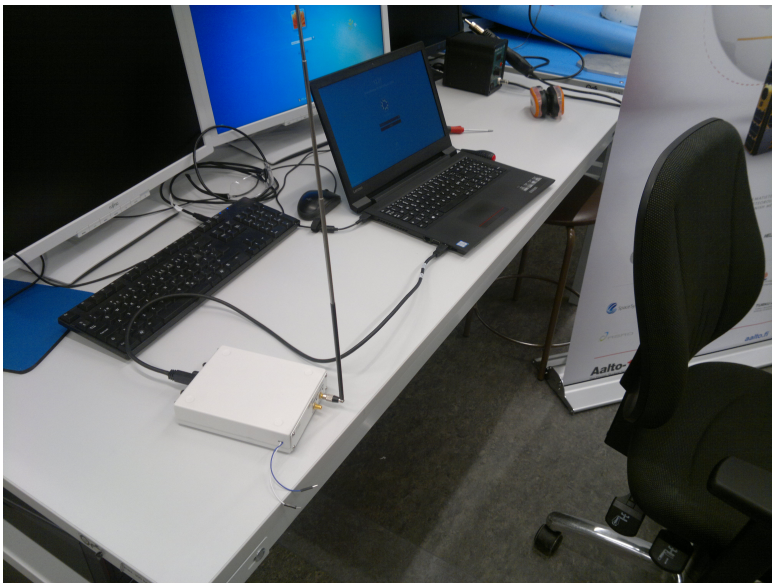


Figure 24: Day in the life setup for a "stripped down" groundstation.

4 Results and Discussion

4.1 Executed tests

The Robot Framework test suites executed can be divided into three categories: test suites done for the two payloads (1), test suites written for general satellite operations (2) and test suites for "day in the life of the satellite" operational scenario tests (3).

The first category of suites follow more traditional method of combinatioral testing. Each test case was derived from the operation modes defined for the camera and radio payloads. Each of the test cases for their representative operation mode are identical in steps, but use different combination of values in the keyword arguments. Since the payloads perform only singular functions, it was felt that using combinatorial input values was necessary to have adequate test coverage for these payload operation modes.

The second category consists of test suites made for the functional testing of the commands that contribute to the basic operations of the satellite. These test cases for different operations are completely different from each other in design and the tests here begin to resemble different smaller scenarios or use cases. Such cases as telemetry gathering, flight planner commands, software update, etc. In addition, a test suite was written for testing 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.

The third category, follows the higher level satellite system integration tests where we test the satellite based on operational scenarios defined for a "Day in the life of a satellite". Separate test suites were written for each operational scenario. The test cases at this level are simply different phases in the operational 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 limited set of test cases, testing each of the operation modes. Yet, no proper continous integration chain was done during this time, but the smoke tests were manually started usually once every week.

4.1.1 Camera payload

The test cases for the NanoCam were identical in structure, or in other words, we made several test cases for the same use case of the camera payload. Only difference was that the main parameters like gain value and exposure time differed between test cases. These two are the main parameters according to the NanoCam manual provided GomSpace [58]. Having the exposure time fixed, we demonstrated the use of combinatorial testing to go through different variations of the 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 $10ms$, $30ms$ and $90ms$. Below is Figure 25 showing the test case structure used in 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 25: Robot Framework test case structure for camera payload testing. The 1st column represents the command or action performed in CSP client (see section 3.2.2-3.3.3), second, third and fourth columns define parameter options and values for the commands.

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

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 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, we could have gotten some baseline information about the affects of the different parameters on the images.

Nonetheless, no test cases failed due to software errors. These tests on the camera ~~provided certain confidence~~ that changing different parameters didn't crash the software or that any combination of parameters didn't cause problems in the functioning of the satellite. We also observed that the images didn't become distorted in any way. Out of the 39 test cases, 9 failed because the light level was too high. Yet, the brightness in space around Earth is much higher than what we can have here on the surface even at the brightest day [67], not to mention the fact that the

tests were conducted in Finland. Thus, increasing the exposure time or gain value while in orbit will make the images to be too bright and therefore, setting the camera parameters to their default values would possibly give us the best pictures. One should recall that, the automation of these tests was the first time our automation libraries along with Robot Framework were used to properly test Suomi100 satellite and therefore these tests worked as a technology demonstration as well. Figures 26 and 27 show some pictures taken with the NanoCam camera on the Suomi100 satellite.

Most importantly, the tests demonstrated that the integration of the camera subsystem with the rest of the satellite was succesful. As in fact, there was a defect in the integration of the camera with the satellite at the first time the satellite platform arrived to us. The camera lense was too far away from the cell in the PCB of the camera subsystem and the first test pictures taken with the camera were distorted. The manufacturer of the satellite platform provided ~~us with~~ a test picture which showed that the camera was working properly, but it seems that they didn't test the camera after it was integrated to the satellite. After this problem was found, GomSpace provided us with 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 26: Picture taken at Maarintie 8, Espoo, with NanoCam integrated on the Suomi100 satellite. Camera parameters were set to the default values provided by NanoCam manual [58].

Figure 27: 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 to 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 to the satellite platform. As the hardware and software were of our own design and the system integration occurred with a platform manufactured by another organization, these tests took the most time of all automated tests done on Suomi100 satellite. A few sessions were held where we ran these comprehensive test suites for the radio payload and each time new defects in the code and in the integration were found. Of all

the tasks related to Suomi100, the proper integration of the radio payload to the GomSpace 1U NanoEye took the most effort from us.

Test cases for our payload followed the combinatorial testing method in a way where all the the test suites for a given radio operation mode had the same test cases but the frequency used was different. In addition, the test suites for different radio operation modes differed as each had a bit 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 and the figures were added to the Robot Framework HTML log files.

In Figure 28 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 performs several sweeps over the
 ↪  entire frequency range.
  [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 28: Robot Framework test case structure for radio payload testing.

Most interesting information about the payload operation was found from the CSP client replies outputted to the log files. What we were measuring was static and as such, nothing too much could have been said about the test cases just by looking the measurement plots. Beyond the fact that the values were not zero or that there was actual variance in the values measured. In Figure 29 below we have one plot produced from measurement made with the radio while the satellite was at our laboratory at Aalto University and no external radio signal sources were generated by us.

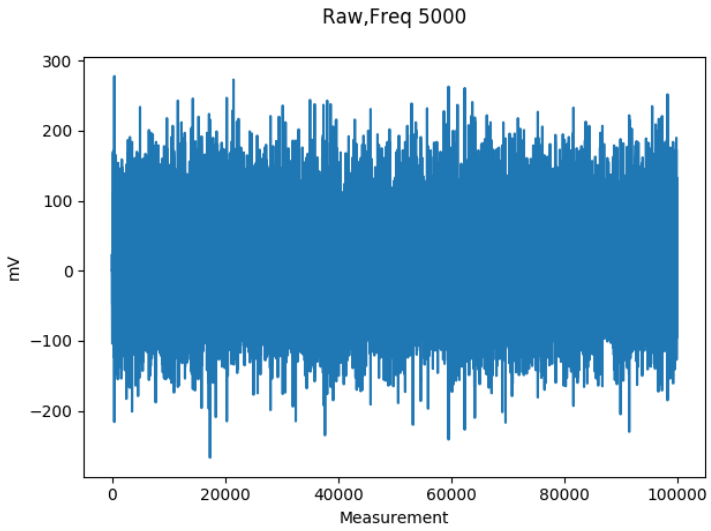


Figure 29: Plot of measurements of radio static made by the radio payload at 5 MHz.

During integration and testing of the radio payload, several problems occurred. Some of these problems can 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 [68] and the processor in the NanoMind OBC in Suomi100 is a 32 MHz processor with a single core [55]. The data from the payload can be read and stored within the 31.25 microseconds with the RaspberryPi 3 in order to have a sample rate of 32 kHz at lowest. But, the reading and storing with NanoMind takes considerably longer, over 60 microseconds. In addition, we need to read the data 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 we are reading the data from one channel at a time. Therefore, with the NanoMind OBC, we can theoretically have a sample rate of 8 MHz.

In addition, it was found that the reading and storing even at this rate is not consistent as the FreeRTOS runs other tasks while we are reading data from the payload. Figure 30 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 happened. In an attempt to counter the inconsistent reading, we first increased the priority of the FreeRTOS task assiocated with the operations of the radio instrument to highest among all of the tasks in NanoMind. This didn't improve the situation though. Thus, we made our task to call for a FreeRTOS *vTaskSuspendAll()* command, which would give all the processing power of the processor for our task and freeze all the other tasks in the OBC. This caused some *watchdog* process unknown to us to reboot the computer, which possibly assumed that the computer was "frozen" and as a safety feature rebooted the computer.

Therefore, as a compromise the sample rate was dropped down to 1 kHz. This

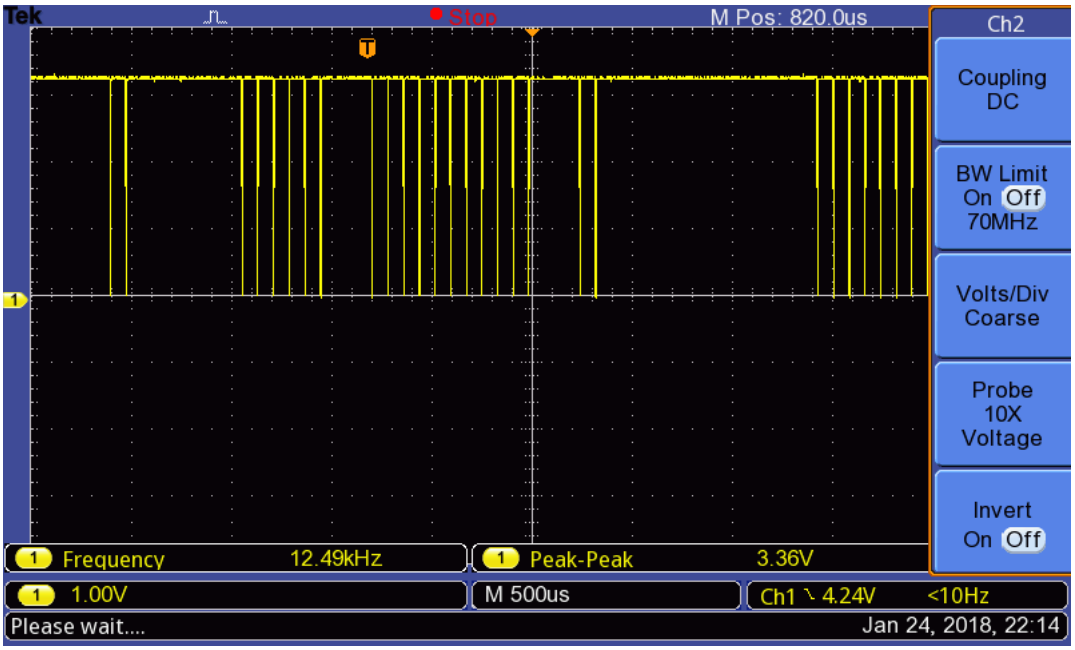


Figure 30: Screenshot from a Tektronix oscilloscope showing inconsistency in data reading rate from the radio payload instrument.

was done by adding a *vTaskDelay(1)* command to our FreeRTOS task after every read and store cycle. This was the shortest time we could add to our task, a shorter wait time would have possibly given a higher sample rate. But, this would have required us to modify the general definitions of the NanoMind code, which could have caused unforeseen consequences to the operation of the satellite. Figure 31 shows that on certain periods we can obtain the data more consistently. Yet, during a longer time period, there still exists longer gaps between data reading events than what we could anticipate. Nonetheless, this data rate was felt to be good enough for our needs. Though, any "real-time" radio noise that could be listened from the radio instrument by human ears is not feasible with this setup.

