Nico Bucher

Merging Spacecraft Software Development and System Tests

an agile Verification Approach

MERGING SPACECRAFT SOFTWARE DEVELOPMENT AND SYSTEM TESTS

AN AGILE VERIFICATION APPROACH

A thesis accepted by the Faculty of Aerospace Engineering and Geodesy of the University of Stuttgart in partial fulfilment of the requirements for the degree of Doctor of Engineering Sciences (Dr.-Ing.)

by

Nico Florian Bucher

born in Göppingen, Germany

Main referee:	Prof. Dr.-Ing. Jens Eickhoff
Co referee:	Prof. Dr.-Ing. Andreas Rittweger
Co referee:	Prof. Dr.-Ing. Sabine Klinkner
Date of defense:	December 3rd, 2020

Institute of Space Systems
University of Stuttgart
2021

CONTENTS

—

I. ABSTRACT I

II. KURZZUSAMMENFASSUNG III

1. INTRODUCTION 1
1.1. Satellite Missions at Universities 3
1.2. Flying Laptop 5
1.3. Outline and Aim of this Thesis 7

2. SOFTWARE DEVELOPMENT AND SPACE PROJECTS 9
2.1. Software Development Principles 9
2.1.1. Plan-driven Software Engineering 10
2.1.2. Agile Software Development 15
2.1.3. Test-driven Development 19
2.1.4. Combining Different Development Approaches 20
2.2. Spacecraft Development and Testing Process 23
2.2.1. Lifecycle of Traditional Space Flight Software 25
2.2.2. Independent Software Verification and Validation 27
2.2.3. The Use of Simulators in Space Missions and Model-based Satellite Simulation 28
2.2.4. Methods of Spacecraft Test Automation 30
2.3. Software Testing 32
2.3.1. Definition of Software Tests 33
2.3.2. Different Levels of Testing 35
2.3.3. Regression Testing 37
2.3.4. Static Analysis Methods 37
2.3.5. Continuous Integration Tests 38
2.3.6. Exploratory Testing 39
2.3.7. Software Quality Metrics 39

3. FLYING LAPTOP SYSTEM DESCRIPTION 41
3.1. The Flying Laptop Project 41
3.2. System Overview 43
3.2.1. The Flying Laptop Onboard Computer 44
3.2.2. Attitude Control 46
3.3. Flying Laptop Onboard Software 47
3.3.1. Basic Architecture of the Flying Laptop Onboard Software 48
3.3.2. TM/TC and Ground Services 51
3.3.3. The Platform Abstraction Model, Health- and Mode-Tree 52
3.3.4. Testability of OBSW 55

3.4. The Flying Laptop System Testbed 56
3.4.1. Simulation Models 60
3.4.2. Error Injection 63

4. DEVELOPMENT PROCESS AND TESTING OF FLYING LAPTOP OBSW 65
4.1. Classic System Test Approach 65
4.2. Agile System Test Driven Development Approach 66
4.3. Phases of OBSW Development 73
4.4. The OBSW Development Approach Summarized 78
4.5. Release Life Cycle, Toolchain and Versioning 78
4.5.1. Release Life Cycle 79
4.5.2. Toolchain 79
4.5.3. Versioning 81
4.6. Handling of Regressions 82
4.7. Event Reporting Service for Satellite Introspection 82

5. FLYING LAPTOP SYSTEM TEST CAMPAIGN 87
5.1. From Flatsat to Fully Integrated Satellite 88
5.1.1. System Startup Procedure and Basic Communication 90
5.1.2. Power Supply and Distribution 91
5.1.3. Device Handler Development and System Integration 95
5.1.4. Electromagnetic Compatibility 108
5.1.5. Benefits of System Test Driven Approach in early System Integration 110
5.2. System Mode Transitions 111
5.3. Thermal Vacuum Tests 116
5.3.1. Wrong Calibration of a Single Temperature Sensor 119
5.3.2. Benefits of System Test Driven Approach 120
5.4. Attitude Control Subsystem 121
5.4.1. Flying Laptop Attitude Control Subsystem Tests 122
5.4.2. Verification of Magnetic Commands 123
5.4.3. Sun Sensors 128
5.4.4. Testing the GPS Receivers 129
5.4.5. Difficulties with Testing Star Tracker 133
5.4.6. Evaluation of System Test Driven Approach for Verification Attitude Control 137
5.5. Payload Tests 138
5.5.1. Payload Flatsat 141
5.5.2. Payload Testbed for Mission Support 142
5.5.3. Possible Benefits for Payload Instrument Development 143
5.6. Operations Scenarios 144
5.6.1. Launch and Early Orbit Phase Test 146
5.6.2. Week-in-the-life Test 148
5.6.3. Evaluation of Operational Tests in the Context of the System Test Driven Approach 149
5.7. Final Acceptance Test 150

6. PURE SOFTWARE TESTS IN SYSTEM TESTBED 153
6.1. Evaluation of Simulation Equipment Models..153
6.1.1. Evaluation of Power Supply Unit, Battery and Solar Panel Model..........153
6.1.2. Evaluation of Sun Sensor Model 157
6.1.3. Evaluation of Reaction Wheel Model..161
6.2. Failure Detection, Isolation and Recovery..163
6.2.1. Using Event Injection to Test Failure Handling..166
6.2.2. Failure Detection..168
6.3. Abbreviated Functional Tests..170
6.4. PyOps..172
6.5. Automation of Integration Tests and Build Process..175
6.6. Application of Exploratory Testing..178

7. DISCUSSION AND OUTLOOK..179
7.1. Discussion..180
7.1.1. Retrospective on the Software Development Approach..181
7.1.2. Simplification of System Tests due to Simulation 183
7.1.3. Automation of Software Tests and Build Chain..184
7.1.4. Increasing OBSW Quality 185
7.1.5. Risk and OBSW Metrics..186
7.2. Outlook 190

III. BIBLIOGRAPHY 193

—

APPENDIX A: Detailed Functional Test Program..201

APPENDIX B: Flying Laptop Simulation Models 203

APPENDIX C: System Modes..205

APPENDIX D: Full Operations Tests..211

APPENDIX E: AFT Modules and Test Scripts..213

LIST OF ABBREVIATIONS

ACS	Attitude Control (Sub-)System
AIS	Automatic Identification System
API	Application Programming Interface
AR	Acceptance Review
BOL/EOL	Begin of Operational Life/End of Operational Life
CCSDS	Consultative Committee for Space Standardization
CDPI	Combined Data and Power Infrastructure
CDR	Critical Design Review
CI	Continuous Integration
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-shelf
DDS	Data Downlink System
DLR	German Aerospace Center
ECEF	Earth Centered, Earth Fixed Coordinate System
ECSS	European Cooperation for Space Standardization
EFM	Electrical Functional Model a.k.a. Flatsat
EGSE	Electrical Ground Support Equipment
EM	Engineering Model
ESA	European Space Agency
FDIR	Failure Detection, Isolation and Recovery
FM	Flight Model
FORTRAN	Formula Translation (Programming Language)
FP	Function Point
FPGA	Field Programmable Gated Array
FSFW	Flight Software Framework
GMFE	Generic Modular Frontend
GPIO	General-Purpose Input/Output
GPS	Global Positioning System
HIL	Hardware-in-the-Loop
IDE	Integrated Development Environment
IRS	Institute of Space Systems at the University of Stuttgart
IST	Integrated Subsystem Test
JTAG	Joint Test Action Group
KLOC	Kilo Lines of Code

LEO	Low Earth Orbit
LEOP	Launch and Early Orbit Phase
LSB	Least Significant Bit
MCS	Mission Control System
MDVE	Model-based Development and Verification Environment
MGM	Magnetometer
MGT	Magnetic Torquer
MIB	Mission Information Base (of SCOS-2000)
MOIS	Manufacturing and Operations Information System
MVP	Minimum Viable Product
NASA	National Aeronautics and Space Administration
OBC	Onboard Computer
OBSW	Onboard Software
OS	Operating System
OSAL	Operating System Abstraction Layer
OST	Operational System Timeline Test
PCB	Printed Circuit Board
PCDU	Power Control and Distribution Unit
PDR	Preliminary Design Review
PLOC	Payload Onboard Computer
PLUTO	Procedure Language for Users in Test and Operations
POSIX	Portable Operating System Interface
PPS	Pulse per Second
PROM	Programmable Read-Only Memory
PSS	Power Supply (Sub-)System
PUS	Packet Utilization Standard
QR	Qualification Review
REST	Representational State Transfer
RF	Radio Frequency
RMAP	Remote Memory Access Protocol
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Realtime Operating System
RTS	Real-Time Simulator
RW	Reaction Wheel
SCOE	Special Checkout Equipment
SDR	System Design Review

SFT	System Function Test
SIF	Service Interface
SLOC	Single/Effective Lines of Code
SRD	System/Software Requirements Document
SRR	System Requirements Review
STR	Star Tracker
SuS	Sun Sensor
SVT	System Validation Test
TBTV	Thermal Balance, Thermal Vacuum
TC	(Tele-)Command
TCS	Thermal Control (Sub-)System
TM	Telemetry
TRR	Test Readiness Review
TT&C	Telemetry, Tracking and Command
V&V	Verification and Validation

ABSTRACT

The ever-increasing complexity of software requires new approaches to software development to still be able to deliver software quickly but at the same time maintain its quality. Software for satellite systems is no exception. Satellites use overwhelmingly complex software, while at the same time satellite missions have strong demands towards its quality.

This thesis presents an agile functional verification approach for satellite systems which combines functional system testing and flight software development. In the presented approach, functional testing is used as the driver for flight software development. Model-based simulation techniques and modern continuous integration and test methodologies are used as a basis for the proposed approach to increase the effectiveness and efficiency of software development activities. By viewing functional testing and software testing together as one unified method, synergies are created which can be exploited to increase dynamism in satellite design and project management.

The approach has been developed for the small satellite of the University of Stuttgart: *Flying Laptop*. The *Flying Laptop* mission is used as an example for the application of the approach and is used to evaluate its downsides and benefits.

The *Flying Laptop* onboard software was developed by a team of graduate students that applied the test approach throughout the flight software development. Although the approach was developed especially for the object-oriented *Flying Laptop* onboard software it can be applied to many other software and mission types in which flexibility is required. The approach is well suited for small agile teams and presents an alternative to the traditional requirements engineering approach based on waterfall or V-model development methods.

The unique structure of the *Flying Laptop* project allowed for some experimentation with unusual agile software development practices which eventually led to the feasibility of this study. In strongly customer-oriented industry projects, this degree of experimentation would not have been possible. The presented development process leverages these unique opportunities, especially the possibility to build the onboard software step by step during the system testing campaign of the satellite whereas in normal satellite projects this is done consecutively.

Additionally, the testing methodologies, testbeds and toolchains for *Flying Laptop* are described. As part of the approach, many functional system tests were performed in simulation, which means the onboard computer was embedded in a real-time simulation where the rest of the satellite is represented through functional equipment models.

As a further part of this work, the question of whether the degree of resemblance achieved with the used simulation models is suitable for the performed functional system tests is addressed. This assessment is based on comparing equipment model data to real in-orbit data gathered throughout the *Flying Laptop* mission. At the same time, possible simplifications of simulation models are considered in order to reduce simulation complexity.

Automating test procedure execution was another key element of the approach. The used automation techniques are described including a novel operations script module based on Python. Exploratory software testing methods were explored as an additional method of test case creation to increase test coverage and to uncover additional software problems but also to increase the usability of the software.

KURZZUSAMMENFASSUNG

Um der zunehmenden komplexität von Software Herr zu werden ist es erforderlich, neue Ansätze der Softwareentwicklung zu entwerfen die aber gleichzeitig die Qaulität der Software nicht vermindern. Auch Satellitensysteme verwenden mehr und mehr komplexe Software und stellen aber gleichzeitig hohe Anforderungen an ihre Qualität.

In dieser Arbeit wird ein agiler Entwicklungsansatz für Flugsoftware von Satellitensysteme vorgestellt, welcher die Durchführung von Systemtests mit der Entwicklung der Flugsoftware vereint. Hierbei werden funktionale Systemtests als Richtungsvorgabe für die Softwareentwicklung verwendet. Modellbasierte Simulationstechniken und Continuous Integration- und Testmethodiken dienen als Basis des Ansatzes. Gleichsam wird untersucht ob der Einsatz dieser Techniken die Effektivität und Effizienz der Flugsoftwareentwicklung begünstigt. Durch das Vereinen von funktionalen Systemtests und Softwareentwicklung zu einem einheitlichen Prozess ergeben sich Synergien die die Dynamik des Satellitendesigns erhöhen und vorteilhaft für das Projektmanagement sind.

Der Ansatz wurde für das Kleinsatellitenprojekt *Flying Laptop* der Universität Stuttgart entwickelt. Das *Flying Laptop* Projekt wird auch als exemplarisches Fallbeispiel zur Anwendung des Ansatzes verwendet. Anhand von beispielhaften Erfahrung, welche mit der Anwendung des Ansatzes gesammelt wurden, werden die Vor- und Nachteile für die Softwareentwicklung erläutert.

Die Flugsoftware von *Flying Laptop* wurde durch ein Team aus Doktoranden entwickelt, welches den Entwicklungsansatz über das gesamte Entwicklungsprojekt hinweg anwendete. Auch wenn der Entwicklungsansatz speziell für das *Flying Laptop* Projekt entwickelt wurde, ist die Verwendung des Ansatzes auch in anderen Projekten denkbar, in denen erhöhte Flexibilität erforderlich ist. Der Ansatz eignet sich insbesondere für kleine, agile Teams und stellt eine alternative zum klassischen Entwicklungsansatz für anspruchsvolle, eingebettete Software, welcher auf der statischen Umsetzung von Anforderungen nach dem V-Modell beruht, dar.

Die einzigartige Struktur des *Flying Laptop* Projekts ermöglichte es, unübliche, agile Verfahren im Rahmen dieser Arbeit auszuprobieren. Dies wäre in stark kundenorientierten Industrieprojekten unmöglich gewesen. Im Rahmen dieser Freiheit wurde die Flugsoftware des *Flying Laptop* Schritt für Schritt, während der parallel ablaufenden Systemtestkampagne, erstellt. In gewöhnlichen Satellitenprojekten folgen diese Phasen zeitlich aufeinander, größtenteils ohne sich dabei zu überschneiden.