In addition to the data rate problem, some problems were discovered with 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, we had to write a separate command ourselves which calculates the capacitance value. In the consequent tests with this modification, the values seemed to be correct and it could also be seen from the figures plotted from the measurement data.

Another issue which we faced involved the GPIO pins in the NanoMind. We would have required four of them for our needs, yet only three out of six worked for us. By changing the purpose of these three pins in relation to our payload instrument was sufficient. This made it possible for us to read the data 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 have higher sample rate and with more

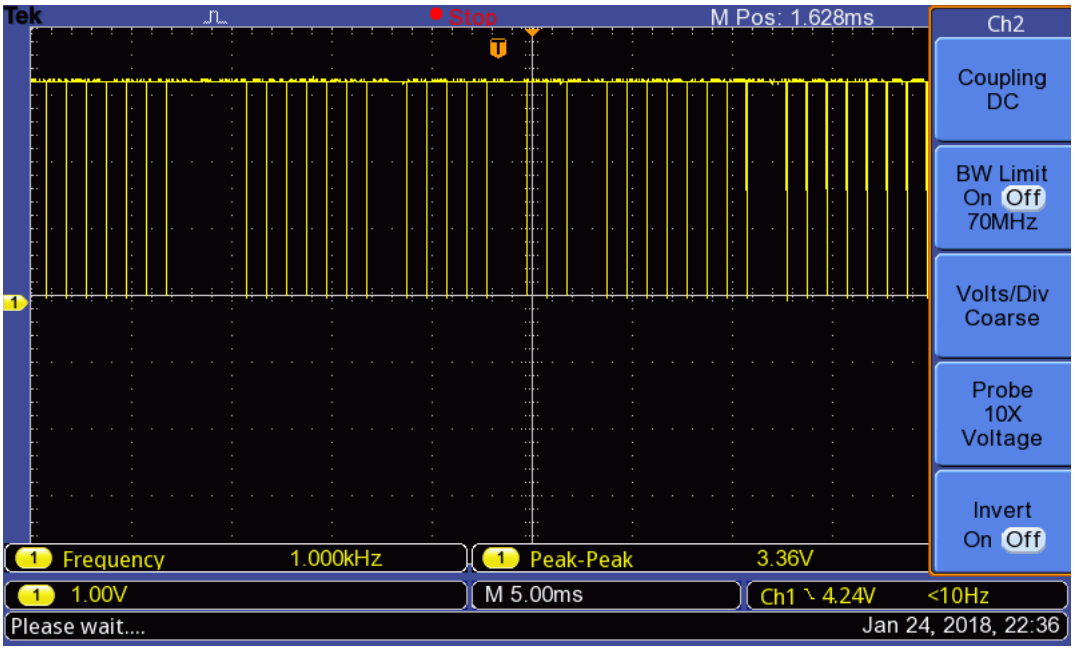


Figure 31: Screenshot from the oscilloscope showing better consistency with lower sample rate in data reading rate from the radio payload instrument.

consistent data reading and storing. As is the case 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, we can gather some valuable information from the ionosphere [66].

4.1.3 Satellite basic operations

Testing of the basic functionalities of the satellite software is necessary in order to obtain a reliable satellite [3, 24, 40]. Therefore, tests were performed for satellite platform features such as safe satellite reboots, 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. No requirements were defined by us for these features, therefore this testing is informal in nature. Nonetheless, execution of these tests was felt to be necessary.

Test suites for these features don't follow the combinatorial testing methods used for two payload subsystems. Instead, test cases are based on different use cases or different situations we might end up facing with the satellite. 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*.

Reboot 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, we can get the NanoMind "frozen" so that it doesn't reboot safely anymore. 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 couldn't 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 led to one test case of the reboot test suite to fail, the test can be seen to fail positively. As such, eleven test cases passed and one failed positively from this test suite. Few Robot Framework test cases are presented in Figure 32.

```
*** 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-COMM
    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 32: Robot Framework test case structure for radio payload testing.

The other types of test cases in the reboot test suite tested reboots during satellite operations and the reboots 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 that the NanoMind can get stuck and in those situations to get the OBC working again, it has required us to either manually reboot the EPS or to send a command via another subsystem to reboot the NanoMind. The satellite

has recovered safely from these situations after reboot. Fortunately, when using the CSP client over the radio link, we have been able to reboot the system. Nonetheless, this situation rose the question whether there can happen something during orbit that causes the satellite to be "frozen" forever.

Fortunately, there are several watchdogs in the satellite that in theory can trigger a reboot after a certain time. Unfortunately, we were not able to deliberately trigger a situation where NanoMind is stuck and thus 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 33.

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 do, which was obviously the preferred result. In Figure ?? we have an example picture of a plot of the system current provided by the beacon at different timestamps.

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 basic functionalities of our satellite, this one had the most failed test cases. For first, it was assumed that giving the commands in wrong format (string instead of an integer) would cause the CSP client to indicate an error. Such a thing didn't happen but giving the commands in wrong format didn't 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 flight plan list. Which happened to be the case when we tried to give one specific command for our radio payload, but this command to run the radio payload in one of the defined operation modes happens to be the single most essential command for the payload. Therefore, in order to make it work with the flight planner we need modify the source code for the flight planner so that it can accept longer strings as commands. Otherwise, we can only use the radio payload when the satellite is in the reach of our communication radios. This is not preferrable as we would be radically limiting the area what we can measure.

Software update tests

Three test cases were performed for the software update feature. As we had to upload a new software to the satellite, these tests took the longest time to execute. These cases tested basic uploading of a new software image, rebooting back to the software

```
*** 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 33: Robot Framework test case structure for radio payload testing.

that was flashed to NanoMind and tested uploading of an invalid file as an image to the new software. The satellite passed all these tests as expected, giving us some confirmation that we actually can safely upload a new software to the satellite and command it to reboot with that software.

In the test case where we uploaded and invalid file as an image, the file itself was just a binary file containing measurement values from the payload radio. Booting with this file caused the NanoMind to crash with EXCEPTION 13 error message. In addition, we had set the NanoMind to try to boot with this file three times and it crashed each time with this error. Eventually as the boot counter reached zero, the satellite managed to recover the proper software it was flashed with. This test gave us knowledge that the satellite manages to recover itself if we happen to upload a software to the satellite that causes unexpected reboots.

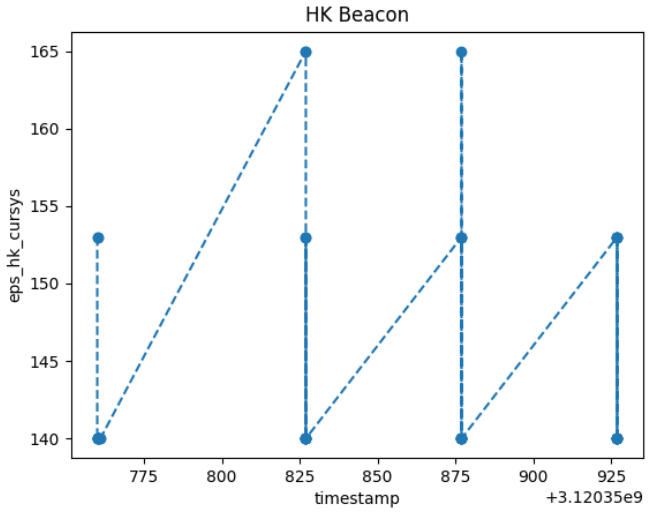


Figure 34: System current from the beacon data during imaging mode operation mode.

4.1.4 "Day in the life" operational scenarios

Test suites were formed to test different phases of the scenarios that the satellite will encounter while in orbit. A scenario would be e.g. where we come from the eclipse, send commands to the satellite and downlink data. Each step in the scenario, like downlinking of data, formed its own test case within the test suite. Four different operational scenarios were tested. These are described as:

1: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. 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 out of the reach of the groudstation.

2: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and 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 groundstation. After the satellite has orbited the Earth, the satellite comes again from the eclipse and is in the reach of the groundstation. Housekeeping data and the taken image is downloaded from the satellite. The satellite goes again out from the sight of the groundstation.

3: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and flight planner is used to set the camera to take pictures continuosly. Flight planner is used to gather housekeeping data continuosly as well. A sudden reboot happens and 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: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and a file is uploaded to the satellite. Finally the satellite goes out of sight of the groundstation.

One test suite for each scenario was written. Different phases of a scenario were made in to different test cases in the suite. Majority of the keywords used in the test suites were the *Persistent Command* and *Verify Reply Contained* keywords. Persistent commanding was required for 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 the electric plug 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, another 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 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 CubSats has been that the solar panels were not properly connected to the power bus [3]. Thus, testing of this was felt to be crucial. Figure 35 shows how the charge in the batteries changed over the execution of the first scenario. All the test cases for scenario 1 were executed succesfully. The housekeeping data and radio measurement were downloaded from the satellite succesfully 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. We were able to again take an image and verify that the solar panels were again charging the satellite. Test cases for scenario 4 were passed as well and a file was uploaded succesfully to the satellite.

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

One reason for the slow download speed was the setting for *rdpopt* command, which defines the wait times between each packets received among other things. With a short wait time the time between received packets is shorter, but the connection

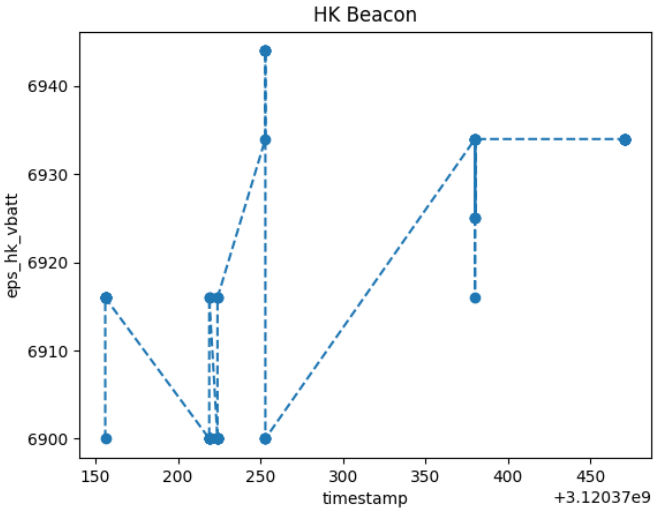


Figure 35: EPS battery charge from the beacon data during the first satellite operational scenario.

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 in our case is $0.9kb/s$. From the test suite logs we could see that at best we could receive data at the speed of $0.3kb/s$. Therefore, some fine tuning of the *rdpcpt* command's parameters would be needed 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. Only the download speed was too slow so that we couldn't download a full picture from the satellite. Besides this, all the test cases passed.

4.2 Release version of CubeSatAutomation test library

The release version of the core function library, *CubeSatAutomation*, was cleared from Suomi100 related dependencies and only the generic process communication keywords are present. Explain some strategy in process communication (only some *Run Command* and *Run Command Persistently* keywords are used and the software knows how to choose between, send, write and type.)

Automation of several Linux terminal programs were tested with the test library succesfully. It was decided that library would be available for everyone who wishes to use test automation with terminal based programs (like some groundstation softwares), thus the library can be found in GitHub. In addition, some example Robot Framework test suites using the CubeSatAutomation library were added to the repository. Any team willing to use the library for test automation with Robot Framework would easily be able to start automating their groundstation software.

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* [7]. These are demanded only for the reason of ensuring safe integration of the P-POD deployer and the CubeSat in to the launch vehicle. The specifications for these tests in fact usually are defined by the launch provider [7].