Außerdem werden die für *Flying Laptop* eingesetzten kontinuierlichen Testmethoden, die verwendeten Teststände, sowie verwendeten Softwarewerkzeuge beschrieben. Ein entscheidender Aspekt der Systemteststrategie war es, einen Großteil der funktionalen Systemtests auf die Simulationsumgebung zu stützen. Die Simulationsumgebung besteht aus dem Bordrechner des *Flying Laptop*, welcher zur Ausführung der Flugsoftware in eine Echtzeitsimulation eingebunden ist. Der Rest des Satellitensystems wird dort durch funktionalen Equipmentmodelle abgebildet.

Im Zuge des verstärkten Einsatzes von Simulationstechnologie wird im Verlauf der Arbeit bewertet, ob der Grad an Modellierung der Simulationsmodelle angemessen für die durchgeführten funktionalen Tests war. Bei dieser Bewertung werden Simulationsdaten mit echten Orbitdaten, die im Verlauf der *Flying Laptop* Mission aufgezeichnet wurde, verglichen.

Das Ausführen von automatisierten Testprozeduren ist ein weiteres Schlüsselelement des Testverfahrens. Außerdem sollte durch die zusätzliche Anwendung von explorativen Verfahren zur Testfallerstellung die Abdeckungsrate der Tests erhöht werden. Die Erfahrungen mit den verwendeten Automatisierungstechniken sowie explorativen Testmethoden werden als weiterer Bestandteil der Arbeit erläutert.

1 Introduction

“HAL, you have an enormous responsibility on this mission, in many ways perhaps the greatest responsibility of any single mission element. You're the brain, and central nervous system of the ship. […] Does this ever cause you any lack of confidence?”

“Let me put it this way, Mr. Amer. The 9000 Series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words foolproof and incapable of error.” [Kubrick 1968]

Interview between reporter Martin Amer and the onboard computer of the spaceship Discovery 1 on its way to Jupiter.

—

It is a bold statement by one of cinema history's most infamous electrical brains. The quotation from the movie "2001: A Space Odyssey" shows remarkably that computers and space have always fascinated engineers and science fiction writers alike. Statements like these are the reason for onboard computing and space software to be surrounded by an aura of mysticism and esteem.

But despite all the glorious achievements that space missions had over decades, we all know examples of failing software causing trouble. Not only in movies, have engineers overestimated their abilities to create error-free systems. In fact, the quote from the film shows impressively the euphoric arrogance that prevailed in the early days of computer-controlled space flight towards man's capability to create perfect computer systems.

In the real world, history has shown that problems caused by computers or, more precisely, software range from minor glitches in performance of a spacecraft up to mission loss. Interplanetary probes are missing their destination because of mismatches in metric and imperial units [Stephenson 1999], launch vehicles explode because of unexpected effects of software reuse [Lions 1996], and numerous CubeSat missions are failing because their onboard software is error-prone [Swartwout 2016]. Nowadays, spacecraft and in particular satellites are autonomous systems that rely heavily on embedded software. Errors in complex software can be hard to find yet their effects can be catastrophic.

Various factors influence the success probability of space missions such as spacecraft complexity, mission type, design choices, failure tolerant design, and testing. While most of these influences are dictated by mission requirements, testing is a field that is subject to much debate, especially in the software world. The right amount of testing and the kind of testing which is required is often not as clear as one might expect. However, a widely accepted opinion of most engineers is that you can never test too much. Many different testing techniques can be applied to identify design flaws or production errors.

Besides testing the spacecraft system itself, software tests are in a category of their own. In recent years there have been significant advancements in the field of software testing. Companies that draw their revenue from software products realized that good software is an asset on its own. Excellent quality of software (that includes the number of errors but also usability) can either capture or lose entire markets. These companies are shaping the modern world like none before and their software has become a major part of everyday life and society.

At the same time, software development has become extremely fast-paced. With new development methodologies and paradigms like agile and continuous integration and deployment techniques, it can take only seconds, in extreme cases, until modifications to a program by a developer are propagated into the production environment without diminishing software quality. All of this is made possible by automated testing and integration. As easy as it may seem to produce good quality software, it is tough to develop error-free software especially if the complexity of the software increases.

Spaceflight software is very complex and it has to be very reliable. In contrast to modern tech industry standards, demands regarding "time to market" are comparably low for spaceflight software because space mission programs are usually long-lasting. Nevertheless, new companies like Planet[1] (formerly Planet Labs) and SpaceX[2] are speeding up space software development at the moment especially in the low earth orbit (LEO) market. Because of this increase in development speed, there is a need for efficient software development and test-

[1] Planet Labs website: www.planet.com

[2] SpaceX website www.spacex.com

ing techniques for space projects which at the same time increase the quality of the software and on the other hand lower the workload on individual testers.

1.1 Satellite Missions at Universities

Amongst organizations who build satellites today are companies, agencies, the military but also Universities. Universities have a unique role as their incentive to have their own satellite missions is less obvious compared to, e.g. space agencies. However, by building and operating their own satellites Universities pursue one or a combination of the following goals:

- Education in the field of space engineering
- Technology demonstration for new emerging trends which major space agencies are hesitant to put into space
- Scientific applications in the field of astronomy, Earth observation and physics

Because space missions are expensive and funds at Universities are contested, Universities are often criticized for their efforts in space. The money spent on space technology could be much better used otherwise. On the other hand, space fascinates students. Especially in the field of engineering, there is no better alternative than to acquire knowledge and experience first hand which makes the participation of students in real space projects invaluable. Besides, as a benefit of the establishment of the CubeSat standard, it is getting cheaper and cheaper for Universities to build and launch their own space technology.

Although most Universities are currently building CubeSats, there are a few exceptions. The University of Surrey started its first small satellite UoSat-1 as early as 1981. The spin-out company SSTL Ltd (Surrey Satellite Technology Ltd) was founded in 1985[3] [Kramer et al. 2008].

The Technical University of Berlin (TUB) maintains a small satellite program and has built a number of satellites whose form factor differs from the CubeSat standard. Their micro and nano satellite buses TUBiX20 and TUBSAT were successfully used in several missions for about 25 years [Brieß 2017], [Barschke et al. 2016].

[3] SSTL website: www.sstl.co.uk

Since 2004, the University of Stuttgart runs a small satellites program as well. At the Institute of space systems of the University of Stuttgart, the small satellite *Flying Laptop* was the first built small satellite. *Flying Laptop* was launched in 2017.[4]

In 2005, SSETI Express, an educational satellite designed and built in a joint effort of several European Universities was launched by the European space agency. With their unique approach of working together over the internet, the international team managed to design and build the satellite remarkably quick. Onboard SSETI Express were three CubeSats which were deployed successfully right after launch. After a few hours of operation, the student team had to realize the loss of their mission due to a failure in the power supply subsystem of the micro satellite which prevented charging of the batteries. According to [Viscor 2006] the design flaw could have been noticed by better testing [Viscor 2006].

Likewise, although the number of CubeSat launches has increased dramatically over the last years, CubeSat missions have a high failure rate compared to larger satellite missions. The success rate of CubeSat mission is about 50% (shown in Figure 1.1.1) which means every second CubeSat either does not make it to orbit, fails early during missions time or communication to the satellite could never be established [Swartwout 2017].

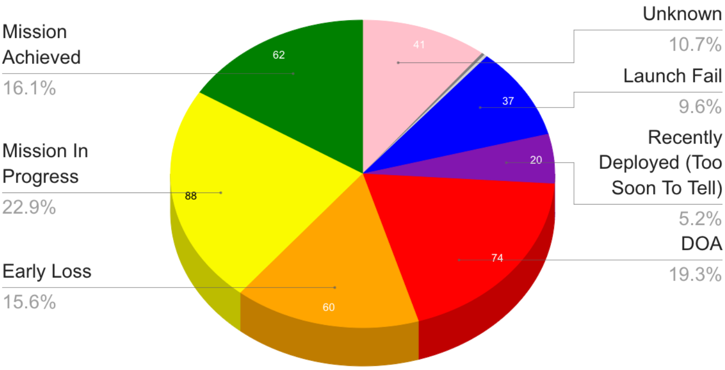


Figure 1.1.1: Overview of CubeSat mission success; DOA = dead on arrival [Swartwout 2017]

[4] See www.kleinsatelliten.de for information

1.2 Flying Laptop

Flying Laptop is the first satellite of the small satellites program at the University of Stuttgart initiated by Prof. Hans-Peter Röser(†) in 2004. Its mission goals are education, scientific Earth observation and technology demonstration.

Flying Laptop was launched into a sun-synchronous orbit of 600 km altitude and an inclination of 97.6° on July 14th, 2017 from the Baikonur Cosmodrome launch complex by a Russian Soyuz-2-1a rocket with Fregat upper stage. With its mass of 112 kg, *Flying Laptop* is in the category of micro satellites according to the definitions in [Brieß 2017] or a mini satellite when following the nomenclature of [Helvajian et al. 2008].

As described in section 3.1, as part of the *Flying Laptop* project a state-of-the-art onboard computer design was developed in cooperation with industry partners which in combination with the power control and distribution unit forms the combined data & power management infrastructure (CDPI) [Eickhoff 2013]. The CDPI reduces system complexity by integrating reconfiguration logic into the power unit of the satellite. This revolutionary onboard computer together with the flight software, developed at the University of Stuttgart [Bätz 2018], represents the heart of the *Flying Laptop* satellite bus.

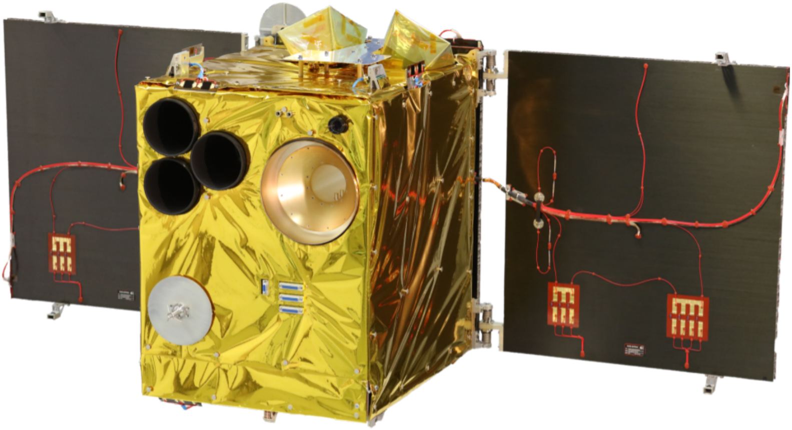


Figure 1.2.1: The small satellite Flying Laptop ©Jonas Keim

Flying Laptop is an educational satellite. Its development, operation and data analysis has been the topic of many student theses. Likewise, students can gain hands-on experience in satellite operation through lectures and exercises offered throughout its mission time. Because of the involvement of industry partners in the project and the application of industry standards like ECSS and CCSDS, students get to know technology and processes very similar to the ones used in space industry. Thus, after their involvement in the *Flying Laptop* project, graduating students are well trained to work in different areas of space industry.

1.3 Outline and Aim of this Thesis

Growing demand to produce error-free software for systems with ever-increasing complexity, like satellites, requires new approaches for software development and testing to contain costs and project duration. While exploring solutions for these challenges, this thesis aims to determine whether:

- It is possible to form a software development approach that better merges flight software development and functional system testing.
- This software development approach increases dynamism within the development team and thus enables it to better react to unexpected changes and problems.
- Through the application of this approach, satellite development can be streamlined without reducing the overall quality of the software.
- Functional system test complexity can be reduced by performing a portion of tests in full simulation which would typically require the satellite system under test in a closed-loop test environment.
- The application of modern continuous deployment tools and test automation further increases flight software quality and benefits the overall scheme of the approach.
- The software development approach can be applied to similar projects and if so, which conditions for the project have to be fulfilled so that the approach can be adopted.

To find answers to these questions, the approach was applied during the flight software development of the small satellite *Flying Laptop*. The present thesis describes the experiences made during the application of the approach and at the end draws conclusions to answer the questions stated above.

As a further introduction to the topic, chapter 2 summarizes the state of the art of software development in space projects. Chapter 3 then focuses on the specific project that served as the studying object of this thesis, the *Flying Laptop* project and the special conditions that led to the adoption of the here presented approach. Chapter 3 also goes into detail about the digital twin of *Flying Laptop*, the system simulation testbed, and describes the development of its simulation models.

Chapter 4 introduces the software development process for *Flying Laptop* based on the foundations laid out in chapter 3 and explains how this process was embedded in the rest of the *Flying Laptop* lifecycle.

Since the software development approach is based on the system test campaign of the satellite, chapter 5 provides an in-depth report about this phase of the *Flying Laptop* project and describes how the flight software development progressed along with it. At the same time, chapter 5 gives concrete examples of how specific problems during system testing could be solved as a result of the new development approach.

Finally, chapter 6 then describes pure software tests that were performed within the model-based simulation environment. Chapter 6 further focuses on the evaluation of used simulation models and finally describes the automation of software testing activities that were performed for the *Flying Laptop* besides functional system testing. All of the automated software tests were executed on the system simulation testbed including the model-based spacecraft simulator and modern continuous integration tools.

This thesis is concluded by a discussion and outlook provided in chapter 7 where the achievements of this thesis concerning the goals listed above are reflected and potential future application, as well as improvements, are discussed.

2 Software Development and Space Projects

This chapter contains information about the state of the art in software development in the space industry and software testing in general. Section 2.1 gives an overview of the principles of software development to lay a foundation for the reader to be able to classify the software development approach applied for *Flying Laptop*. Sections 2.2 describes how software development is traditionally performed in space projects and also contains information about spacecraft simulation practices. Section 2.3 contains general information about methods of software testing in general, continuous integration practices and software quality metrics.

2.1 Software Development Principles

Since the beginning of software development as an engineering discipline, a debate about the correct development principles is ongoing. Principles are the basis from which processes in engineering are derived. It quickly became clear that processes, as used for classic engineering disciplines, are not especially suited for software engineering. However, processes are required to guide developers in their daily work and to measure the progress of a project. A rational, formal process derived from helpful principles provides a clear understanding of the roles of team members and a guideline on how to approach the task of transforming ideas into real software.

In an ideal top-down approach, a developer would implement a program compliant with a set of requirements. It is the job of the project manager to monitor the progress of these development activities and to shift priorities as required. In real-world projects, however, requirements are often not clear at the beginning, change over time or conflict with each other. In reality, managers often struggle to make the right decisions. Also, management is often too isolated from the team so that managers can not sense what is going on. Another problem is that it is very difficult to phrase exact requirements right from the start of a project. The accurate details of a design only occur to the developers during the implementation process which often becomes apparent when several versions of prototypes are created before a design finally suits all needs.