As can be seen, no testing is required for electrical or functional/operational testing of the satellite. In the research data represented in section 2.1 it was found that failure rates from 40 % to 20 % were prevalent in CubeSat missions [3, 19, 22]. 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 could have been 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 are suggesting at least for some form of guidelines for functional integration testing to be added to the CubeSat project concept. 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 also pointed to the lack of proper integration and testing as a source for mission failures. In addition, when the established testing practices used in NASA were "streamlined", consequent missions showed significant decrease in performance. 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 to the CubeSat project.

4.3.1 Test design

The "Day in the life" operational scenario tests are required for NASA and ESA missions [39]. Besides the mechanical tests mentioned, a test following that principle could be devised for another test that is as a guideline recommended for CubeSat missions. A test known as ***"Day in the life of a CubeSat"***, following the methods described in sections 4.1.4 and 3.3.4. Test like this would test the functionality and proper integration of the satellite. The communication with the groundstation could likewise be verified. In theory, this test could decrease the amount of DOA cases for CubeSat missions. As noted in [3, 23] 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 groundstation. A test like the one discussed here could on ground verify these two aspects of the mission.

In fact, a study done in NASA during 2017 came to similar conclusions about a "Day in the life" testing for CubeSats [69]. A recommendation was set forth for a test like this by operating the satellite via 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 Suomi100 in this thesis and suggested for a "Day in the life of a CubeSat" test by us, in addition to NASA.

One technical solution for a "Day in the life of a CubeSat" test is presented in this thesis. Which includes the integrated satellite, a solar simulator and the groundstation. Some 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 "Day in the life of a CubeSat" is presented in this thesis. This test automation includes Robot Framework and the function libraries developed by us. 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 terminal environment. Therefore, many other groundstation control softwares could possibly be automated, given that they are terminal based. In addition, as CubeSats often fall short in resources and time for testing, the solution to automate the testing could help in this aspect as well. A setup for a "Day in the life of a CubeSat" test is presented in Figure 36.

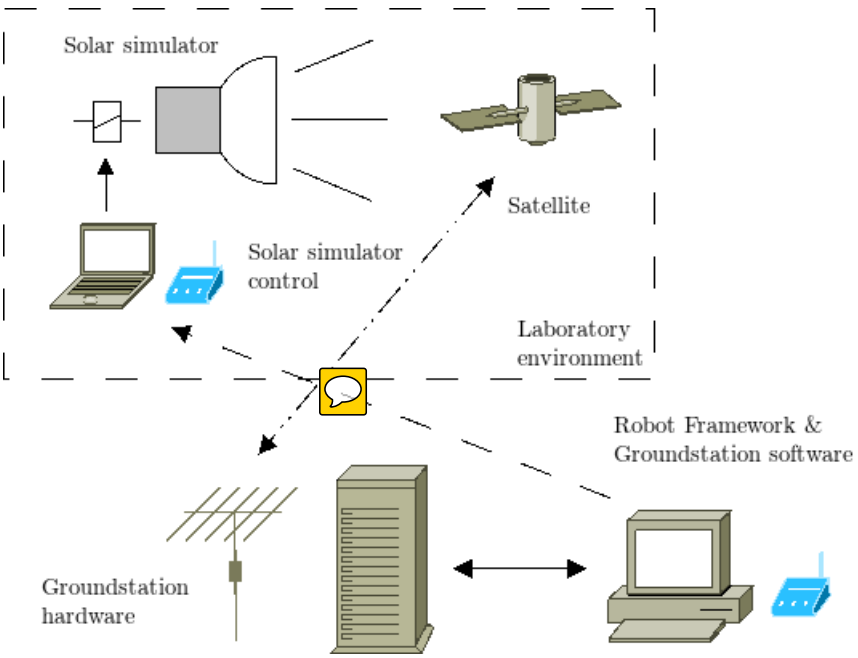


Figure 36: Presentative diagram for a "Day in the life of a CubeSat" test with automated control of groundstation and solar simulator.

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

Therefore, the need for a more detailed requirements and operational documentation was understood from testing of Suomi100. Though, the rigour used by the traditional space missions could be avoided. In the vein of the ideology behind CubeSats, an easily approachable and writable approach towards documentation could be taken.

Definition of different operation modes and assembling different operational scenarios from them could be beneficial for the "Day in the life" testing. Each operation mode could be further described with more detail: *What should the mode do? What it shouldn't do? Which commands 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. In fact, according to [34], modeling of a system is preferred if not required for mission- and safety-critical applications.

In conclusion, a design for "Day in the life of a CubeSat" test could consist of the next 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 as an example like *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.

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.

5 Conclusions

In this thesis automated functional system integration tests for Suomi100 CubeSat were performed with 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 University led missions 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 design of testing performed for Suomi100. The testing was automated with Robot Framework, which is an industry proven acceptance testing framework. It was felt that doing the testing with the help of an automated computer software would give the testing more rigour and reproducability. Function libraries that were used in automation of the testing were written with Python programming language.

First tests conducted for the satellite were functional tests for integrated Suomi100 payloads, which are an optical white light camera and a ~~FM~~ radio instrument for ionospheric measurements. Second test set included testing of the core satellite functions such as housekeeping data collection, safe reboot handling, software updates and so forth. Final set consisted of tests for operational mission scenarios or "day in the life of the satellite"-tests.

~~An attempt was made with our limited resources to~~ simulate the functional environment for testing of the payloads as well as for the operational scenario tests. For testing of the camera payload, the satellite was taken to a balcony at our department 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, we tested not just the proper functionality of the satellite software but that the radio link to the satellite works and that the solar panels can 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 proper functionality of the camera payload as well as proper functionality for almost all of the tests conducted for 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 can charge the batteries in Suomi100. 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 presented several defects and most importantly emergent problems in the integration with the rest of the satellite platform. The defects were fixed with consequent software updates, but certain problems with integration couldn't be overcome. The biggest one being the slow speed of reading and storing of data from the payload by the OBC. This forced us to drop the sample rate from the possible 32-48 kHz to just 1 kHz for reliable data measurement.

Performing these tests proved to us that Robot Framework can work as a testing framework for CubeSats and that we could carry out automated testing of 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 Linux terminal software, such as a satellite ground station control software.

A solution for improving the success rate of CubeSat missions was presented in this thesis as "Day in the life of a CubeSat" test. This 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 tested. As noted, technical solution for automating the control of the satellite via the ground station is provided in this thesis. Therefore, the test desribed here could in part be run automatically.

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

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

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

[8] *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.

[9] 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.

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

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

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

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

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

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

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

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

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

[19] 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.

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

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

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

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

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

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

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

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

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

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

[30] 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.

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

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

[33] 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.

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

[35] *http://www.thespacereview.com/article/1579/1*, accessed 5th of March 2018.

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

[37] 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.

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

[39] 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.

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

[41] 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.

[42] Jacklin, Stephen A., *Survey of Verification and Validation Techniques for Small Satellite Software Development*, Space Tech Expo Conference USA, CA, 2015.

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

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

[45] 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.

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

[47] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[66] 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.

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

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

[69] 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. The code can also be found in Github with an automatic installer. Required libraries are:?

```
import socket
import sys
import os
import signal
import binascii
import math
import subprocess
import thread
import time    # For time stamps
import robot
from time import sleep
from fcntl import fcntl, F_GETFL, F_SETFL
from os import O_NONBLOCK, read
from ConfigParser import SafeConfigParser
import numpy as np
import matplotlib.pyplot as plt

class CubeSatAutomation:
  ''' Implement cleaned version and verify its operation with some linux
  ↪  programs and with csp client.
    Use some QWeb stuff.
  '''

  # Only one instance of the class for all test cases

  ROBOT_LIBRARY_SCOPE = 'TEST_SUITE'
  proc = None
  server = None
  port = 0
  sock = None
  writing = False
  writing_done = False
  reply_buffer = ""
  operation_timer = 0

  def __init__(self):
    self.parser = SafeConfigParser()

  def connect_socket(self, config_file=None, server=None, port=None):
    '''  ADD DOCUMENTATION
    '''