Several development methodologies have been proposed to deal with these problems such as Scrum, test-driven software development or extreme programming. These processes try to solve the described problem by describing processes that aim to structure daily, weekly and monthly planning and work. The descriptions of a development process used in these methodologies are always an idealization, and in real projects, a mixture of all these methodologies is applied. Rarely does a project apply a methodology as described by the book. Different preferences of team members, managers and organizational guidelines all shape the process that is applied in reality. Nevertheless, it is required to describe the applied approach in a way that a common understanding of what everybody's duties and deadlines are, is created for all participants.

In their 1986 paper "A rational design process, how and why to fake it" [Parnas et al. 1986], Parnas and Clements argue that although one can never fully comply with such a rational design process, it is still very useful to create and to follow one. They list four items any rational development process should describe:

- What item to work on next
- What criteria must that item satisfy
- What kind of person should do the work
- What information they should use in their work

In this chapter, the development principles are described which are relevant to the proposed development approach.

2.1.1 Plan-driven Software Engineering

This section provides an overview of plan-driven development methods which are sometimes also referred to as requirements engineering, waterfall methods or big design up-front in contrast to agile development methods (described in section 2.1.2). Such processes are applied by space projects that adhere to software development standards like [ECSS Committee 2009] and [ECSS Committee 2017], the ECSS standards on space software engineering and product assurance.

A requirement or specification is a short statement that expresses a need towards a product or a process. Requirements can have different levels of technicality, e.g., a requirement can be a superficial description of an expected functionality of a product expressed by a customer or an in-depth technical specification written by an engineer. Two different categories of requirements are usually differentiated, i.e., functional requirements and non-functional requirements. While functional requirements describe functionalities a product should provide, non-functional requirements describe other aspects of a design like external interfaces, performance or design considerations like the overall architecture, safety and security measures or quality metrics [Laplante 2013].

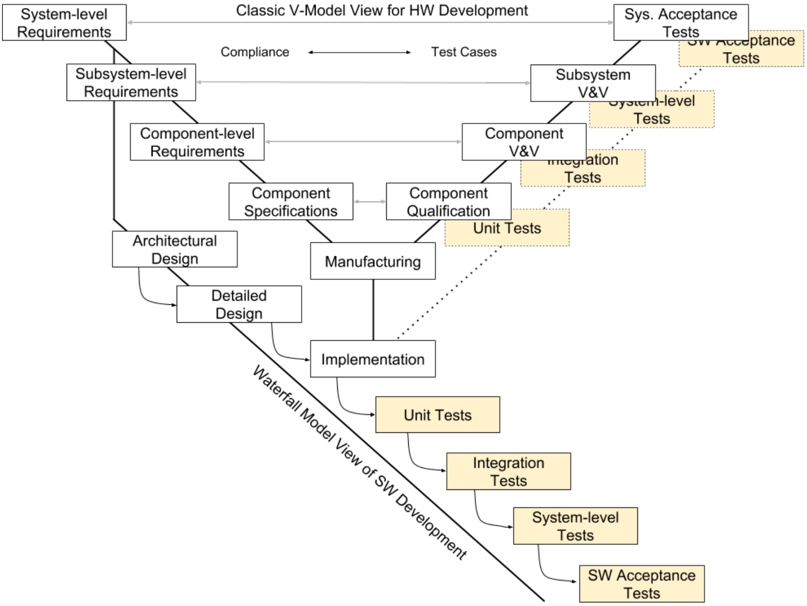


Figure 2.1.1: Waterfall model view and V-model view for software/hardware development compared

Figure 2.1.1 visualizes the classic process of plan-driven development. Requirements are derived from needs expressed by stakeholders and domain-specific constraints. Starting with abstract high-level requirements, each requirement can be broken down into more and more tangible specifications until each statement is specific enough to be implemented by a developer. In Figure 2.1.1, software development branches off from regular hardware development which is typical for plan-driven development. Once implementation and manufacturing are finished, the testing of the implemented software and the manufactured hardware against the specifications begins. In the requirements-based approach each test is performed to show that a requirement is correctly implemented. Testing again starts at the lower requirements level and then gradually moves towards higher-level tests until all requirements are fulfilled.

Requirements engineering is an engineering approach that uses requirement specifications as the source for design decisions and as a guideline for verification and validation (V&V) activities. While verification describes the process of showing that something has been built according to specification, validation is the process of determining if the specifications are correct. [IEEE 1998]

Requirements engineering, or more generally, plan-driven systems engineering was originally applied in high-stakes engineering projects which need to prove a certain level of compliance to both customers and authorities. Most software engineering standards like IEEE 830-1998 [IEEE 1998] describe some form of a requirement based or waterfall engineering approach.

[Laplante 2013] defines requirements engineering as follows: "Requirements engineering is the branch of engineering concerned with the real-world goals for, functions of, and constraints on, systems. It is also concerned with relationship of these factors to precise specifications of system behavior and to their evolution over time and across families of related systems." [Laplante 2013]

At their core requirements are a means of communication between supplier and customer or between engineers in different teams within an organization. An engineer needs to efficiently describe the features of something that shall be designed to avoid misconceptions. The precise specification of software features is essential to have a common conception of the system (or software) that is being developed [IEEE 1998]. When formulated in natural language, requirements share similar characteristics with legal texts (e.g. contracts) because they aim to be precise in meaning. At the same time requirements share the same weaknesses as legal texts in achieving this goal, because natural language is always ambiguous (i.e. depending on context) and can never be precise. In their 2012 study on requirements engineering for embedded systems Sikora, Tenbergen and Pohl suggest that many system engineers and practitioners are dissatisfied with the use of natural language for requirements specification [Sikora et al. 2012].

In large projects, excessive requirements engineering leads to contradicting statements in specifications which is why it is so difficult to manage large sets of requirements. For this reason, several tools can be used to track and analyze requirements.

Usually, requirements are written down in text-documents and distributed within the project's organization and amongst suppliers. Requirement documents are typically created using a word processor. Each requirement is assigned an identifier followed by a detailed description of the requirement usually in text-form or with the help of additional schematics. Spreadsheets are also very commonly used to track the performed implementation and test activities to corresponding requirements. More sophisticated software tools exist, which allow working with requirements through a graphical user interface specifically designed for that purpose. One example of such a software tool is the Dynamic Object-Oriented Requirements System (DOORS)[5] which can be used to organize requirements in the form of dependency trees. This way, requirements can be linked more easily to corresponding test cases. While tools like DOORS offer a convenient way to work with requirements, the tools do not consider the semantics of the data. This means that e.g. contradictions within a set of requirements cannot be automatically detected. However, this would be a desirable capability of such a tool because maintaining consistency within a large set of requirements is hard to maintain.

Another effort to enhance requirements engineering by using software tools was made by NASA in the late 1990s. NASA created a tool called ARM (Automated Requirements Measurement) which statistically analyzed requirement documents with respect to the usage of specific words and grammar as quality indicators. The tool was later discontinued and reverse-engineered by [Carlson et al. 2014] to acquire quality metrics for future work in the field of natural language processing, document understanding and data mining which are related to requirements engineering [Carlson et al. 2014].

[5] See www.ibm.com/us-en/marketplace/rational-doors/ for details.

In the perception of engineers, requirement engineering is often associated with an idea of a strict and rigid process. However, the fact that a project uses requirements to guide the design process does not make it rigid per se. What makes the process inflexible is the strict commitment of sticking to project phases and a large amount of documentation that is required to provide proof of compliance to reviewers.

Such processes of high complexity have been adopted in the early days of software engineering for the lack of better alternatives. However, software development is naturally done in iterations. The product itself (the software) is not a static object which once specified will forever correspond to that specification. Customer needs towards software and insight into functional details may change but also surrounding conditions like the advancement of operating systems, and computer architectures may require changing a software product drastically during its lifetime. All of these aspects make the traditional requirements based engineering process unsuitable for the needs of a flexible software development project.

In software engineering, the term waterfall model or V-model is often used when describing a process that is inflexible and bound to the different project phases. It is debated among experts, however, if the term waterfall model describes a process of any real-world relevance[6]. As visible from Figure 2.1.1, V- and waterfall models are basically different views on the same process. For the sake of explicitness, throughout this thesis, the term requirements based engineering will refer to a development process that is based on specifications with a strict commitment to project phases and milestones. Although this definition refers to systems and not to software specifically, it is equally applicable to pure software development. The fact that the definition is valid not only for software already indicates its inherent problem:

> The process of requirements engineering is not especially suited for the purpose of software development.

[6] © Conrad Weisert, Information Disciplines, Inc., Chicago 8 February, 2003 www.idinews.com/waterfall.html

2.1.2 Agile Software Development

In 2001, a group of experienced software developers summarized their ideas for an efficient way to develop software which they gathered throughout their work life in the so-called manifesto for agile software development [Beck et al. 2018]. This manifesto summarizes aspects that have emerged in the computer and software development industry for some time favoring a pragmatic way of working.

The term agile points towards the understanding that teams who adhere to agile principles have increased agility, very similar to the way in which lean manufacturing shaped the automobile industry. The terms leanness and agility suggest an efficient, yet simple and lightweight approach.

As the authors of [Beck et al. 2018] remark, processes, tools, documentation, contracts, and plans are nevertheless essential but, according to them, there is more value in direct interactions with colleagues, users and customers and in producing working software in a dynamic way.

One aspect of agile is the ability to dynamically react to change and to welcome change. An often heard guideline in the computer industry is to "never change a running system," but this guideline is an obvious contradiction with the agile idea. In an agile mindset, the corresponding guideline would be to *always* change a running system if a better solution presents itself.

The process of constantly adapting the codebase without changing its functionality is called refactoring [Fowler et al. 1999]. To handle the problems which arise through constant refactoring, i.e., the possibility to introduce new problems, a significant effort in regression testing is required.

Thorough testing is the building block that makes constant refactoring feasible by making sure that changes in existing code do not break already implemented functionalities (cf. [Wolf et al. 2005], [Freeman et al. 2010]) and is thus an integral part of agile development.

Another step forward in technology that favored constant refactoring was the advancement of version control systems in recent years. Today, it is absolutely common to quickly merge different branches of source code and to merge conflicts automatically which significantly boosts the productivity of a developer and reduces errors introduced by manually handling conflicts.

Agile software development methods are adaptive rather than predictive. Responding to change over following a plan can thus be understood as a hint to not create a big design upfront but rather to work out the details of a design (the requirements) along the way and to learn aspects of the design from interactions with customers and team members.

This does not mean that there should not be an architecture worked out before implementation begins, but initial planning should be kept short and vague. This kind of flexibility is often realized through constantly repeating short incrementation and iteration cycles while always having a working prototype ready as an object of discussion.

The reader will notice that the traditional method of software development for space projects as described in section 2.2 is rather inflexible when compared to this agile approach. Carefully introducing agile practices into the domain of space software development could yield the potential to save time and money for future missions.

While agile describes a mindset rather than a concrete process several frameworks like extreme programming or Scrum have been derived from the agile principles. These frameworks describe how these principles should be applied in real projects. All of these frameworks are inspired by the agile principles but their focus differs.

2.1.2.1 Extreme Programming

Extreme programming (XP) focuses on practices and development techniques. Following the agile idea, XP is based on four core values:

- Simplicity
- communication
- feedback
- and courage.

From these core values the XP principles like rapid feedback, assume simplicity, incremental change, embracing change, quality work, teach learning, small initial investment, open communication and honest measurement (among others) are derived. The core of XP is the definition of the so-called XP practices which are depicted in Figure 2.1.2.

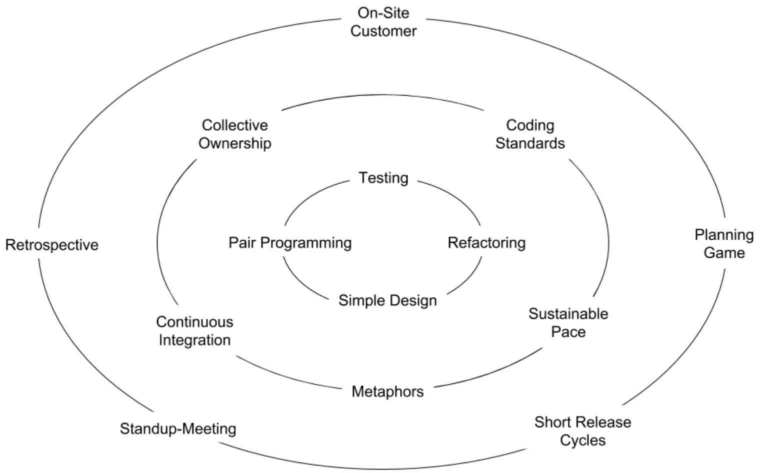


Figure 2.1.2: Extreme programming practices overview; adapted from [Wolf et al. 2005]

Particularly the XP practice called the on-site customer plays a significant role in this thesis. The idea is to integrate a customer representative into the team. This brings several advantages.

Rather than working out the details of implementation through bulky system/software requirements documents (SRDs), the customer can constantly provide direct feedback about the suitability of a solution. In addition to providing feedback, the customer provides so-called user stories that serve as a description of what the user intends to do. A user story is a form of specification but much less formal without any claim of being complete or very accurate. User-stories are a means of communicating the needs of the user to the developers. In special software projects where no customer is available, e.g., an uncommercial software or in-house development, the role of the customer can also be assumed by individuals who behave like a potential customer would [Beck et al. 2005], [Wolf et al. 2005].

It is clear that the individual on-site customer has to be fond of all particular areas which are of relevance for the product. Also, the on-site customer needs to be proficient in the motives and the philosophy of the company he or she represents and has to maintain a constant dialogue about the developed product which requires a certain amount of experience. Whether this concept is feasible for space software projects remains to be demonstrated. While some space agencies might be open to experimenting with these agile methods, most of the more traditional agencies might only adopt a similar technique once it has been shown to bring enough benefit in other, previous projects.

2.1.2.2 Scrum

Another popular agile development framework is Scrum. While extreme programming focuses on development techniques, Scrum focuses on organization and management. Originally described by Ken Schwaber and Jeff Sutherland the approach became widely spread after Ken Schwaber and Mike Beedle described it in their 2002 book [Schwaber et al. 2002]. Scrum is described completely in The Scrum Guide™ which can be downloaded from the scrum.org website[7].

Scrum defines the roles of individuals within a team. The product owner much like the onsite customer found in XP projects represents the interests of the customer. The scrum master's job is to lead the team with respect to adhering to Scrum principles. The scrum master educates the team about the process and makes sure the developers can focus on their job. In Scrum, a product backlog contains a list of features, non-functional requirements of the developed software and is maintained by the product owner [Linz 2017].