```

```
42      if config_file:
43        self.parser.read(str(config_file))
44      if server is None:
45        self.server = self.parser.get('SOCKET', 'server')
46        print "Read:"
47        print self.server
48      else:
49        self.server = str(server)
50      if port is None:
51        self.port = self.parser.get('SOCKET', 'port')
52        print "Read:"
53        print self.port
54      else:
55        self.port = str(port)
56      #self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
57      CubeSatAutomation.sock = socket.socket(socket.AF_INET,
        ↪  socket.SOCK_STREAM)
58      server_address = (str(self.server), int(self.port))
59      print 'connecting to %s port %s' % server_address
60      CubeSatAutomation.sock.connect(server_address)
61      print CubeSatAutomation.sock
62
63    # Kaikki None aikasemmin ja esim if config_file is None:'
64    #    self.parser.read('s100_config.cfg')
65    #  else:
66    #    self.parser.read(str(config_file))'
67    def client_start(self, config_file=None, prog=None, params=None):
68      ''' ADD DOCUMENTATION
69      '''
70
71      if "None" in str(config_file):
72        print "None in config file"
73        pass
74      else:
75        self.parser.read(str(config_file))
76      if prog is None:
77        prog = self.parser.get('PROGRAM', 'path')
78        print "Read:"
79        print prog
80      elif "None" in str(prog):
81        prog = ""
82      else:
83        print "Prog" is str(prog)
84        prog = str(prog)
85      if params is None:
86        params = self.parser.get('PROGRAM', 'params')
87        print "Read:"
88        print params
```

```
89      elif "None" in str(params):
90         params = ""
91      else:
92        params = str(params)
93      CubeSatAutomation.proc = subprocess.Popen([str(prog) + " " +
         ↪  str(params)], stdin=subprocess.PIPE,
94        stdout=subprocess.PIPE, stderr=subprocess.STDOUT, shell=True)
95      flags = fcntl(CubeSatAutomation.proc.stdout, F_GETFL) # get current
         ↪  p.stdout flags
96      print "fcntl flags:"
97      print flags
98      fcntl(CubeSatAutomation.proc.stdout, F_SETFL, flags | O_NONBLOCK)
99      print "Started process " + str(self.proc) + " with parameters " +
         ↪  str(params)
100
101   def client_close(self, prog, socket_comm=None):
102     if socket_comm:
103       CubeSatAutomation.sock.shutdown(socket.SHUT_RDWR)
104       CubeSatAutomation.sock.close()      #CubesatAutomation.sock to be
           ↪  sure?
105     if prog:
106           subprocess.Popen([str(kill_command)], shell=True)
107     else:
108       raise OSError ("Program to be closed must be defined!")
109
110   def monitor_start(self):
111     ''' Start a terminal window and send the keywords, commands and
         ↪  replies to there.
112     '''
113     pass
114
115   def send_message(self, message):
116     print "Sending message %s to socket" % message
117     message = message + "\r"
118     try:
119       CubeSatAutomation.sock.sendall(message)
120     except:
121       print "Failed to send message to socket, trying to write message to
           ↪  stdin" % message
122       try:
123         CubeSatAutomation.proc.stdin.write(command)
124       except:
125         print "Failed to write message to stdin, trying to simulate
           ↪  keyboard pressing" % message
126         import pyautogui
127         pyautogui.typewrite(str(message))
128         pyautogui.press('enter')
129     sleep(1)
```

```
130
131   def send_command(self, message, option="Store", timeout=5,
      ↪  read_timeout=5):
132     self.send_message(str(message))
133     console_lines = self.read_console_reply(int(timeout),
        ↪  int(read_timeout))
134     console_lines = str(console_lines).split("\\n")
135     if "Store" in str(option):
136       CubeSatAutomation.reply_buffer = console_lines
137       print "Console lines"
138       print CubeSatAutomation.reply_buffer
139     for line in console_lines:
140       if "error -1" in line or "error -2" in line:
141         raise ValueError("%s" % line)
142
143
144   def write_command(self, message, option="Store", timeout=2,
      ↪  read_timeout=2):
145     command = str(message) + '\r'
146     CubeSatAutomation.proc.stdin.write(command)
147     console_lines = self.read_console_reply(int(timeout),
        ↪  int(read_timeout))
148     console_lines = str(console_lines).split("\\n")
149     if "Store" in str(option):
150       CubeSatAutomation.reply_buffer = console_lines
151       print "Console lines"
152       print CubeSatAutomation.reply_buffer
153
154   def type_command(self, message, option="Store", timeout=2,
      ↪  read_timeout=2):
155     '''  pyautogui is needed. Some solution without it?
156     '''
157     import pyautogui
158     pyautogui.typewrite(str(message))
159     pyautogui.press('enter')
160     console_lines = self.read_console_reply(int(timeout),
        ↪  int(read_timeout))
161     console_lines = str(console_lines).split("\\n")
162     if "Store" in str(option):
163       CubeSatAutomation.reply_buffer = console_lines
164       print "Console lines"
165       print CubeSatAutomation.reply_buffer
166
167   def read_console_reply(self, timeout, read_timeout=10):
168     console_lines = []
169     time_count = 0
170     while time_count < int(timeout):
171       sleep(1)  # Wait for data to be 'cooked'
```

```
172         time_count = time_count + 1
173         try:
174           line = read(CubeSatAutomation.proc.stdout.fileno(), 1024)
175         except OSError:  # No data to be read, wait if more comes
176           console_lines.append("Waiting for more data from process..\n")
177           read_timecount = 0
178           while read_timecount < int(read_timeout):
179             try:
180               line = read(CubeSatAutomation.proc.stdout.fileno(), 1024)
181             except OSError:
182               sleep(1)
183               read_timecount = read_timecount + 1
184               continue
185             else:
186               break
187           if read_timecount >= int(read_timeout):
188             console_lines.append("Process data read timeout!\n")
189             break
190         print "term:" + line.rstrip()
191         if 'exit_client' in line:
192           os.killpg(os.getpgid(CubeSatAutomation.proc.pid), signal.SIGTERM)
193           break
194         else:
195           if line != '':
196             console_lines.append(line)
197       console_lines_str = console_lines
198       console_lines = ''.join(console_lines_str)
199       console_lines = str(console_lines).split("\\n")  # Why doesn't return
          ↪  this? Because of the join?
200       print console_lines
201       return console_lines
202
203     def clear_replies(self, option="None", read_timeout=5):
204       ''' Clear process replies with flush command
205       '''
206       CubeSatAutomation.proc.stdout.flush()
207       if "Stored" in str(option) or "All" in str(option):
208         CubeSatAutomation.reply_buffer = ""
209       if "Stored" not in str(option):
210         try:
211           read(CubeSatAutomation.proc.stdout.fileno(), 1024)
212         except OSError:  # No data to be read, wait if more comes
213           read_timecount = 0
214           while read_timecount < int(read_timeout):
215             try:
216               read(CubeSatAutomation.proc.stdout.fileno(), 1024)
217             except OSError:
218               sleep(1)
```

```
219             read_timecount = read_timecount + 1
220             continue
221           else:
222             break
223
224
225   def store_client_responses_thread(self, filename, timeout,
      ↪  read_timeout):
226     """  Write the responses to a file.
227          Thus we can keep track of what we have done with the satellite.
228          Various verify_result keywords parse the responses and fail the
    ↪  cases accordingly.
229          Use the Queue module to send information to thread(s) that it is
    ↪  okay to read now.
230          Or some threading. Queue as a class variable? Or just with a
    ↪  class variable?
231          Close this if after few loops we get nothing.
232     """
233     #CubeSatAutomation.read_queue = Queue.Queue()
234     time_count = 0
235     while time_count < int(timeout):
236       if CubeSatAutomation.writing is False:
237         try:
238           f = open(str(filename), 'a')  # Creates a new file if the old
            ↪  one was moved already
239         except IOError:
240           print "Storing couldn't open %s" % str(filename)
241           raise IOError ("Couldn't open %s" % str(filename))
242         console_lines = []
243         console_lines = self.read_console_reply(10, int(read_timeout))
244         time_count = time_count + int(read_timeout)
245         f.writelines(console_lines)
246         f.close()
247         CubeSatAutomation.writing = True  # Lock the file, implement a
          ↪  proper lock
248       else:
249         sleep(1)
250         time_count = time_count + 1
251       if CubeSatAutomation.writing_done is True:
252         CubeSatAutomation.writing = False
253         CubeSatAutomation.writing_done = False
254         break
255     # SVA style solution, write some end of test indicator to the file and
       ↪  the verifying keyword polls for that?
256     # Add time to the filename
257       #sys.exit()
258     # Write beginning time and end time to the file
259
```

```
260   def store_client_responses(self, filename, timeout=5, read_timeout=20):
261     thread.start_new_thread(self.store_client_responses_thread, (filename,
        ↪  timeout, read_timeout))
262
263   def verify_reply_message(self, message, timeout=5, read_timeout=10):
264     ''' Read messages from the process stdout and verify if the desired
        ↪  text exists.
265     '''
266     console_lines = self.read_console_reply(int(timeout),
        ↪  int(read_timeout))
267     console_lines = str(console_lines).split("\\n")
268     found = False
269     for line in console_lines:
270       if str(message) in line:
271         found = True
272         break
273     if not found:
274       print console_lines
275       raise ValueError ("Message %s was not found in the process
          ↪  replies!\n" % str(message))
276
277   def verify_reply_contains(self, message, timeout=5, read_timeout=10):
278     self.verify_reply_message(message, timeout, read_timeout)
279
280   def verify_reply_contains_not(self, message, timeout=5,
      ↪  read_timeout=10):
281     console_lines = self.read_console_reply(int(timeout),
        ↪  int(read_timeout))
282     console_lines = str(console_lines).split("\\n")
283     found = False
284     for line in console_lines:
285       if str(message) in line:
286         found = True
287         break
288     if found:
289       print console_lines
290       raise ValueError ("Message %s was not supposed to be found in the
          ↪  process replies!\n" % str(message))
291
292   def verify_reply_contained(self, message):
293     console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
294     found = False
295     for line in console_lines:
296       if str(message) in line:
297         found = True
298         break
299     if not found:
300       print console_lines
```

```
301         raise ValueError ("Message %s was not found in the recent process
            ↪  replies!\n" % str(message))
302
303     def verify_reply_contained_not(self, message):
304       console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
305       found = False
306       for line in console_lines:
307         if str(message) in line:
308           found = True
309           break
310       if found:
311         print console_lines
312         raise ValueError ("Message %s was not supposed to be found in the
            ↪  recent process replies!\n" % str(message))
313
314     def wait_until_reply_contains(self, message, timeout=20,
         ↪  read_timeout=5):
315       completed = False
316       found = False
317       time_count = 0
318       for line in CubeSatAutomation.reply_buffer:
319         if str(message) in str(line):
320           completed = True
321           found = True
322       while not completed:
323         console_lines = self.read_console_reply(1, int(read_timeout))
324         console_lines = str(console_lines).split("\\n")
325         time_count = time_count + 1
326         sleep(1)
327         if time_count > int(timeout):
328           completed = True
329         for line in console_lines:
330           if str(message) in line:
331             found = True
332             completed = 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 wait_until_reply_contains_not(self, message, timeout=20,
         ↪  read_timeout=5):
339       completed = False
340       time_count = 0
341       while not completed:
342         console_lines = self.read_console_reply(1, 10)
343         console_lines = str(console_lines).split("\\n")
```

```
344         time_count = time_count + 1
345         print time_count
346         sleep(1)
347         if time_count > int(timeout):
348           completed = True
349         print console_lines
350         completed = True
351         for line in console_lines:
352           if str(message) in line:
353             print "found, not finished"
354             completed = False
355             break
356       if time_count > int(timeout):
357         print console_lines
358         raise ValueError ("Message %s was found in the process replies until
          ↪  timeout!\n" % str(message))
359
360     def verify_stored_reply_message(self, message, filename, timeout=30):
361       completed = False
362       time_count = 0
363       while not completed:
364         time_count = time_count + 1
365         sleep(1)
366         if time_count > int(timeout):
367           print "Time count is larger? " + str(time_count) + " " +
            ↪  str(timeout)
368           CubeSatAutomation.writing = False
369           CubeSatAutomation.writing_done = True
370           completed = True
371           break
372         if CubeSatAutomation.writing is True:
373           try:
374             f = open(str(filename), 'r')
375           except IOError:
376             raise IOError ("Couldn't open %s" % str(filename))
377           console_lines = f.readlines()
378           for line in console_lines:
379             if str(message) in line:
380               f.close()
381               CubeSatAutomation.writing = False
382               CubeSatAutomation.writing_done = True
383               completed = True
384               break
385           f.close()
386           CubeSatAutomation.writing = False  # Free the file
387           #CubeSatAutomation.writing_done = True
388         else:
389           continue
```

```
390       # Move the file storing to another keyword, but we want to store the
          ↪  failed ones as well
391       if completed:
392         try:
393           f = open(str(filename), 'r')
394         except IOError:
395           print "Verifying couldn't open %s\n" % str(filename)
396         console_lines = f.readlines()
397       f.close()
398       # Rename and move file
399       new_filename = str(os.getcwd()) + "/satellite_passes/"  +
          ↪  str(filename) + "_" + str(time.time())  # Unix time
400       os.rename(str(filename), new_filename)
401       if time_count > int(timeout):
402         raise ValueError("Message %s was not found in the stored process
            ↪  replies!\n" % str(message))
403
404
405     def wait_and_notify(self, event, timeout, soundfile):
406       time_count = 0
407       while True:
408         if time_count >= int(timeout):
409           break
410         time_count = time_count + 1
411         time.sleep(1)
412       # Play some sound signal when time is reached
413       cwd = os.getcwd()
414       path = str(cwd) + str(soundfile)
415       from playsound import playsound
416       playsound(path)
417
418     def notify_after_thread(self, event, timeout, soundfile):
419       time_count = 0
420       while True:
421         if time_count >= int(timeout):
422           break
423         time_count = time_count + 1
424         CubeSatAutomation.operation_timer =
            ↪  CubeSatAutomation.operation_timer + 1
425         time.sleep(1)
426       # Play some sound signal when time is reached
427       cwd = os.getcwd()
428       path = str(cwd) + str(soundfile)
429       from playsound import playsound
430       playsound(path)
431
432     def notify_after(self, event, timeout, soundfile):
```

```
433       thread.start_new_thread(self.notify_after_thread, (event, timeout,
          ↪  soundfile))
434
435     def wait_until_time_event(self, event, timeout):
436       while True:
437         if CubeSatAutomation.operation_timer >= int(timeout):
438           print "Timed event %s reached" % str(event)
439           CubeSatAutomation.operation_timer = 0
440           time.sleep(20)    # Average time for the notification sound to
              ↪  play
441           break
442
443     def persistent_command(self, message, exception_replies,
        ↪  end_reply="None", timeout=5, read_timeout=2):
444       ''' Sends a command persistently until either time runs out or a
          ↪  certain reply is received
445        '''
446       time_count = 0
447       completed = False
448       found = False
449       error_found = False
450       command = str(message) + '\r'
451       CubeSatAutomation.proc.stdin.write(command)
452       exception_replies = str(exception_replies)
453       exception_replies = exception_replies.split(';')
454       print exception_replies
455       while not completed:
456         if time_count >= int(timeout):
457           completed = True
458           break
459         console_lines = self.read_console_reply(10, int(read_timeout))
460         console_lines = str(console_lines).split("\\n")
461         CubeSatAutomation.reply_buffer = console_lines
462         for line in console_lines:
463           for exception_reply in exception_replies:
464             print "exception_reply:" + str(exception_reply)
465             print "reply line:" + str(line)
466             if str(exception_reply) in str(line):
467               print "Exception %s found, retrying to send command" %
                  ↪  str(exception_reply)
468               #error_found = True
469               CubeSatAutomation.proc.stdin.write(command)
470               break
471           # if len(end_reply) > 1:
472           if str(end_reply) in str(line):
473             completed = True
474             found = True
475             break
```

```
476           if "None" in str(end_reply):
477             completed = True
478             found = True
479             break
480         time_count = time_count + 1
481         time.sleep(1)
482       if len(end_reply) > 1:
483         if found:
484           print "Desired reply %s was found in process replies" %
              ↪  str(end_reply)
485         else:
486           if str(end_reply) == "Timeout":
487             pass
488           else:
489             raise ValueError ("Desired reply %s was not found in process
              ↪  replies" % str(end_reply))
490       if "None" in str(end_reply) or "Timeout" in str(end_reply):
491         console_lines = self.read_console_reply(1, int(read_timeout))
492         console_lines = str(console_lines).split("\\n")
493         CubeSatAutomation.reply_buffer = console_lines
494         for line in console_lines:
495             print "exception_reply:" + str(exception_reply)
496             print "reply line:" + str(line)
497             if str(exception_reply) in str(line):
498               raise ValueError ("Exception %s still found after timeout" %
                  ↪  str(exception_reply))
```

B Socket API for CSP client

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

```
#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(sockfd, buffer, strlen(buffer)); --> broken pipe
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 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 like the *Satellite State* keyword.

```
1  #--------------s100_keywords.robot-------------
2
3  *** Keywords ***
4  Start Suite
5    Client Start
6    Sleep     5
7    Connect Socket
8
9  End Suite
10   Send Message    exit_client
11   Close Connection
12   Client Close
13
14 Satellite State
15   [Arguments] ${state}
16   ${state}= Convert To Lowercase  ${state}
17   Run Keyword If  '${state}' == 'idle'
18   ... 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 ***
Library   String
Library   ../CubeSatAutomation.py
Library   ../NanoCam.py
Resource  ../s100_keywords.robot
Suite Setup   Start Suite
Suite Teardown  Client Close

*** Test Cases ***

Imaging mode - Default parameters
  [Documentation]   The onboard camera is used to take images of the
 ↪  earth.
  [Tags]        OPMODE-IMAGING
  Satellite State   Reboot
  Camera Startup    15
  Verify Startup    Camera
  Verify Device Detected    Camera  5
  Set Satellite Parameter   Camera  exposure-us   10000
  Set Satellite Parameter   Camera  gain-target   30
  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  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
67    [Documentation]    The payload radio performs several sweeps over the
    ↪  entire frequency range.
68    [Tags]         OPMODE-LOWOBS
69    Satellite State    Reboot
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;0;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_debug1.dat
83    Radio Plot Data    m2_debug1.dat          m2_debug1.txt   m2_debug1.png
84    Log    <img src="m2_debug1.png" width="640" height="480">   html=yes
85
86  Lowobs Mode - 5 Mhz Default parameters, output_type 3
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;100;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_debug5.dat
103   Radio Plot Data   m2_debug5.dat   m2_debug5.txt   m2_debug5.png
104   Log   <img src="m2_debug5.png" width="640" height="480">  html=yes
105
106 Lowobs Mode - 5 Mhz 100 times, 10 points in average
107   [Documentation]   The payload radio performs several sweeps over the
      ↪  entire frequency range.
108   [Tags]        OPMODE-LOWOBS
109   Satellite State   Idle
110   Radio Startup   3 0 1
111   Verify Startup    Radio
112   Verify Device Detected  Radio   5
113   Verify Radio Status
114   Store Client Responses  Lowobs Mode   80    15
115   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  2
      ↪  0;5000;100;10;100;3;0;
116   Sleep     2
117   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
118   Send Beacon   10  4  1
119   Sleep     10
120   Verify Radio Results  Lowobs Mode 80
121   Radio Power Down
122   Radio Load Data   /flash/data/m2_debug.dat  m2_debug8.dat
123   Radio Plot Data   m2_debug8.dat   m2_debug8.txt m2_debug8.png
124   Log   <img src="m2_debug8.png" width="640" height="480">  html=yes
125
126 #--------------payload_tests_targetmode.robot-------------
127
128 *** Settings ***
129 Library   String
130 Library   ../libraries/CubeSatAutomation.py
131 Library   ../libraries/RadioPayload.py
132 Resource  ../resources/s100_keywords.robot
```

```
133 Suite Setup   Start Suite
134 Suite Teardown  Client Close
135
136 *** Test Cases ***
137
138 Target Mode - Default parameters
139   [Documentation]   The payload radio performs several sweeps over the
    ↪  entire frequency range.
140   [Tags]        OPMODE-TARGET
141   Satellite State   Reboot
142   Radio Startup   3 0 1
143   Verify Startup    Radio
144   Verify Device Detected  Radio   5
145   Verify Radio Status
146   Store Client Responses  Target Mode   1420    15
147   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  3
    ↪  0;1000;10000;10;10;1000;100;0;0;
148   Sleep     2
149   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
150   Send Beacon   10  4  1
151   Sleep     10
152   Verify Radio Results  Target Mode   1420
153   Sleep         20
154   Radio Power Down
155   Radio Load Data   /flash/data/m3_debug.dat  m3_debug1.dat
156   Radio Plot Data   m3_debug1.dat         m3_debug1.txt  m3_debug1.png
157   Log     <img src="m3_debug1.png" width="640" height="480">  html=yes
158
159 Target Mode - Other antenna
160   [Documentation]   The payload radio performs several sweeps over the
    ↪  entire frequency range.
161   [Tags]        OPMODE-TARGET
162   Satellite State   Idle
163   Radio Startup   3 0 0
164   Verify Startup    Radio
165   Verify Device Detected  Radio   5
166   Verify Radio Status
167   Store Client Responses  Target Mode   1420    15
168   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  3
    ↪  0;1000;10000;10;10;1000;100;0;0;
169   Sleep     2
170   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
171   Send Beacon   10  4  1
172   Sleep     10
173   Verify Radio Results  Target Mode   1420
174   Sleep         20
175   Radio Power Down
176   Radio Load Data   /flash/data/m3_debug.dat  m3_debug4.dat
```

```
177   Radio Plot Data   m3_debug4.dat         m3_debug4.txt  m3_debug4.png
178   Log   <img src="m3_debug4.png" width="640" height="480">  html=yes
179
180
181 Target Mode - N_ave 1000
182   [Documentation]   The payload radio performs several sweeps over the
      ↪  entire frequency range.
183   [Tags]        OPMODE-TARGET
184   Satellite State   Idle
185   Radio Startup   3 0 1
186   Verify Startup    Radio
187   Verify Device Detected  Radio   5
188   Verify Radio Status
189   Store Client Responses  Target Mode   1920    15
190   Run Radio Mode      /flash/radio_params.cfg  /flash/radio_props.cfg  3
      ↪  0;1000;10000;10;10;1000;1000;3;0;
191   Sleep     2
192   Get HK      30  2  1  1  5  2  /flash/hk_test_lom
193   Send Beacon   10  4  1
194   Sleep     10
195   Verify Radio Results  Target Mode   1920
196   Sleep         20
197   Radio Power Down
198   Radio Load Data   /flash/data/m3_debug.dat  m3_debug6.dat    260
199   Radio Plot Data   m3_debug6.dat         m3_debug6.txt  m3_debug6.png
200   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-COMM
  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-COMM
  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-COMM
  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-COMM
  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-COMM
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 one test suite for a "Day in the life of a satellite" presented.