As shown in Figure 2.1.3, Scrum is organized in iterations of fixed duration, usually one or two weeks but one month at most, called sprints. The team identifies sprint goals at the start of each sprint iteration. At the end of a sprint, a sprint review is performed where the achievements are summarized and communicated to the product owner. The outcome of a sprint is always a potentially releasable software prototype [Linz 2017].

[7] See www.scrum.org/resources/scrum-guide

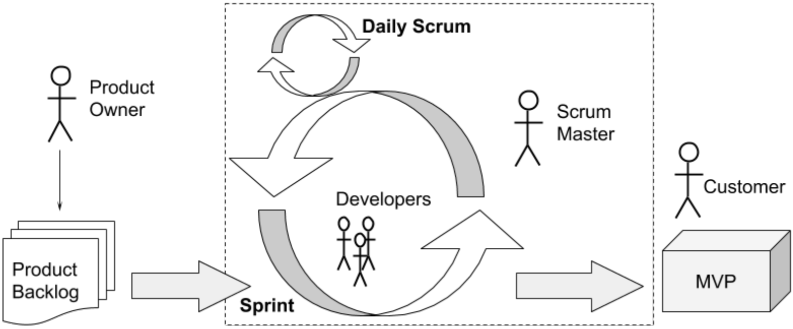


Figure 2.1.3: The Scrum process visualized; licensed under Creative Commons © User: Sebastian Wallroth/ Wikimedia Commons/CC-BY-SA-3.0; [Linz 2017]

2.1.3 Test-driven Development

Test-driven development (TDD) is not an agile framework like Scrum or XP. Instead, test-driven development is a practice used in many agile but also non-agile projects [Freeman et al. 2010].

Test-driven development realizes an intriguing idea: "To write tests for software in advance to writing the software itself." In other words, create test cases first and then build the software to satisfy those test cases [Beck 2003]. The concept of letting the development process be guided by already existing tests is turning traditional software development principles upside down since traditionally tests are written for an existing codebase after all functionalities have been implemented. The purpose of testing is no longer only to find defects in software but to help developers to understand the required functionalities [Freeman et al. 2010]. In TDD therefore, tests assume the role of specifications.

Another benefit of the TDD approach is that one problem of many projects is resolved incidentally. Many projects struggle to write good unit, regression and end-to-end tests for an existing codebase because oftentimes developers tend to concern themselves with new challenges rather than writing obvious test code for something that, in their view, is already finished. That is why the implemented tests are sometimes unsatisfying although the intent to create good tests is clearly stated by team members. Writing tests for the sake of writing tests is boring. Writing tests to define a design and to figure out what to expect from it is much more enticing.

By applying TDD, the creation of sufficient tests is guaranteed.

Normally, TDD is associated with unit tests (see section 2.3.2). The common practice is to write a unit test that determines the functionalities of the class it is written for. Usually, all major programming language provide unit testing frameworks which enable writing tests in the same programming language as the actual program. Interactions of objects with each other are tested by implementing tests using mock objects. However, TDD can also be realized with higher-level tests like integration or system tests. While unit tests are used to test the details of implementation and the intricacies of inter-object communications, system-level acceptance tests can check the fulfillment of high-level user requirements of the software and provide guidance for the larger picture.

2.1.4 Combining Different Development Approaches

Unfortunately, instead of balancing the benefits of both the agile and the plan-driven world, hard camps have evolved on both sides which consider each other as opponents. Advocates of plan-driven development accuse agile projects of being undisciplined and of producing software of low quality, unsuited for applications of high criticality. On the other hand, advocates of agile say that a plan-driven process is inhuman and inefficient because of the overwhelmingly large amount of work that is spent on planning and documentation (cf. [Boehm et al. 2009]).

[Boehm et al. 2009] suggests an evaluation scheme like the one presented in Figure 2.1.4 to find a suitable home ground for a given software development project.

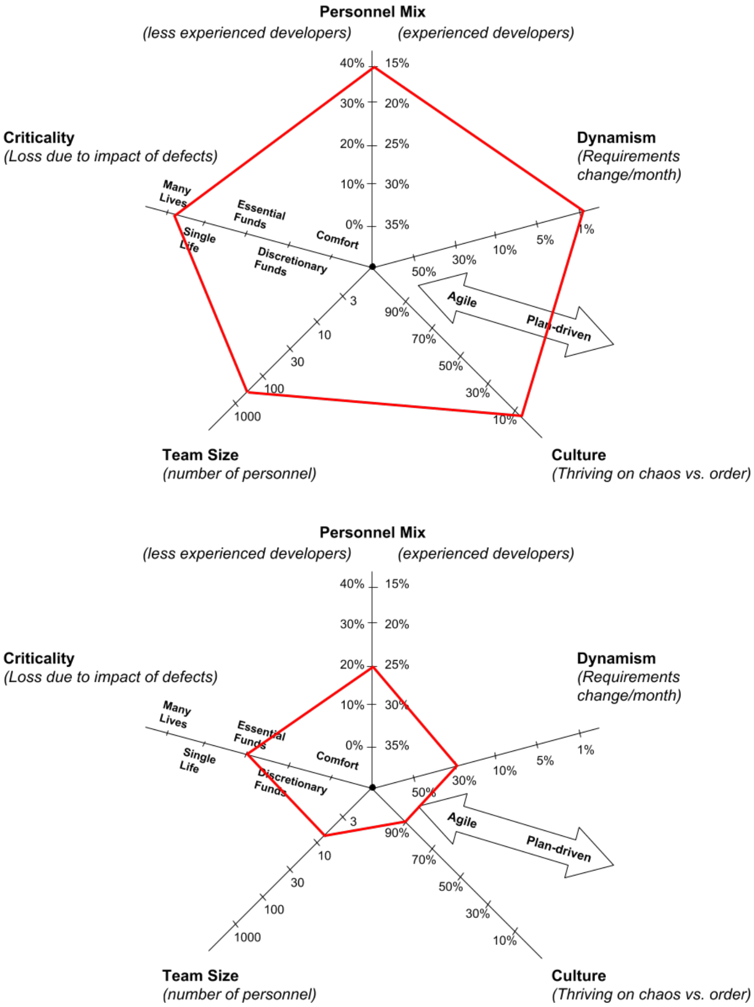


Figure 2.1.4: Identification of software development home ground for a fictional manned space project of a major space agency (top); for *Flying Laptop* flight software (bottom), as presented in [Bätz et al. 2014]; adapted from [Boehm et al. 2009]

Because of the rather extreme presentations of each development process in literature, a common misunderstanding among both camps is created, i.e. that these processes are the only way to go. In fact, none of these principles guarantee improvement over others and their application should be reconsidered and specially tailored for every project.

The evaluation scheme in Figure 2.1.4 tries to balance agility and discipline based on the following key criteria:

- Criticality determines what is at stake if the developed software fails. Plan-driven methods are better suited for critical software because of their increased efforts in documentation and quality assurance.
- Culture expresses whether a team's way of working is more chaotic or based on strict processes followed rigorously.
- Personnel mix represents a measure of the mix of skill levels within a development team.
- Projects with high dynamism have many changing requirements. Plan-driven methods prefer low rates of change.

If the evaluation of a project tends towards the center of the diagram, agile methods are likely to be suited for the given project. Likewise, if the evaluation results accumulate at the outer part of the diagram more traditional, plan-driven engineering approaches should be considered to provide the required amount of traceability and quality assurance in projects of high criticality (cf. [Boehm et al. 2009]).

The bottom graph of Figure 2.1.4 shows the evaluation results for *Flying Laptop* flight software compared with the results of a fictional crewed spacecraft like e.g. the space shuttle in the upper graph of Figure 2.1.4. It is clear that the *Flying Laptop* onboard software project leans towards agile methods with respect to culture, dynamism and team size but at the same time has considerable criticality.

2.2 Spacecraft Development and Testing Process

A traditional space project is deeply plan-driven. Space projects are divided into seven phases (phases 0 to F) which are illustrated in Figure 2.2.1. Since space missions of the European Space Agency are typically subcontracted, the standard for project planning and implementation by the European Cooperation for Space Standardization (ECSS) [ECSS Committee 2009] is widely accepted in European space industry.

After ECSS, throughout phases 0, A and B, studies are conducted to identify mission concepts and requirements to assess mission risks and to plan the subsequent phase. During phases C and D the spacecraft and ground segment are designed, produced and qualified. At the end of phase C, the procurement of parts contributed by subcontractors begins. The subcontractors are required to organize their production after the requirements released at the end of phase B. Phase E is the mission phase where the spacecraft is launched, commissioned and utilized. Finally, phase F marks the spacecraft's disposal. The conclusion of each phase is marked by a milestone in the form of a formal project review. Only if these milestone reviews are passed, the project is allowed to move on to the next phase.

The mission analysis phase (phase 0) is concluded by the mission definition review (MDR). The feasibility analysis phase (phase A) is concluded by the preliminary requirements review (PRR). The preliminary design review (PDR) judges the preliminary design and verification plan. Especially the verification plan will undergo many changes until the final version because obviously, the detailed design and interfaces are not yet fully determined. After the critical design review (CDR) the final design documents are released and based on these documents, parts procurement, production, and the verification program officially begin. Phase D milestone reviews are held to attest flight readiness. The qualification review (QR), the operational readiness review (ORR) and the acceptance review (AR) are to establish that all verification activities were successfully performed [ECSS Committee 2009].

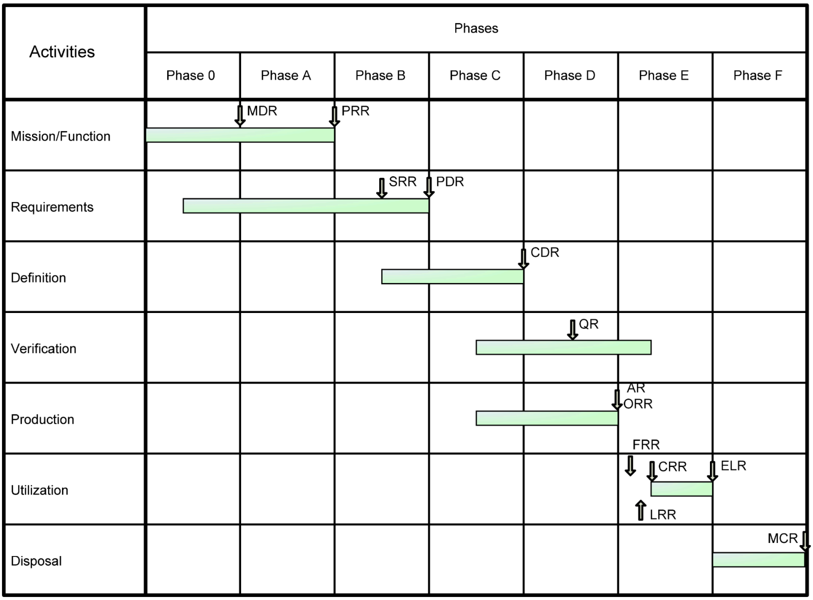


Figure 2.2.1: Traditional project phases of a space project after the European Space Agency's ECSS standard for project planning and implementation; **MDR**: mission definition review; **PRR**: preliminary requirements review; **SRR**: system requirements review; **PDR**: preliminary design review; **CDR**: critical design review; **QR**: qualification review; **ORR**: operations readiness review; **AR**: acceptance review; **FRR**: flight readiness review; **LRR**: launch readiness review; **CRR**: commissioning result review; **ELR**: end of life review; **MCR**: mission close-out review; from [ECSS Committee 2009]

As can be seen from the description above, reviews are the main tool for keeping track of project progress for space project management. Large amounts of documentation are produced throughout the whole process as documentation is necessary for reviewers, who are usually customer representatives, to gain insight into the work performed by the designers and developers. In a commercial satellite project, these milestones are of course bound to payments of the customer to finance the upcoming phase. The reviewers base their judgment over the item under review on both their understanding of quality and the project requirements that are also documented in the form of SRDs. Deviations or possible items to improve are filed as review identified discrepancies (RIDs) that need to be resolved before the project can move on [Ley et al. 2011].

Especially the often politically controversial CDR is hindering a project's agility since all subsystems have to demonstrate that their maturity level is ready for production at an unfavorable point in time. Once the CDR is passed, it is increasingly difficult to change the original design. It can be speculated that if a more agile way of shaping this period, perhaps replacing the CDR altogether with a gradual or staged process of determining the final design up until the finished product could be rewarding.

2.2.1 Lifecycle of Traditional Space Flight Software

The traditional software development process of space flight software is a V-model process like, e.g. described in the ECSS standard on project planning and implementation [ECSS Committee 2009]. It includes phases and milestones as shown in Figure 2.2.1. Figure 2.2.2 (as described by [Brown 2002]) shows the general phases such a software development process entails. After an initial planning phase, the system requirements towards the onboard software are analyzed and defined. Once all requirements are defined, the design process is started wherein a preliminary design of the overall architecture of the onboard software is determined. After this architecture is reviewed in the preliminary design review (PDR) the detailed design is worked out. Before the actual implementation of the design, a critical design review is performed [Ley et al. 2011].

Once the implementation is finished the elaborated onboard software test candidates undergo a formal qualification process which is at the end concluded by a qualification review (QR). With this fully qualified onboard software system tests including the complete satellite system and real hardware are begun. Right before launch, a formal acceptance review (AR) of all conducted system tests releases the onboard software for launch into space [Brown 2002].

ECSS describes a similar approach in their standard for space software engineering ECSS-E-ST-40C [ECSS Committee 2009]. Unlike dynamic or agile software development approach, this traditional process is static and very similar to a process applying the waterfall model (refer to section 2.1.1). The design is finished before *implementation and test* which makes modifications to the design practically impossible once formal tests have been performed, because it invalidates all previously completed *qualification and test* activities.

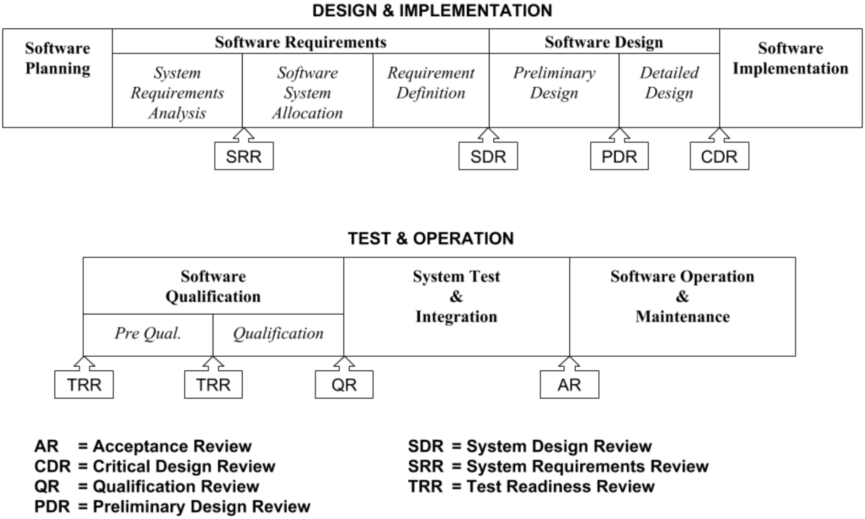

DESIGN & IMPLEMENTATION
Software Planning
Software Requirements
System Requirements Analysis
Software System Allocation
Requirement Definition
Software Design
Preliminary Design
Detailed Design
Software Implementation
SRR
SDR
PDR
CDR
TEST & OPERATION
Software Qualification
Pre Qual.
Qualification
System Test & Integration
Software Operation & Maintenance
TRR
TRR
QR
AR
AR = Acceptance Review
CDR = Critical Design Review
QR = Qualification Review
PDR = Preliminary Design Review
SDR = System Design Review
SRR = System Requirements Review
TRR = Test Readiness Review

Figure 2.2.2: Traditional onboard software design process as described by [Brown 2002]

A number of problems result from this approach. Although the final design is not yet known in early requirement definition phases, a lot of debate is already created early on due to the fact that it is difficult to change the test plan later. The planning of the project must be completed very early on basis of thin knowledge about software and spacecraft design. This static planning approach leads to a lot of test activities in later phases which are not quite necessary. It is clear that this kind of test planning consumes a lot of resources that could better be spent otherwise.

Changing requirements or plans is a complicated formal process because all of the already created documentation has to be updated along with formal review meetings as e.g. described in detail in [ECSS Committee 2009].

These complications which are basically created by decisions made too early in an inflexible environment cause project deadlines and milestones to be delayed and delayed. In order to catch up with final deadlines in the overall project timeline activities have to be cut towards the project end. The consequence of this is often the reduction of performed tests in late phases to a barely acceptable minimum although testing is valuable and should not be cut just because planning in earlier phases was insufficient.

2.2.2 Independent Software Verification and Validation

Especially for safety-critical software where high confidence in the correct functionality and stability is required, an independent validation and verification (IV&V) process is recommended. All major software development standards like IEEE, DO-178B and ISO mention IV&V as a suitable approach for safety-critical software and space agencies like NASA and ESA, recommend an IV&V process for flight software development [Hernek et al. 2008], [NASA 2006].

Independent activities after ESA include verification of technical specifications, design analysis, code analysis as well as validation of the final software including various testing techniques like black- and white-box testing and performance analysis.

IV&V in contrast to normal verification and validation (V&V) is characterized by an arrangement where a third party, usually an external contractor or a different team, performs verification and validation tasks based on documentation review and testing. The benefits of IV&V are clearly that developers can focus on developing while testers focus on testing. This does not mean that developers do not perform testing in such projects, however, the task of verification and validation is outsourced and complementary. Experts within the IV&V team can usually work much more efficiently because of their independent viewpoint and also their domain expertise. By using independent teams, it is at the same time made sure that appropriate documentation is produced to support the verification and validation activities. The downside of IV&V is that it is a costly and time-consuming process. Not only does the coordination of two independent teams require resources but also the setup of dedicated test infrastructure for the validation team has to be realized in addition to the infrastructure required by the development team.

ESA's IV&V guide [Hernek et al. 2008] states that IV&V personnel should have sufficient experience in the field of computer science and software development within the space domain and should have the mindset of a mindful, rigorous, creative destructor.

2.2.3 The Use of Simulators in Space Missions and Model-based Satellite Simulation

Simulators are widely used in all engineering domains where complex functional systems need to perform reliably. Well-known examples of the use of simulators are cockpit and flight simulators of aircraft which are used for pilot training. Power plants are simulated to train staff for emergencies. Also, the automotive industry is using hardware-in-the-loop (HIL) testbeds with engine simulators to test embedded control modules of cars.

In space technology, simulators also have a long tradition. Since the early days of spacecraft development simulators have been used and many of the success stories would have been unthinkable without simulation technology. Sophisticated simulators have been built, e.g. by NASA in the late 1960s ranging from Mercury to Apollo [Woodling et al. 1973]. Development of spacecraft simulators was continued during the 1980s in the Space Shuttle and Space Station programs [St. John et al. 1987]. While in the early days, simulators were mainly used for repeatedly training astronauts, their usefulness for other engineering applications became apparent very soon.

Nowadays, system simulation is beneficial in all phases of a space mission (see Figure 2.2.1). In the preliminary design (phase 0 & A), simulation is used to determine orbit requirements, power- and link-budgets. Starting in phase B control algorithm design is supported by simulation technology. Tools like MATLAB/Simulink® which offer a visual way to link simulation models together are used today to verify control algorithms. Once the spacecraft design is finished and assembly starts, functional verification is also supported by simulators because space conditions are difficult to establish on ground. In addition, it is also helpful for engineers to be able to perform tests on prototypes rather than the actual flight hardware. Simulation technology helps to reduce the complexity of such prototypes. Finally, during spacecraft utilization (phase E) simulation is essential to work out solutions to problems and to train operators.

Because of the increasing capabilities of computers, model-based simulation has become a key technology in modern satellite development and had a significant impact on the model philosophy for spacecraft design. The idea is to replace costly spacecraft engineering models with software digital twins. In a traditional model-based simulation, the onboard computer which is the core component is either a software model itself or connected to the simulator. This way the functional behavior of the spacecraft and its onboard equipment in interaction with the space environment is realistically reproduced.

The benefits of such an approach are manifold. First, it enables early verification activities that would be costly or impossible to perform with real hardware in ground tests. Second, it helps engineers during early design activities. Software tests can begin earlier because no actual equipment is required to verify, e.g. control algorithms. Third, simulation saves resources because simulation infrastructure can be reused between missions in contrast to engineering model hardware. And finally, because new software simulators can be spawned as required with relatively low-cost testing activities can easily be scaled based on demand without blocking the main spacecraft [Eickhoff 2009].

In the late 1990s, the European space industry started to use satellite simulators as a design tool and also to verify onboard software and flight procedures in closed-loop simulations. The first installment of this endeavor was the System Simulation & Verification Facility (SSVF) which started as a study conducted by ESA and was later used as the system testbed (STB) for the Grace project by NASA as a demonstrator mission [Irvine et al. 2002]. SSVF was written in FORTRAN and was soon replaced by an object-oriented satellite simulator, the model-based development and verification environment (MDVE), developed by the former Astrium GmbH, which is now part of Airbus Defense and Space, and was successfully used in ESA's CryoSat mission [Hendricks et al. 2005], [Eickhoff 2009].

Because of the increasing importance of model-based simulation, the European Space Agency has proposed a standard for a unified simulation modeling platform (SMP) which is based on the simulation model portability standard (SMP1 and SMP2) to create simulation models that are exchangeable between missions [ECSS Committee 2011].

2.2.4 Methods of Spacecraft Test Automation

In this section methods for the automation of test procedures for spacecraft are described. These methods serve two purposes.

- Enable the execution of automatic test procedures during ground tests and
- Enable automation for mission operations.

The automation of tests is the capability that is of relevance in the context of this thesis.

Automated test procedures work from the standpoint of the spacecraft operator. The flight software in this context is regarded as a black box system. A suitable solution for test automation has to provide the possibility to programmatically send commands to the spacecraft and evaluate telemetry, i.e. base decisions on events and telemetry parameters. This software which acts as a human operator and executes test procedures programmatically is called *test agent* or *procedure execution engine*.

2.2.4.1 Mission Control Systems

A mission control system (MCS) is a special software that enables sending commands to the satellite and display telemetry parameters and events to the operator. It is the user interface of the operator to the spacecraft. Typically the mission control system and its mission database, which contains all mission parameters, commands, etc. are set up during the spacecraft development and test phase (phases C and D).

The mission control system already plays an important role in system testing. Sometimes, a different checkout system than the original mission control system is used for tests by the manufacturer. The ideal solution, however, is to use the same mission control system for all operations with the spacecraft. This way, the original system receives more testing hours and is thus more mature.

Most mission control systems have limited capabilities when it comes to the automation of tasks that are normally performed by the operator. Such tasks include selecting commands, populating commands with appropriate command parameters, reacting to onboard events, and monitoring telemetry. Normally operators compile stacks of pre-populated commands which are then saved for repeated usage.

To better automate tests and to be able to create test scripts, third-party solutions are often required depending on which mission control system is used. Many solutions exist to date but not all of them are freely available and a lot of the solutions are either project or agency-specific and not very well suited for general application across missions. Most solutions focus on the automation of one specific mission control system, like e.g. SCOS-2000[8], and are specially tailored to interact with that specific system.

2.2.4.2 Spacecraft Operations Automation and Script Languages

Flight procedure editors, like e.g. the Manufacturing and Operations Information System (MOIS) [Pearson et al. 2012], [Chaudhri et al. 2006], are software applications that can be used to create and execute operations procedures. They are capable of remotely sending commands and basing decisions on telemetry which is, e.g., transferred over SCOS-2000 external interfaces.

The Procedure Language for Users in Test and Operations (PLUTO) is a spacecraft operations language that was proposed by ESA [Demeuse et al. 1998]. In an effort to standardize ground systems automation, [Demeuse et al. 1998] describes PLUTO as "a *simple* but powerful language adapted to the test and operations of spacecraft as well as a *common* language for system test and operations but also for unit/subsystem integration and functional tests to be performed during the AIV (AIT) phase of the spacecraft developer" [Demeuse et al. 1998]. Unfortunately, the adoption of the PLUTO language besides ESA projects is minimal.

Another scripting language that was designed to perform spacecraft operations is the Spacecraft Test and Operations Language (STOL). STOL is more common in NASA missions and its specifications were published as early as 1978 [Desjardins et al. 1978].

In an informal survey, [Chaudhri et al. 2006] found out that 18 spacecraft manufacturers use more than 19 different, unique languages or tools for spacecraft operations and test automation. This is a problem because that way, the application of each of the many solutions is scarce and development and proper maintenance of each language is almost impossible. It would rather benefit the whole industry if a common standard spacecraft operations language were adopted.

[8] See www.esa.int/Our_Activities/Operations/gse/SCOS-2000

It is worth mentioning that the programming language Python has received popularity in recent years and a few companies like, e.g. GMV have begun adopting Python as the preferred scripting language for all kinds of operational tasks [Garcia 2008].

Depending on individual mission requirements, each project needs to decide whether a general-purpose solution like automation using scripting languages like Python or PLUTO or a commercial software product like MOIS is better suited for the mission. Commercial products offer a wider range of features that are more advanced than plain procedure execution. E.g., they also include other aspects of quality assurance like test report generation, versioning and validation of procedures. In a more simple mission where the focus lies on efficiency and where staff-hours are limited, a general-purpose solution which only offers scripted control over the mission control system might be better suited.

2.3 Software Testing

Software plays an increasing role in modern engineering projects. Almost all technical solutions use software to achieve specific functionalities in the final product. If this software is invisible to the end-user and runs without direct human interaction, the software is called embedded software. A valid distinction when talking about testing is the distinction between software tests and product tests. Product tests are tests conducted with the entire product including all of its (embedded) software. Software tests, on the other hand, are tests exclusively focusing on the design and implementation of the software itself and often do not even require the original target computer because they are executed on a dedicated test system.

Software testing is a quality assurance discipline. The goals of software quality assurance according to ISO/IEC 9126 (cf. [Rau 2016] and [Balzert 2011]) are to increase:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

At first glance, software testing does not sound too difficult. One would naively create a set of test cases, let the software run these test cases and measure the output. If the output satisfies the specifications, the test is passed and one can move on to the next test case. In reality, proper testing is much more difficult.

2.3.1 Definition of Software Tests

Testing is a discipline that is performed in all areas of engineering. Whenever a product is developed, the design is the blueprint for later manufactured specimens of that product. A test is essentially an experiment and ideally follows scientific principles, i.e. it shall observe the cause and effect relationship of an action expressed as a hypothesis and shall be repeatable. In that regard before a test is conducted, an expected result should be formulated to be able to assess the outcome of the test. During the process of design and manufacturing different kinds of tests are performed with different goals in mind. These different kinds of tests are:

- Testing a design against specifications to make sure that the elaborated design meets all requirements and legal regulations
- Testing a prototype to provide insight into feasibility and technology readiness
- Testing every single manufactured specimen of a series (in case of software the specimen is the release) to show that the single item has no deficiencies
- Testing usability to improve future versions of a design [Rubin et al. 2008]

On a more abstract level, testing can be described as the process of searching for defects in either design or manufacturing through the usage or operation of the product. While manufacturing defects only affect one specimen, design defects appear in all specimen alike [Kaner 2008].

These two faces of testing are often referred to as verification and validation. But in stark contrast to the definitions given by [IEEE 1998] as mentioned in section 2.1.1 ECSS describes validation as the "...process to confirm that the requirements baseline functions and performances are correctly and completely implemented in the final product" [ECSS Committee 2017] and verification as the "...process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input" [ECSS Committee 2017]. The discussion about the meaning and application of these terms will not be pursued during the course of this thesis.

It might seem like a contradiction to view both terms together, but they are inherently just two sides of one medal which is to increase the quality of a product.

Also, in some rare projects where only one instance of a design is produced, e.g. research satellites, the distinction between testing for defects introduced during production and design defects is unnecessary from an engineering standpoint.

As a final remark, in the area of software development, the term testing is often confused with the term debugging. While testing comprises all facets described in this chapter, debugging is the manual process of stepping through software code while it is executed to find programming errors. If during a test a problem is found, it is often the next step for the developer to use debugging techniques to identify what line of code is causing the problem. Without debugging tools, this process of pinning down the cause of the problem would be very hard.

2.3.1.1 Test Plan, Test Cases and Test Procedures

Before testing begins, a test plan is usually created. A test plan summarizes all selected test cases for a given test target without further specifying the details of a test case. The main purpose of creating a test plan is to enable people to review the pending test activities and to provide an overview to judge test coverage and progress.

A test case is a set of inputs used as boundary conditions for a test run. The tester who creates a test case has a specific property or functionality in mind which the test should focus on. The creator of the test case formulates an expected outcome for that set of inputs to assess the result of the test run [Myers et al. 2011].

While exploring all possible inputs is impractical in most cases, it is advisable to create categories of test cases and to combine a multitude of similar test cases into one representative equivalence class. This way, a few highly contrasting test cases belonging to different equivalence classes can be identified. Equivalence classes are usually used to select the correct test cases to achieve as much diversity in test inputs as possible. The assumption, which is made when reducing the overall amount of test cases to representatives of equivalence classes is that the results of similar test cases are likely to produce a comparable result whereas contrasting test cases from different equivalence classes are more likely to uncover defects [Myers et al. 2011].