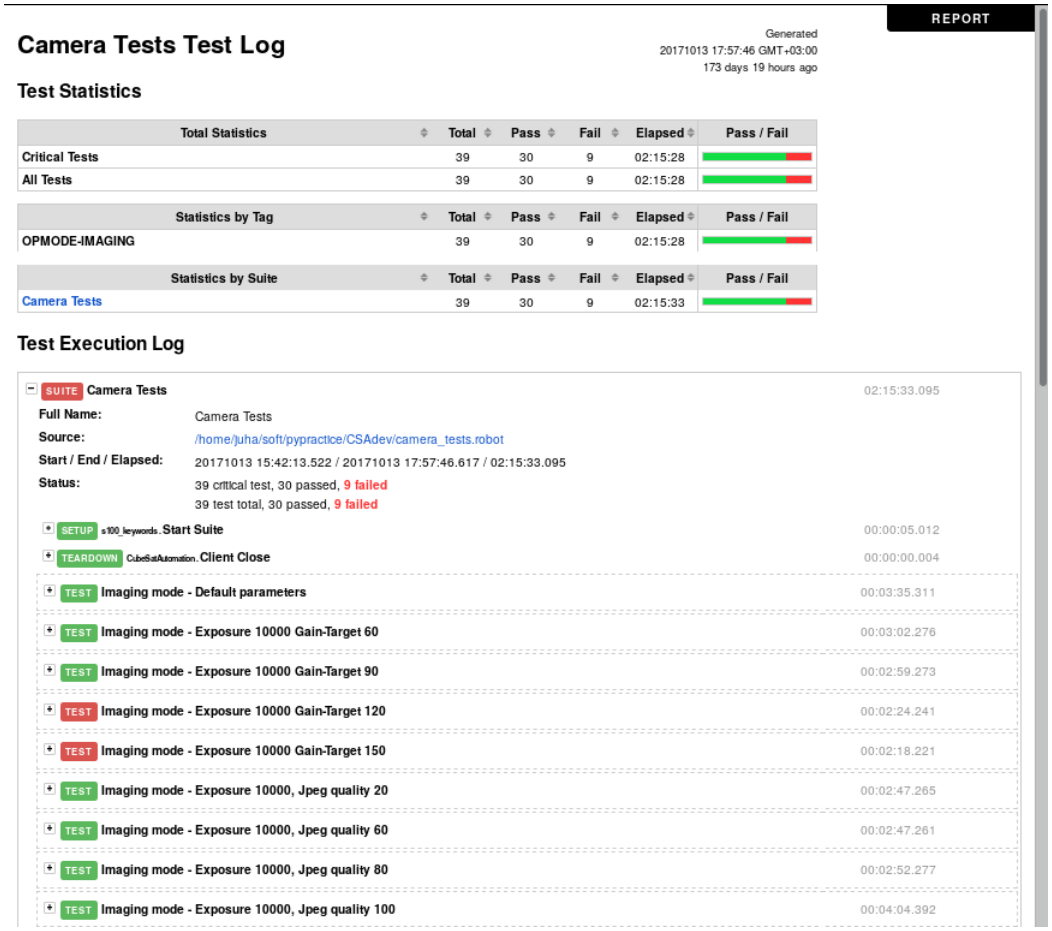
```
#--------------dayinthelife1_mod.robot-------------

*** 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-COMM
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 Logs

The test execution log files for majority of the test suites are presented in this section. Firstly, the log files for tests of the camera payload are shown. The log files for tests



REPORT

Camera Tests Test Log

Generated
20171013 17:57:46 GMT+03:00
173 days 19 hours ago

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

Figure D1: Robot Framework HTML log file for tests performed for the camera payload.

performed for the radio payload are shown below.

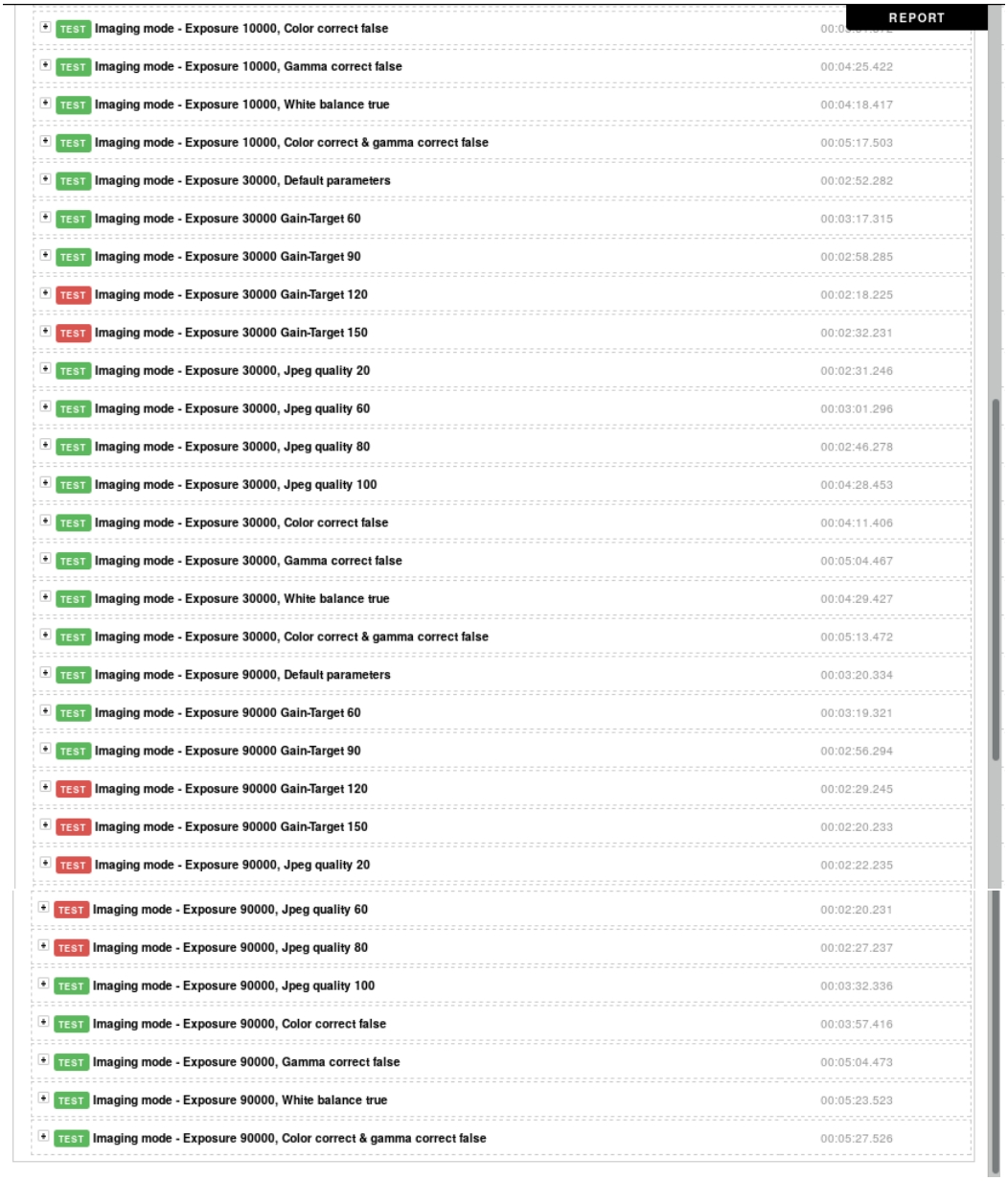


Figure D2: Robot Framework HTML log file for tests performed for the camera payload.

REPORT

Payload Tests Rawmode 5 Test Log

Generated
20180102 13:37:03 GMT+02:00
92 days 22 hours ago

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	

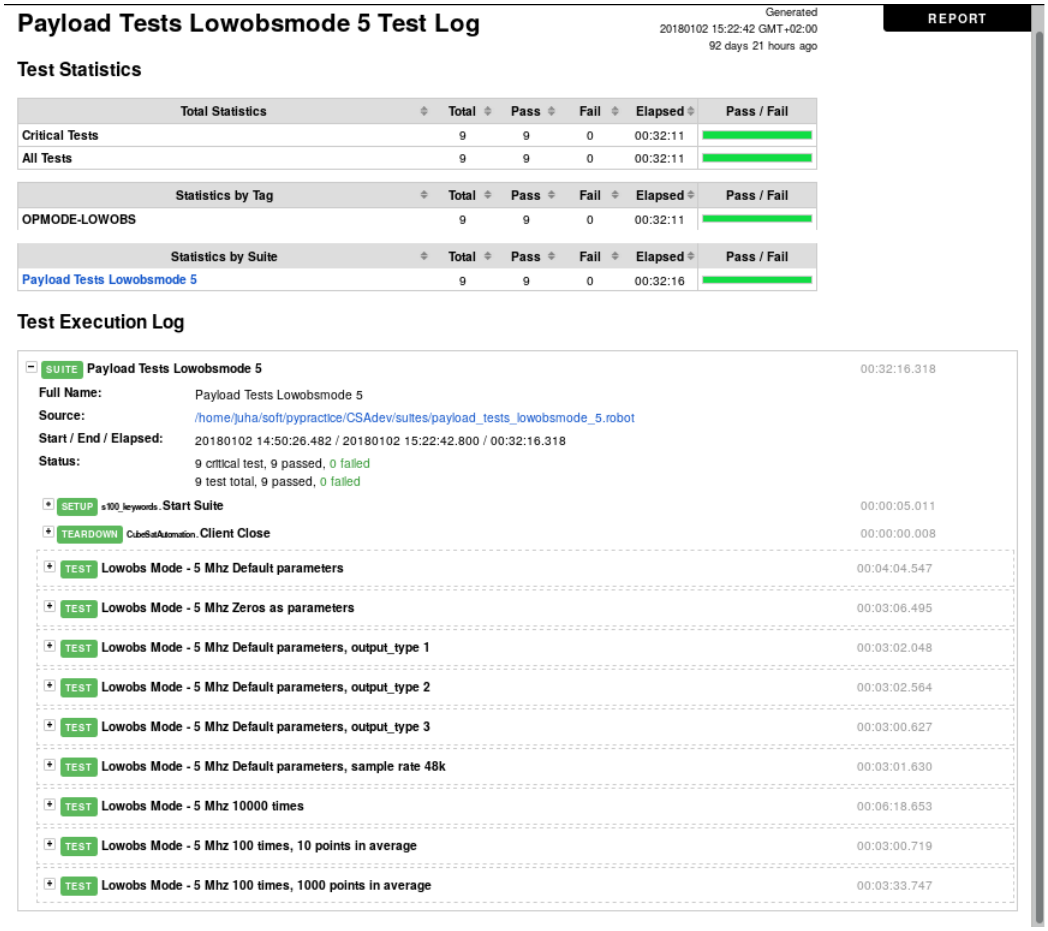
Test Execution Log

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

Figure D3: Test execution log file for radio payload "Raw data mode".



Payload Tests Lowobsmode 5 Test Log

Generated
20180102 15:22:42 GMT+02:00
92 days 21 hours ago

REPORT

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	

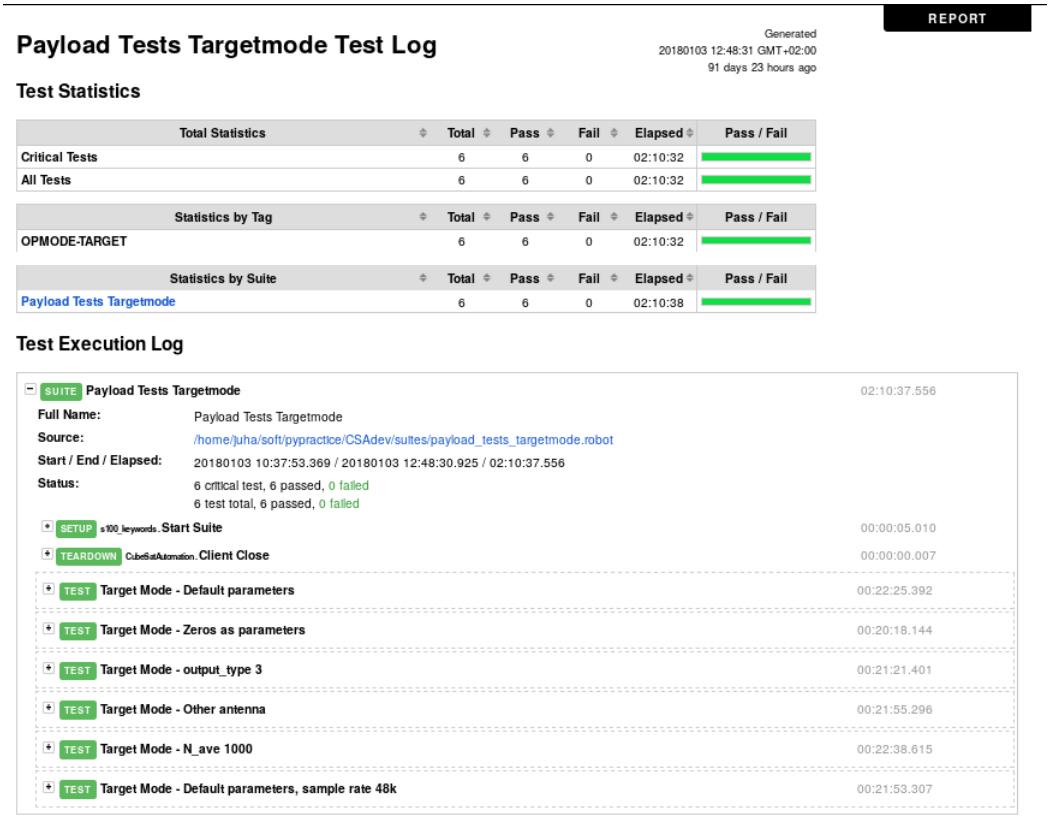
Test Execution Log

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

Figure D4: Test execution log file for radio payload "Average raw data mode".



REPORT

Payload Tests Targetmode Test Log

Generated
20180103 12:48:31 GMT+02:00
91 days 23 hours ago

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/luha/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

Figure D5: Test execution log file for radio payload "Ramped average data mode".

REPORT

Flight Planner Tests Test Log