A test procedure is a detailed description of the required steps to complete a test run. Test procedures ensure the repeatability of a test run by formalizing the way in which the test is performed. Test procedures are often written down in documents which are also used to review the testing activity [Myers et al. 2011].

2.3.1.2 Formal Verification

Ever since computer programs were used to control critical processes, computer scientists were searching for ways to formally prove their correctness. Formal verification in the context of computer systems and software is the attempt to finding a formal, i.e. mathematical, proof that a computer program or algorithm is correct with respect to a formal specification [NASA 1995].

In computer science, formal verification is a topic of continued research and it finds wide application in the computer hardware industry. Formal verification for software, however, is rarely seen except for some examples like compilers and very basic operating systems. PikeOS e.g. is a formally verified real-time OS. PikeOS was formally verified using a special compiler that features built-in formal verification and uses specifications also expressed in the C programming language [Baumann et al. 2010].

In general, the technical specifications of a spacecraft flight software as a whole are too diverse to describe them in a way that would be meaningful to be able to provide a formal proof. Some isolated software parts or algorithms, however, are formally proven in industrial applications primarily in the safety-critical domain, but the scope of these small software parts is minimal and not comparable to a full embedded application [Pan 1999]. Because of the above-mentioned limitations, formal verification cannot be applied in the context of this thesis and shall therefore not be discussed any further here.

2.3.2 Different Levels of Testing

Software testing is performed on several different levels. The standard distinction is made between (cf. [Freeman et al. 2010]):

- **Unit level**: Object-level testing (e.g., do all methods of a class provide the correct results for a given set of inputs)

- **Integration level**: System-level testing (e.g., do objects correctly interact with each other and also with external libraries or dependencies)
- **Acceptance level**: Black box, system-level testing (e.g., does the entire software do what it is supposed to do from a user perspective)

Unit tests are tests that cover small pieces of software like functions, classes or modules. Unit tests are dynamic tests. That means that the source code under test is compiled and executed (in contrast to static methods like static code analysis, see section 2.3.4). Traditionally unit tests are written and executed by the programmer in the same programming language as the program to gain immediate feedback about his or her work [Linz 2017].

If a unit has external interfaces and interacts with other objects by calling their methods it is difficult to check if these methods are called throughout a unit test because ideally, the unit test runs in an isolated environment without the presence of other units. To solve this problem, mock objects are used to provide stub objects that serve as interaction partners for a unit under test but which do not represent the real objects with all implementation details [Freeman et al. 2010].

Integration tests are somewhere in between unit and acceptance tests. They test the interactions of different objects or modules of a larger software or the surrounding environment [Freeman et al. 2010]. E.g., an application might have a distributed architecture consisting of a client and a server side. The integration test would make sure that functionalities like the establishment of a connection, data exchange, or command and control are working as intended when operated together.

While unit and integration tests ensure that the classes are well structured and the software is of good internal quality, acceptance tests aim to make sure that the software meets the end user's requirements but also to make sure that no existing features are broken due to changes to the code (cf. [Freeman et al. 2010]). Acceptance tests are performed end-to-end which means that interactions with the system are only performed from outside and results are gathered by means available to the user.

Because the different levels of testing build upon each other they are usually executed in the order given above, i.e. first unit tests are performed, followed by integration tests and at the end acceptance tests are the gatekeeper for the official release to the user.

2.3.3 Regression Testing

Software development is normally based on continuous release cycles. New features are constantly added to an already existing codebase. In parallel, bugs are fixed and these bug fixes are also introduced to the codebase. Both new features and bug fixes get released to the user by means of software updates. Before a new version of a software product is released, it is recommended to re-run all or a subset of all tests which previous versions have already undergone to find bugs that were introduced together with the new changes.

Bugs that affect a feature that had already been tested successfully in previous releases but now does not function properly anymore because of changes to other software parts are called regressions.

Software testers have two options to deal with regressions. Either they run the full set of tests again which can be time-consuming or they run a subset of these tests which are thus called regression tests.

Because regression tests have to be repeated over and over again for each new software release, regression tests are often implemented in the form of automated test procedures which are expected to produce the same result no matter which software version is tested [Liggesmeyer 2009].

2.3.4 Static Analysis Methods

Finding problems with a program's source code does not always require executing the program itself. Problems can already be identified by reviewing the source code prior to compiling it. This class of methods which reasons about the source code is called *static methods* in contrast to *dynamic methods* which mainly imply testing and running the compiled program. Static code analysis can be performed either manually in the form of peer reviews or automatically by dedicated software tools.

Static code analysis tools parse the source code and try to identify problematic patterns and suspicious operations. This requires the tool to be able to identify variable type conversion errors, problematic pointer usage, syntactic errors, division by zero possibilities and so on. In addition, static code analysis tools can be used to enforce a specific style of code formatting convention like the arrangement of brackets, variable naming or the use of spaces instead of tab indentations.

A wide variety of static code analysis tools are available for almost all popular programming languages. Some of the more advanced tools are only available commercially. The capabilities of these tools also vary from basic functionalities like highlighting problematic lines of code up to symbolic code execution and analyzing race conditions [Louridas 2006]. The usage of static code analysis tools other than the ones already shipped with the used IDE/editor is not adopted very well among software developers. As the Google white-paper Johnson et al. [Johnson et al. 2013] suggests, the reason for this underuse could be the abundance of false-positive warnings and the way results are displayed in general. Johnson [Johnson et al. 2013] suggests that a more interactive way of displaying the findings could benefit static code analysis tools. Nevertheless, the use of these tools within software development projects as well as performing code review amongst peers is highly recommended and should be made a mandatory part of a general software development workflow.

2.3.5 Continuous Integration Tests

Modern software development teams use continuous integration tools to automatically build, test and release new versions of their software. Continuous integration (CI) means that all changes made by developers are merged automatically by a chain of build tools whenever new code is committed. Likewise, unit, integration and regression tests are automatically run for each change on continuous integration servers to ensure that the committed change does not cause any problems.

In recent years CI was extended by adding continuous deployment pipelines to the already existing CI infrastructure. Continuous deployment ships the automatically built software update directly to the user’s installation. Around continuous deployment pipelines, whole teams of software engineers were created whose main job is to keep this infrastructure up and running, the so-called DevOps movement [Bass et al. 2015]. The incentive for software companies to afford such costly continuous integration and deployment pipelines is to reduce the time it takes for new features and bug fixes to reach the end-user.

> Software companies have learned that applying any manual scheme is way more error-prone and, in the long run, more expensive to maintain than paying the initial cost of setting up continuous integration and delivery.

When developers submit code that fails a test, the code will not be integrated into the new release. In fact, most continuous build systems stop further processing if any test has failed until the cause of the problem is fixed. In large teams, it is often hard to identify which particular change caused a test to fail. This problem can be diminished by a good test design.

2.3.6 Exploratory Testing

Exploratory testing is inspired by testing strategies applied for computer games in which large numbers of beta testers play the game in the same way a genuine player would play it, exploring the game world, taking different paths to achieve certain goals. Additionally, beta testers are encouraged to explore aspects of the game which diverge from the most obvious path taken by most players to find edge cases that the developers did not think about, or were unaware of, during their own testing activities. Translated to the domain of spacecraft flight software, exploratory testing would mean to test the software as a black-box system in simulation sessions from the viewpoint of an operator while performing various tasks of daily mission operations. The tester uses his or her expert knowledge to come up with test cases in a creative way, following his or her instincts.

The term exploratory testing was introduced by Cem Kaner who defines exploratory testing as: "[...] a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the value of her work by treating test-related learning, test design, test execution and test result interpretation as mutually supportive activities that run in parallel throughout the project." [Kaner 2008]

The tester should be encouraged to find as many bugs in a program as possible thus this test activity is often compared to the hunt for bugs.

2.3.7 Software Quality Metrics

To be able to judge the quality of software, various metrics can be used. Implicitly, measuring how many defects are included in a program produces a metric of the quality of the software for how well it provides the functionalities that a user expects.

To be able to rate the software, all of these methods need to classify the complexity of the analyzed software somehow because more complex software tends to have more defects.

The practice of determining the complexity or the size of the software is often referred to as sizing.

One way to determine the complexity of the software is to simply count the effective lines of code (SLOC). Software sizing by effective lines of code alone, however, is widely criticized among experts because other factors also influence complexity such as design choices or the pure amount of included functionality. More so, the lines of code can drastically vary between similar applications depending on which programming language is used [Rosenberg 1997].

Other metrics to express the software size have been proposed, such as function points, object points, weighed micro functions, Halstead complexity measures, etc. Function points are the measure that is mostly applied by industry. They estimate software size by the functionality they provide based on design and specifications.

The number of discovered defects per software size is the defect density of software. The defect density can be used to track software quality over time or to assess the overall quality of the software development effort. The defect potential is the expected defect density for a to be developed software based on industry statistics [Jones 2008].

Another useful metric is the defect removal efficiency which is defined as the ratio of the number of defects found and solved before shipping of the software and the total number of defects. The standard time period for calculating defect removal efficiency includes 90 days after shipping [Jones 2008] and [Poensgen et al. 2012].

The code coverage rate is another metric that gives insight into how many lines of code or decision paths have been executed during a test run. Code coverage analysis is helpful in many ways. First, it helps to uncover unused parts of a program (dead code), logical paths that a program did not go through or functions that have not been called. Furthermore, it makes it possible to compare the quality of different test procedures and measure the overall effectiveness of testing activities, in unit tests e.g. Code coverage analysis gives a measurable, quantitative way to express test success. In critical software development such as aerospace, automotive or medicinal applications it is often required to prove a certain rate of code coverage for tests. These requirements usually come from either customers or agencies which have strict regulations towards providing proof of the suitability of the software before it is permitted to be used [Grünfelder 2017].

3 Flying Laptop System Description

This chapter gives an overview of the *Flying Laptop* satellite system which is the case study of the proposed approach. Section 3.1 describes the applied development process for *Flying Laptop* and explains why certain unusual aspects were adopted for the development of the *Flying Laptop* onboard software. It provides the context in which the project was conducted which is important to understand the environment under which this work took place. Section 3.2 provides an overview of the essential subsystems and design of the *Flying Laptop*. Section 3.3 describes the *Flying Laptop* onboard software architecture and section 3.4 describes the system testbed.

3.1 The Flying Laptop Project

It took the *Flying Laptop* project a long time from its initiation until the satellite was finally assembled and launched. Multiple design iterations and the installation of infrastructures like a vacuum chamber for thermal tests of equipment, a clean room for integrating the satellite and an antenna dish for operating satellites at the University contributed to the extended project lifetime.

After first initial ideas in 2004, a feasibility study was performed together with industry experts in the year 2005 [Roeser et al. 2005] [Schoenermark et al. 2005]. The initial approach was to follow a traditional development and verification process as described by the ECCS standard for project planning and implementation [ECSS Committee 2009] (see also section 2.2.1). A first design was presented in 2009 to a board of industry experts in a preliminary design review (PDR). The first design featured an FPGA based onboard computer. However, the review board came to the conclusion that the plan to perform all data handling tasks using the FPGA based onboard computer, while still providing capabilities like advanced onboard services and failure detection and handling, was too ambitious for the presented FPGA based onboard computer design. Thus, a brand new microprocessor-based onboard computer design was developed and the former FPGA based onboard computer was repurposed to perform payload data handling only.

The new onboard computer was available in 2012 [Eickhoff 2013] and the first tests with a bare version of the new *Flying Laptop* flight software written in C++ could be performed.

The team had to shift their development efforts which before consisted of programming an FPGA to developing an embedded flight software application written in C++.

In addition to the delay imposed by the design change of the onboard computer, some components of the satellite were procured as commercial off-the-shelf (COTS) parts and were already available while others were still developed in-house. In addition, all components were facing time-consuming qualification tests, a task which could also not be streamlined well. Because of these discrepancies, there was a broad spread of maturity between the different parts of the system.

The spread of maturity between different parts of the *Flying Laptop* system caused difficulties in project management and the team decided to leave the traditional requirement-based development approach behind and to adopt a more flexible development methodology. One that allowed for different development speeds within the project so that the evolutions applied to the design could be merged into the already existing system.

This approach also allowed for changes to the design which were necessary even late in the project to be able to react to problems that had to be expected but were not yet visible.

Using a traditional ECSS/requirements based management approach (as described in section 2.1.1), this degree of flexibility would not have been possible.

Naturally, changing the development approach from a requirements-based approach to an agile one without verifiable requirements implied also to change the verification strategy from requirements-based testing to a more agile testing approach. This approach is described in further detail in chapter 4.

There was no SRD for the *Flying Laptop* design. Of course, requirements were present in the form of known functionalities/features which the system should support, but there was no process of tracking these requirements formally. Other requirements were imposed by external stakeholders like the launch provider or regulations formulated by law (e.g., the legal framework described in [Kries et al. 2002]) which needed to be respected. As a result, there was no way to track and link requirements to test cases. The required test cases for functional testing were determined through dialogue among the different subsystem architects of the team.

Through discussion, the team settled on a small number of core attributes, features, and functionalities that the system should provide. These attributes, features, and functionalities were intentionally not called requirements because it shall be clear (also in this work) that the approach differs from traditional verification according to the waterfall model where requirements are normally tracked in SRD documents. For that reason from now on in this thesis, the term requirement(s) will be avoided. A list of functionalities of the *Flying Laptop* flight software can be found in [Bätz 2018].

3.2 System Overview

The *Flying Laptop* satellite consists of the satellite bus and payload. The purpose of the satellite bus is to provide power, attitude control and thermal control for the payload instruments. In addition, the satellite bus is responsible for ground communications. *Flying Laptop* uses a passive thermal control system (TCS) to provide suitable thermal conditions for the satellite equipment. Power is generated by three solar panels mounted on the back-side of the satellite structure and two deployable side wings. The energy is stored in a battery consisting of lithium iron phosphate (LiFePO4) cells.

The *Flying Laptop* satellite bus is designed to resist one failure in any equipment by redundancy. A combination of hot and cold redundant components is used as required for each subsystem.

At the core, the patented combined data and power infrastructure (CDPI, see [Eickhoff 2013]) is responsible for data handling and power distribution. The CDPI consists of the *Flying Laptop* onboard computer (OBC) and the power control and distribution unit (PCDU). Onboard data handling is performed by the OBC which runs the *Flying Laptop* onboard software (OBSW). The abbreviation OBC and OBSW will be used from now on in this work when the *Flying Laptop* specific onboard computer and the flight software of *Flying Laptop* is meant without any further explanation.

Payload instruments are controlled by the OBC via a dedicated payload onboard computer (PLOC). The PLOC communicates directly with each payload and serves as a data management device. For the downlink of payload data, a dedicated S-band data downlink system (DDS) is used. The PLOC can either forward payload data directly to the DDS transmitter for downlink or store larger amounts of payload data in the mass memory unit (MMU) for later downlink [Hagel 2018].

3.2.1 The Flying Laptop Onboard Computer

The development of the OBC was a challenging task for the *Flying Laptop* team members and industry partners because of budget and time constraints. The reasons for the fresh development of an onboard computer for *Flying Laptop* instead of relying on the available flight-proven commercial off-the-shelf solutions are that these solutions would have been too expensive and also too large for the small satellite to accommodate. Also, available low-cost solutions as used in CubeSats and other University satellites did not fulfill the requirements towards professional onboard and ground communication standards such as CCSDS. Requirements like radiation hardness and the capability to run a real-time operating system ruled out all available commercial solutions at that time [Eickhoff 2013].

The final design features a modularized stack of Eurocard[9] format printed circuit boards (PCBs) where each board is available twice for redundancy. An overview of the electrical interfaces is given in Figure 3.2.1.

The two core CPU boards are single-board computers equipped with a UT699 LEON3-FT processor, memory and various general-purpose I/O ports (GPIO). For communication with all external satellite components such as bus equipment and payloads, two separate I/O boards were added to the stack. Most onboard devices are connected to OBC via the I/O board. The I/O boards buffer device communication messages in various protocols for timely read and write by the OBSW. Special purpose CCSDS boards encode telemetry and decode commands according to the CCSDS communication protocol for ground communication and are connected directly to the onboard radio system. All boards are cross-connected for redundancy via internal SpaceWire connections over an internal OBC harness [Eickhoff 2013].

[9] Eurocard is a European standardized format for printed circuit boards

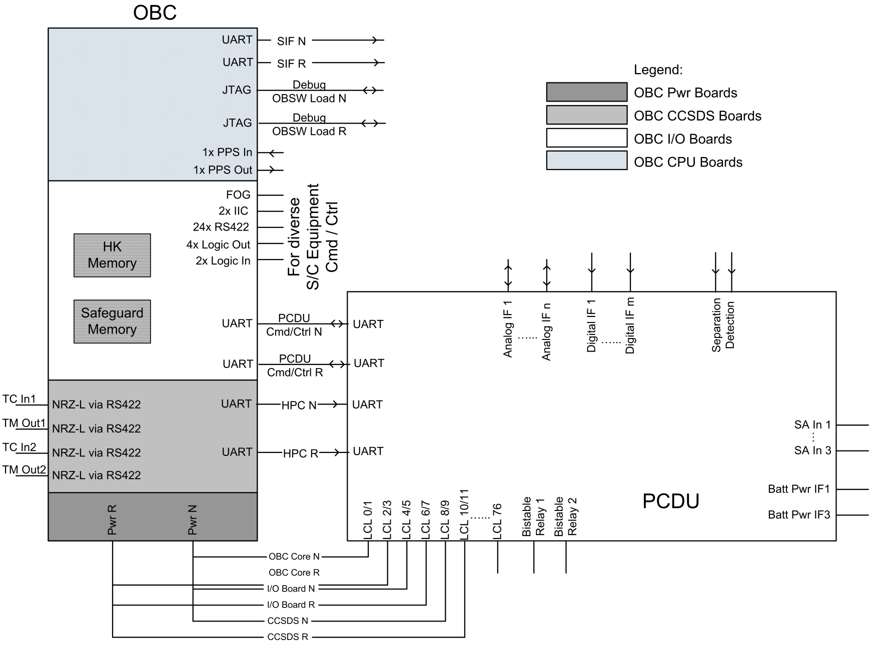


Figure 3.2.1: The CDPI electrical block diagram and interface architecture [Eickhoff 2013]

Within the CDPI design, some fundamental functionalities which usually are the OBCs responsibility have been outsourced into the PCDU. The PCDU is a multipurpose device that fulfills various functionalities like power supply of each equipment, battery charging and monitoring but also analog/digital (A/D) conversion of analog sensor signals like temperature sensors and sun sensors. In a classic onboard computer design, such conversion of analog signals would be performed by a remote interface unit (RIU). At the same time, the PCDU serves as a common reconfiguration controller. A reconfiguration controller realizes a watchdog functionality and triggers failover switching if one of the redundant OBC boards fails [Eickhoff 2013].

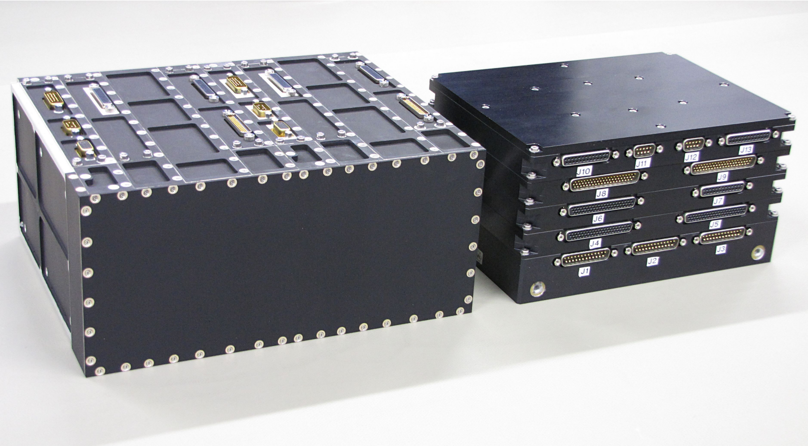


Figure 3.2.2: OBC stack and PCDU forming the CDPI, © IRS, University of Stuttgart - from [Eickhoff 2013]

3.2.2 Attitude Control

The attitude control subsystem of the *Flying Laptop* supports different modes of operation. For Earth observation and communication with ground stations, fine pointing modes are used which utilize the full stack of available attitude control sensors and actuators such as reaction wheels, GPS receivers and star trackers.

Figure 3.2.3 shows all basic attitude control modes of *Flying Laptop*. Nominally, when performing payload operations and communication tasks, *Flying Laptop* operates in one of the pointing modes (3), (4) or (5). *Idle* mode (2) most of the time to restore battery charge. *Idle* mode is designed to orient the satellite's solar panels towards the Sun using reaction wheels. Also, the attitude control subsystem offers a *Safe* mode (1) which is automatically activated if for any reason higher modes cannot be maintained. The detumbling (0) as shown in Figure 3.2.3 is a special control strategy of *Safe* mode where high rates are first reduced using magnetic torque.

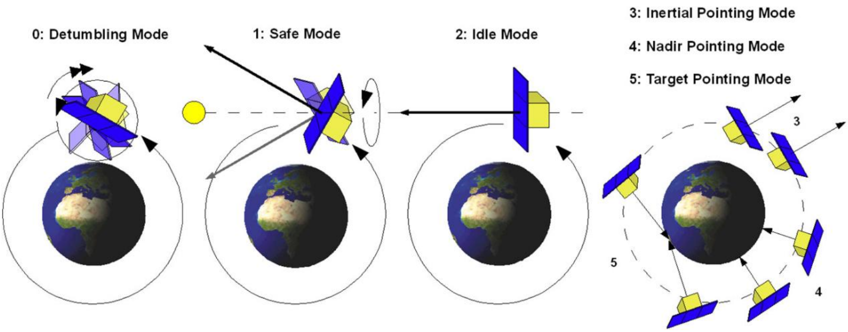


Figure 3.2.3: Different attitude control modes of Flying Laptop supported by the OBSW, © IRS, University of Stuttgart

The accuracy requirements for fine pointing derived from various payload operation scenarios lie in the region of 150 arc seconds. Especially the experimental optical communication terminal requires high pointing accuracy. To achieve this degree of accuracy, the attitude control subsystem needs both good pointing knowledge which is provided by the star tracker camera system but also precise knowledge about the satellite's current position and velocity. For this purpose, the GPS receivers provide not only position and velocity solutions through GPS but also precise onboard time.

3.3 Flying Laptop Onboard Software

The *Flying Laptop OBSW* is divided into a mission-specific part and a part that is reusable between different missions, called framework. The framework part, in this case, is called Flight Software Framework (FSFW) and provides common functionalities required by different types of space missions.

For testing purposes within this thesis, the distinction between framework and mission-specific code is irrelevant since the separation was introduced after the completion of the *Flying Laptop OBSW*. For this reason, from now on the term OBSW refers to the *Flying Laptop OBSW* as a whole.

Most flight software in use today share common attributes and building blocks and the *Flying Laptop OBSW* is no difference. While the implementation details may vary, the approach described in this thesis is suitable for a lot of other flight software solutions which gives the approach more general relevance. A space onboard software reference architecture has been proposed by a European industrial working group in [Terraillon et al. 2010]. The *Flying Laptop OBSW* follows the key aspects of that reference architecture which is described in detail in [Bätz 2018].

3.3.1 Basic Architecture of the Flying Laptop Onboard Software

Figure 3.3.1 depicts an overview of the static architecture of the OBSW. The OBSW is an embedded object-oriented software based on a real-time operating system (RTOS) which is exchangeable by application of the POSIX[10] standard. The interfaces of the used RTOS are wrapped through the operating system abstraction layer (OSAL) provided by the FSFW so that the OBSW developer has an abstract interface to work with which is independent of the used RTOS. The RTOS used for *Flying Laptop OBSW* is the Real-time Operating System for Multiprocessor Systems (RTEMS).

Elements of the object-oriented OBSW are called components. Within this work, the term component refers to an OBSW component. The component framework of the FSFW provides the means to build OBSW components. Components use inter-process communication, i.e. message queues and shared resources (data-pool) to exchange information. In principle and as implemented in the FSFW, there are four types of components:

- (1) **Device handlers**, which are responsible for communicating and controlling onboard devices
- (2) **Controllers** are responsible for sensor monitoring, sensor fusion, execution of control laws and generation of actuator commands
- (3) **Subsystems** and **Assemblies** group associated components into meaningful arrangements and manage the modes of these associated components
- (4) **Ground services** provide services to operators to interact with the OBSW

[10] The Portable Operating System Interface (POSIX) is a standard which aims to maintain compatibility between different Unix based operating systems.

To associate these component types in the OBSW architecture, the numbers (1) through (4) are also visible in Figure 3.3.1. This is meant to clarify where these component types are used in the overall OBSW design.

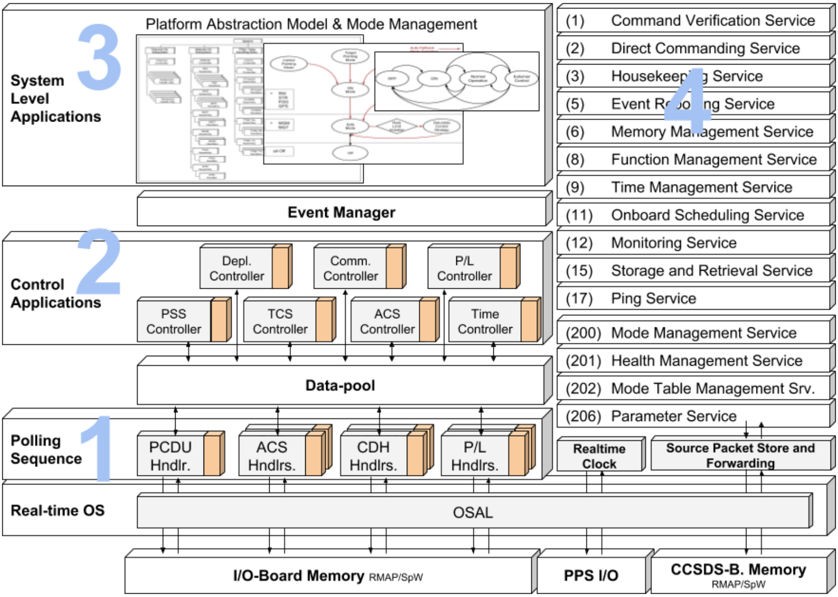

3
System Level Applications
Platform Abstraction Model & Mode Management
Event Manager
2
Control Applications
Depl. Controller
Comm. Controller
P/L Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
Data-pool
1
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW
4
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(8) Function Management Service
(9) Time Management Service
(11) Onboard Scheduling Service
(12) Monitoring Service
(15) Storage and Retrieval Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service
Realtime Clock
Source Packet Store and Forwarding

Figure 3.3.1: Internal architecture of the Flying Laptop OBSW

In addition to components, the FSFW features so-called building blocks that can be used to perform generic tasks which are common to many OBSW implementations, like e.g. parameter monitoring, reporting and failure isolation [Bätz 2018].

To execute the logic of each component periodically, the components are assigned to specific periodic tasks that run concurrently using rate monotonic scheduling. There are 27 periodic tasks defined in total for the final OBSW version. The most noteworthy among these 27 periodic tasks are responsible for executing the device polling sequence, ground services, source packet store and forward, various control applications and mode logic processing in the form of assemblies and subsystems.

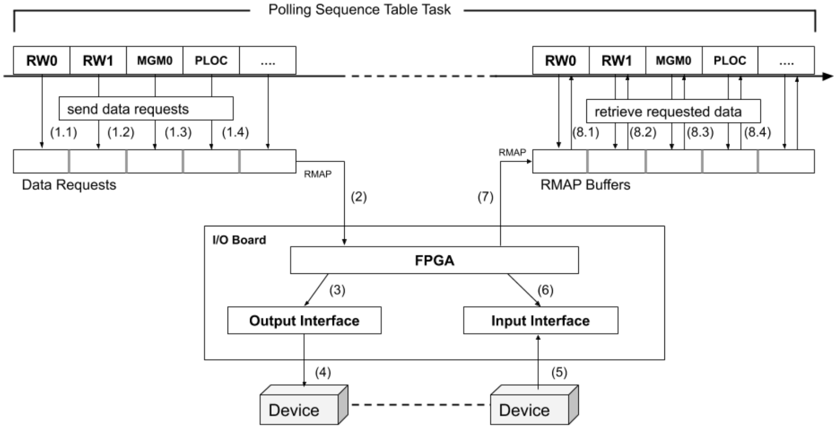


Figure 3.3.2: Example of how the polling sequence table task triggers I/O activity for devices RW0, RW1, MGM0 and PLOC - originally from [Witt 2016] (modified)

The periodic polling sequence task is responsible for orchestrating device communication. The polling sequence table defines when in the execution cycle each device handler is scheduled to communicate with its assigned device. An execution cycle as displayed in Figure 3.3.2 repeats with a fixed time interval. That means that all device communication will be performed on the basis of this interval. This allows for precise control over the timing of device communication. In the first parts of this cycle time slots for data requests (1.1) to (1.4) of each device are predefined. From then on the data request is sent to the device and the device reply is stored in the receive buffers of the I/O board, steps (2) to (7). After a fixed time period, this reply data is read by the device handlers and forwarded to the internal data-pool (8.1) to (8.4) [Bätz 2018] [Witt 2016].