Generated
20171127 11:23:37 GMT+02:00
129 days 1 hour ago

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	

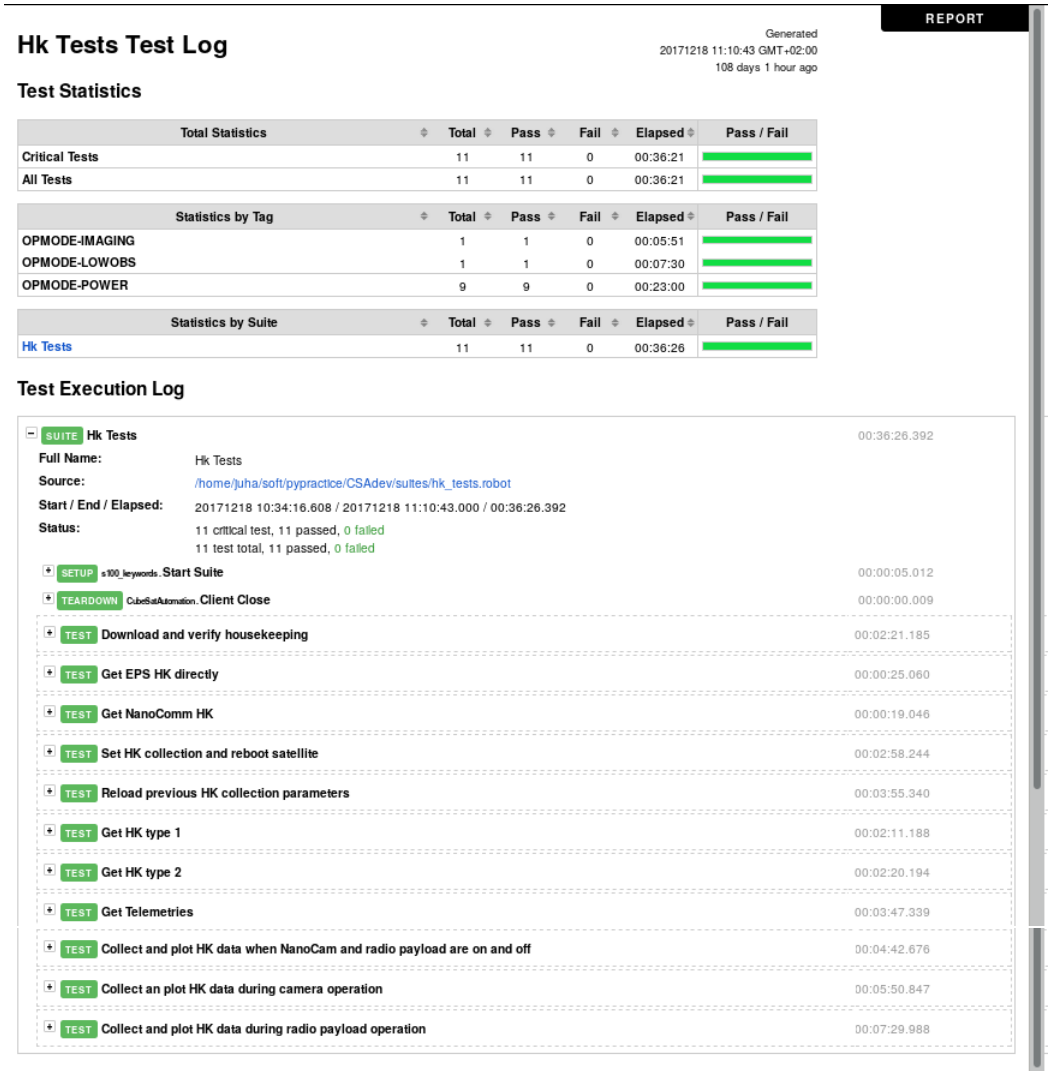
Test Execution Log

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

Figure D6: Test results from testing of GomSpace flight planner.



REPORT

Hk Tests Test Log

Generated
20171218 11:10:43 GMT+02:00
108 days 1 hour ago

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	

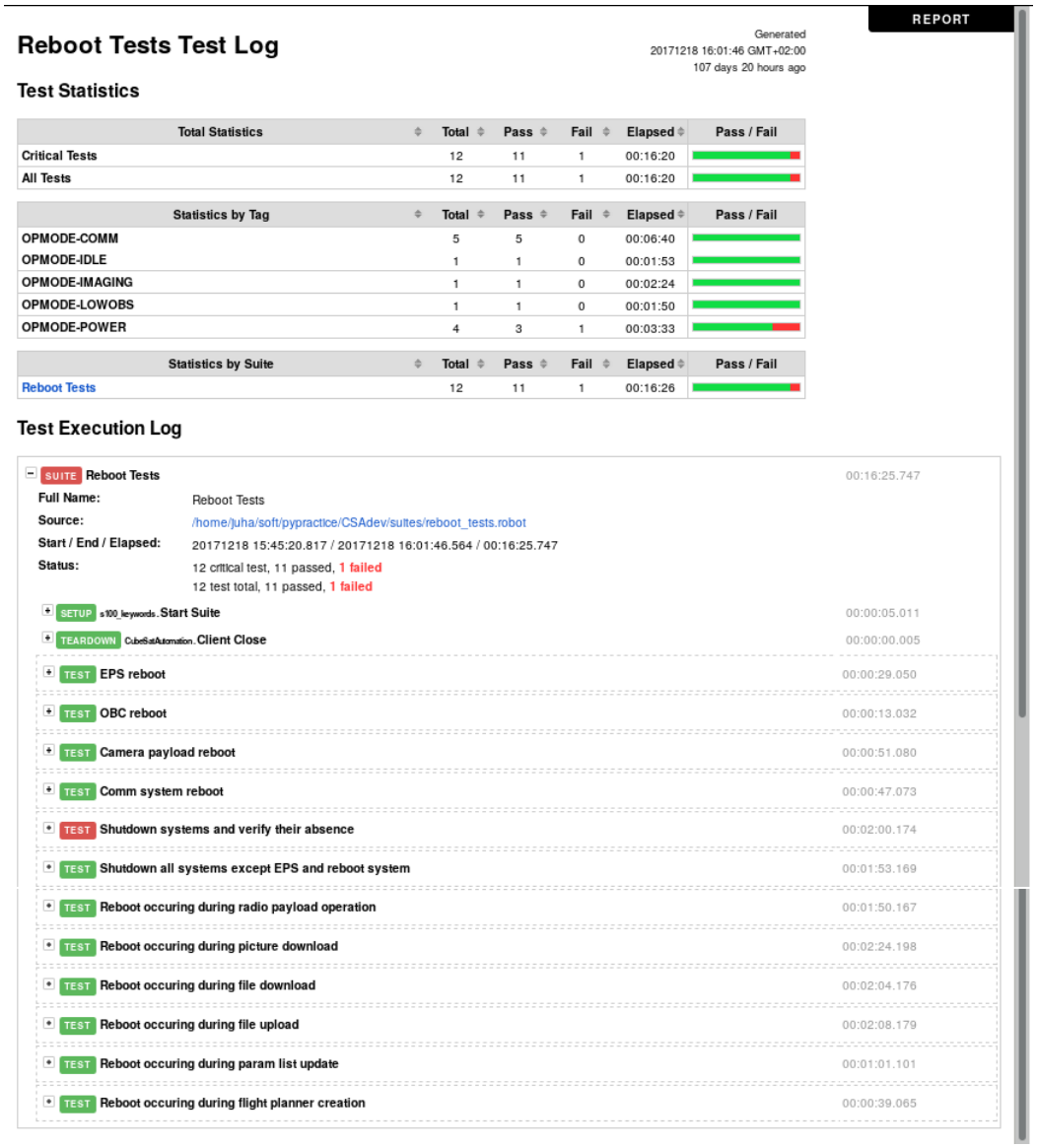
Test Execution Log

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

Figure D7: Test results from testing of GomSpace House Keeping features.



REPORT

Reboot Tests Test Log

Generated
20171218 16:01:46 GMT+02:00
107 days 20 hours ago

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	

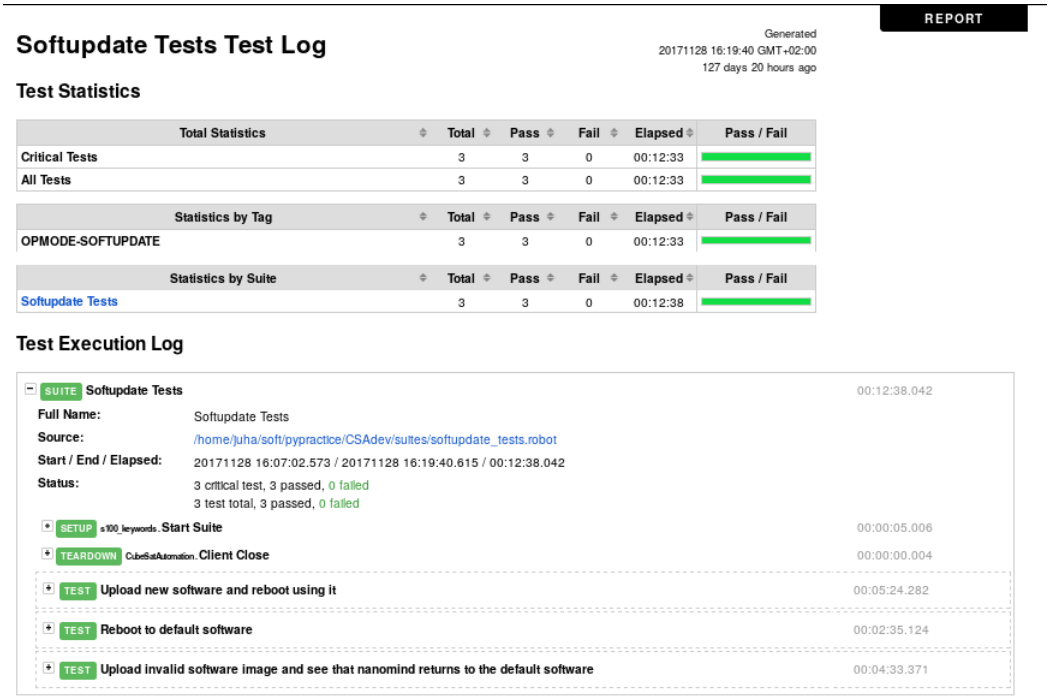
Test Execution Log

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

Figure D8: Test results from testing of reboot and recovery features of GomSpace software.



REPORT

Softupdate Tests Test Log

Generated
20171128 16:19:40 GMT+02:00
127 days 20 hours ago

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	

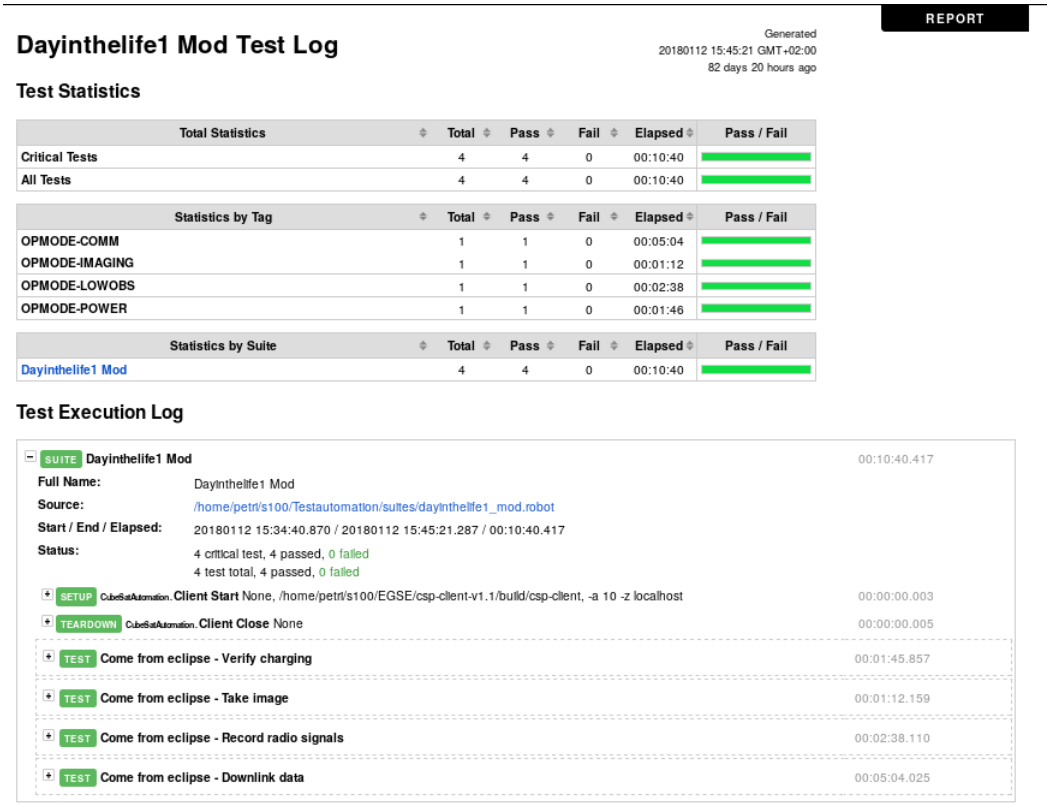
Test Execution Log

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

Figure D9: Log file from testing of software update features of the GomSpace platform.



REPORT

Dayinthelife1 Mod Test Log

Generated
20180112 15:45:21 GMT+02:00
82 days 20 hours ago

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	

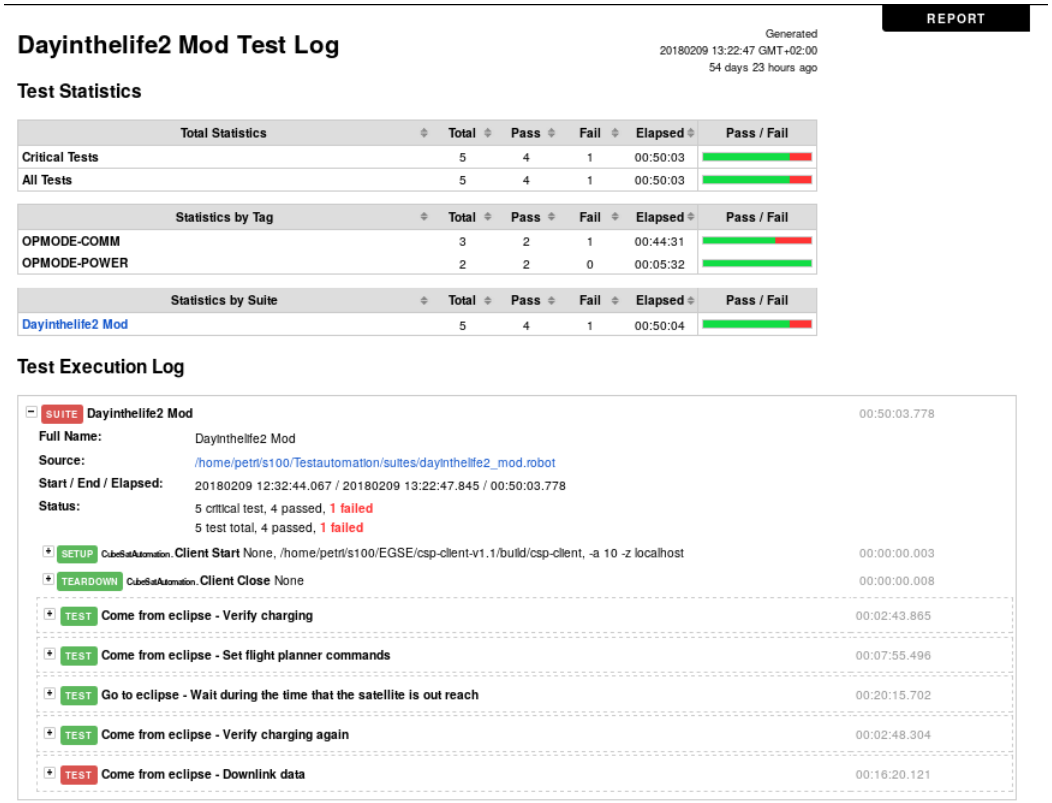
Test Execution Log

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

Figure D10: Test execution results from testing of "Day in the life" scenario 1.



REPORT

Dayinthelife2 Mod Test Log

Generated
20180209 13:22:47 GMT+02:00
54 days 23 hours ago

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

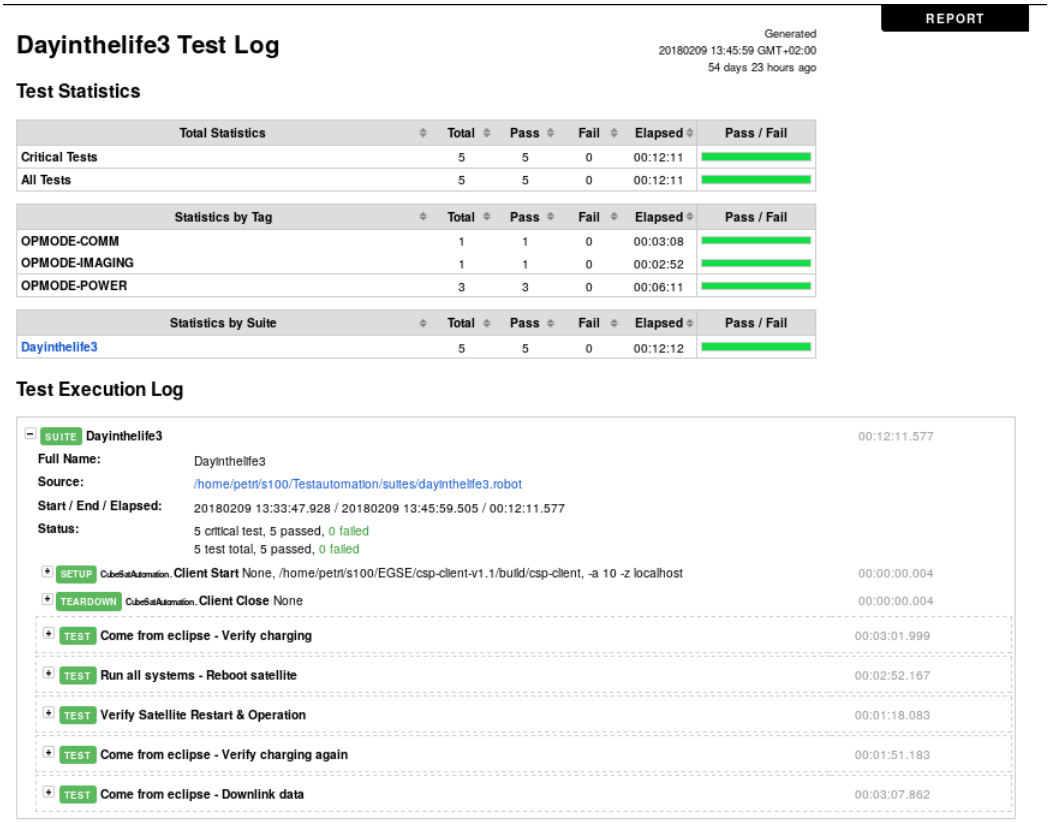
TEST **Come from eclipse - Set flight planner commands** 00:07:55.496

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

Figure D11: Test execution results from testing of "Day in the life" scenario 2.



REPORT

Dayinthelife3 Test Log

Generated
20180209 13:45:59 GMT+02:00
54 days 23 hours ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	5	5	0	00:12:11	
All Tests	5	5	0	00:12:11	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
OPMODE-COMM	1	1	0	00:03:08	
OPMODE-IMAGING	1	1	0	00:02:52	
OPMODE-POWER	3	3	0	00:06:11	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Dayinthelife3	5	5	0	00:12:12	

Test Execution Log

SUITE **Dayinthelife3** 00:12:11.577

Full Name: Dayinthelife3
Source: /home/petri/s100/Testautomation/suites/dayinthelife3.robot
Start / End / Elapsed: 20180209 13:33:47.928 / 20180209 13:45:59.505 / 00:12:11.577
Status: 5 critical test, 5 passed, 0 failed
5 test total, 5 passed, 0 failed

SETUP CubeSatAutomation.**Client Start** None, /home/petri/s100/EGSE/csp-client-v1.1/build/csp-client, -a 10 -z localhost 00:00:00.004

TEARDOWN CubeSatAutomation.**Client Close** None 00:00:00.004

TEST **Come from eclipse - Verify charging** 00:03:01.999

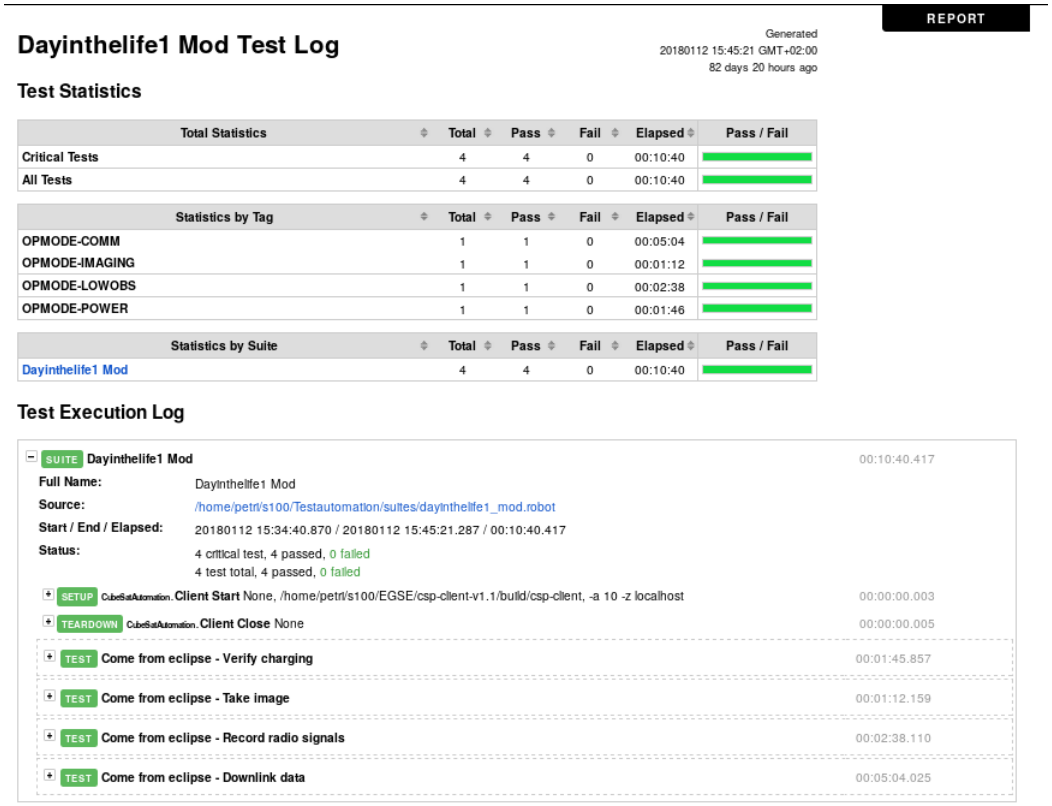
TEST **Run all systems - Reboot satellite** 00:02:52.167

TEST **Verify Satellite Restart & Operation** 00:01:18.083

TEST **Come from eclipse - Verify charging again** 00:01:51.183

TEST **Come from eclipse - Downlink data** 00:03:07.862

Figure D12: Test execution results from testing of "Day in the life" scenario 3.



REPORT

Dayinthelife1 Mod Test Log

Generated
20180112 15:45:21 GMT+02:00
82 days 20 hours ago

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

Figure D13: Test execution results from testing of "Day in the life" scenario 4.