3.3.2 TM/TC and Ground Services

The OBSW supports space communications standards like CCSDS and ECSS. Most prominently, the ground systems and operations - telemetry and telecommand packet utilization standard (PUS) defined in [ECSS Committee 2003] describes the use of CCSDS source packets to communicate with a spacecraft over a space link. The CCSDS only provides a concept of how to perform source packet-based communication but does not provide specific implementation details.

Table 3.3.1: TM/TC ground services implemented in FSFW and OBSW according to [Bätz 2018]

Service Type	Service Name	Functional Component	Defined by
1	Command Verification Service	standalone	ECSS (Agency defined)
2	Direct Device Commanding Service	gateway	ECSS (Agency defined)
3	Housekeeping Service	standalone	ECSS (Agency defined)
5	Event Reporting Service	standalone	ECSS (Agency defined)
6	Memory Management Service	gateway	ECSS (Agency defined)
8	Function Management Service	gateway	ECSS (Agency defined)
9	On-board Time Management Service	standalone	ECSS (Agency defined)
11	Onboard Scheduling Service	standalone	ECSS (Agency defined)
12	Monitoring Service	standalone	ECSS (Agency defined)
15	TM Storage and Retrieval	gateway	ECSS (Agency defined)
17	Test Service (Ping)	standalone	ECSS (Agency defined)
200	Mode Management Service	gateway	FSFW (Custom)
201	Health Management Service	gateway	FSFW (Custom)
202	Mode Table Management Service (planned)	gateway	FSFW (Custom)
206	Config. Parameter Management Service	gateway	FSFW (Custom)

According to ECSS PUS, the onboard service instances are identified by reserved service type numbers. Table 3.3.1 provides a list of TM/TC ground services that are implemented by the OBSW. They represent the same services as already shown in Figure 3.3.1 (4). In addition to the reserved standard PUS services types defined by the packet utilization standard itself whose type identifiers range from 1 to 127, additional mission-specific ground services have been defined for the FSFW.

FSFW service components can also be characterized after their functional mode of operation. Standalone service components process commands themselves and provide a direct service to the operator. In contrast, gateway service components merely translate and forward commands to other OBSW components, e.g. the direct device commanding service can be used to command a device handler component to interact with an onboard device more directly [Bätz 2018].

3.3.3 The Platform Abstraction Model, Health- and Mode-Tree

The *platform abstraction model* (3) in Figure 3.3.1 represents the arrangement of the satellite system's equipment and subsystems in a tree-like structure. On the lowest level of the *platform abstraction model* are the device handlers. As already mentioned in section 3.3.1, each onboard equipment has a device handler associated which controls and monitors the equipment. That means that if a device handler is commanded from *Off* to *On*, it implicitly commands the PCDU to switch on the associated device's power switch through the PCDU device handler.

The collection of all states is reflected in the mode tree formed by the *platform abstraction model* (see Figure 3.3.3). Higher-level objects use mode tables which define the modes of all underlying OBSW components. If a higher-level component is commanded to transition into a specific mode the component will try to change the modes of its children accordingly. Through this mechanism, a mode change of a higher-level object automatically triggers mode changes of lower level components and traverses through the lower level part of the mode tree. A mode command to the system component, the highest level component, will affect the entire mode-tree and thus ensures a consistent state of all subcomponents.

When under *External Control* (see Figure 3.3.4), the component disregards the commands from its parent component and can be checked out undisturbed by an operator. *External Control* is especially useful for testing purposes where the capability to override mode-logic is usually required.

The structure of the *Flying Laptop platform abstraction model* is displayed in Figure 3.3.3. By creating this hierarchical structure of device handlers, assemblies, subsystems and controllers, the platform abstraction represents the current health and mode state of the entire system. Note that in Figure 3.3.3 controller components are placed directly under the subsystem components on the same level as assemblies. Just like for any other component, the mode of a controller determines its operational state (e.g., on, off, and control strategy).

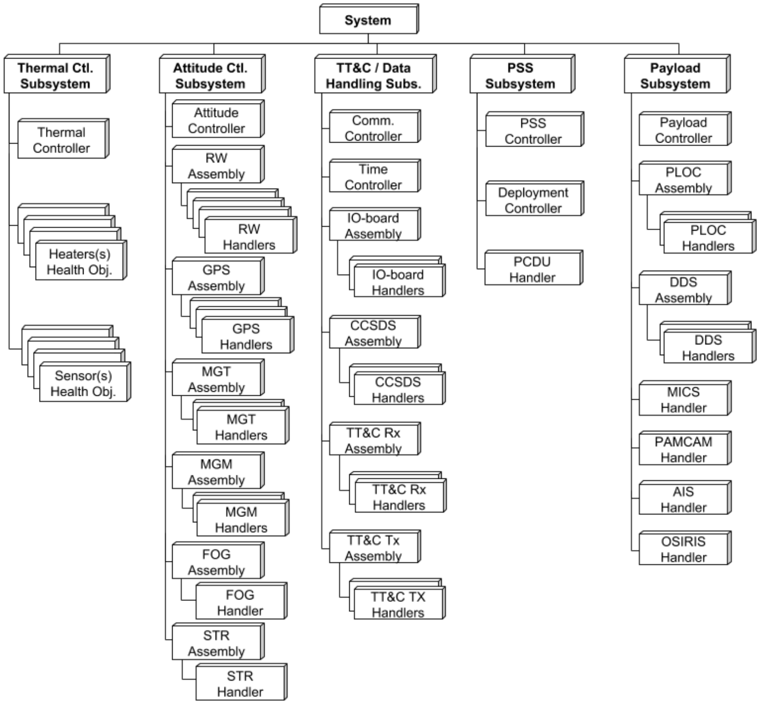


Figure 3.3.3: Components of the OBSW platform abstraction model representing an abstract model of the *Flying Laptop* system

If a system mode is commanded, the OBSW tries to command each individual OBSW component of the mode tree to the desired state. During their transition, the components issue events which provide insight into the progress of their individual mode transition. Once all OBSW components reach their desired mode, the system mode transition is complete. All actions required to command a component to another mode are implemented in such mode transition steps. E.g., if a device handler is commanded from *On* to *Off*, the device handler has to switch the power outlet of the onboard equipment as part of this transition.

If any of these intermediate steps fail, the OBSW undoes all already performed actions as part of the current mode transition and remains in the previous system mode. The operator receives a final "system mode transition failed" event. Through the individual events of all transitioning components, the operator can find out which transition caused the overall system mode transition to fail. This shows that a lot of the business logic that the OBSW performs is hidden within these mode transitions. For this reason, they have to be tested thoroughly which is described in section 5.2.

Low-level components have health states because they have to be able to reflect a faulty device or device group. A component's health determines how it can be used by its parent component. Component health can either be changed by command using the health management service (see Table 3.3.1) or by an automatic reaction to faults or limit violations performed by the components themselves. Higher-level components like subsystems or controllers cannot fail because they do not have a physical counterpart thus high-level OBSW components do not have an associated health state.

Since the satellite has a redundant architecture and often multiple redundant devices exist, groups of similar devices are naturally formed. To manage these groups of equipment, an abstraction layer on top of single device handlers was created with the introduction of assemblies.

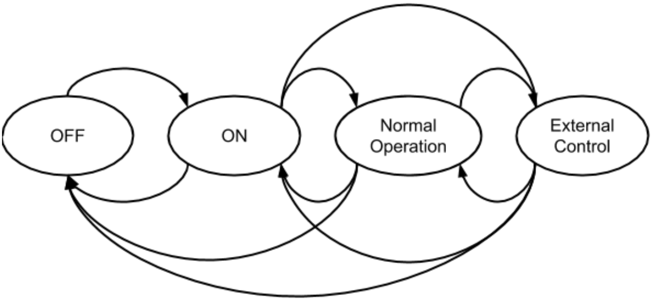


Figure 3.3.4: Possible mode transition of device handler component

Assemblies have built-in rules on how to manage their associated device handler OBSW components depending on their health state. If a device handler's health state becomes faulty, the device handler can no longer be used for nominal operation and is commanded *Off* by the assembly which in turn deactivates the associated equipment. The assembly will check if enough redundant devices are still healthy to keep the current mode. If not, the assembly component will itself transition to a faulty state and let the layer above decide on how to react. The tests of these reactions of the system to fault states are covered in section 6.2.

OBSW components can be regarded as finite-state machines[11] and their current state is represented by their mode. Each component has pre-defined modes and uses mode transitions to go from one mode to another. To provide an example, the standard mode transitions a device handler can perform are shown in Figure 3.3.4.

3.3.4 Testability of OBSW

Optimizing software design for testability includes consideration of test goals, the test methods which are applied and the surrounding conditions in which a test is performed. Important aspects to consider when judging software testability after [Jungmayr 2004] are:

1. if different modules can be tested in isolation
2. if tests are repeatable and
3. whether the software can be observed sufficiently to evaluate the test results.

These points were considered in the OBSW architecture. Especially, to facilitate ground tests with the satellite, the possibility to turn off or passivate subsystems is important. E.g., the OBSW has to be able to passivate the thermal control and attitude control subsystem during ground tests because in these situations the nonsensical sensor values would lead to erratic activation of heaters and actuators. Also, it is often necessary to prevent the system from executing automatic procedures, like e.g., the deployment of solar panels during ground tests.

The OBSW strategies to provide testability in detail are:

[11] A finite-state machine is an abstract machine that is in exactly one of a finite number of states at any given time. A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an external input is received [Sander et al. 1995].

- Taking advantage of PUS service 5 (event reporting service) to increase observability which is further described in section 4.7.
- A custom sub-service for PUS service 5 was added that offers the possibility to inject events into the running OBSW which is described in section 6.2.1.
- Wiretapping is a functionality of the device handler component. It can be used to downlink raw device messages as the device handler exchanges them with the device. This functionality is of great use if there is a problem with device communication especially during early device handler implementation and during device compatibility tests.
- The OBC provides a JTAG[12] interface and a SIF[13] which help to debug problems if the above-mentioned tools cannot provide enough insight.

3.4 The Flying Laptop System Testbed

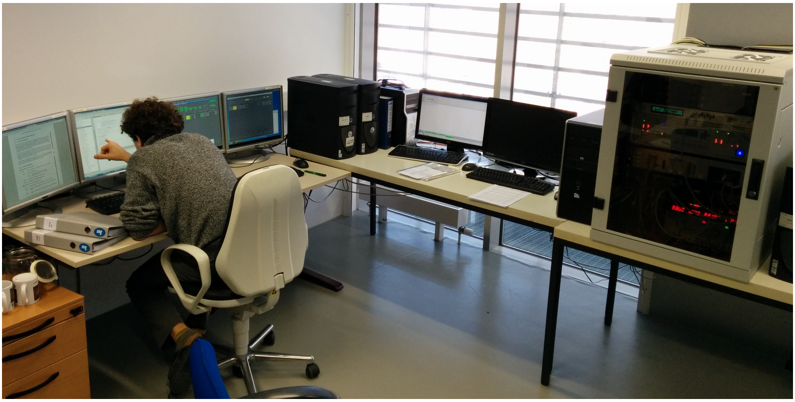
The functionalities listed in section 3.4 are the built-in features for testability of the OBSW and OBC. It is not possible to test control algorithms and failure handling by using these built-in testability features alone.

Typically, closed control loops are required to make such end-to-end testing possible. For this reason, model-based simulation methods have been introduced as a design and verification tool for the small satellite program and are described in detail in [Eickhoff et al. 2006], [Falke 2008] and [Fritz 2013].

A model-based real-time simulation environment based on the MDVE by Airbus Defense and Space (see section 2.3) was established as an OBC-in-the-loop system testbed (STB, Figure 3.4.1). At the heart of the STB lies an OBC engineering model, which is running the OBSW, connected to the real-time simulator representing the rest of the satellite system.

[12] JTAG interfaces are used for on-chip debugging and are widely used in the computer industry to test embedded software and hardware.

[13] Service Interface (SIF) is used to read debugging messages through a direct data line attached to the OBC during ground tests.



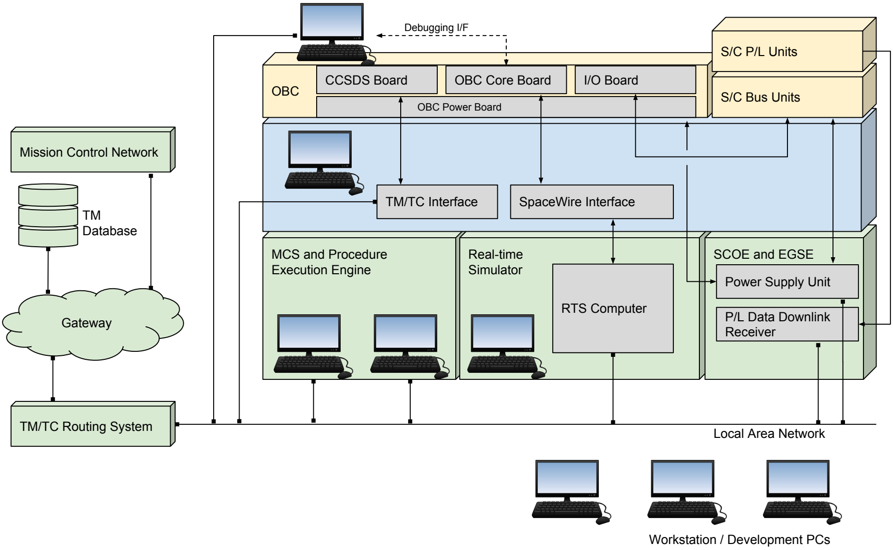


Figure 3.4.1: Final stage Flying Laptop STB as available in 2017; photograph of the working environment (top); schematic overview (bottom)

Because of the introduction of the new OBC design (see section 3.1), the STB could be simplified drastically by the use of the remote memory access protocol (RMAP) over one single SpaceWire line (cf. [Parkes et al. 2005]). The process of adapting the simulation environment to the new OBC design is described in [Fritz 2013].

The STB, as it was used for OBSW tests had the following capabilities:

- closed-loop verification of control algorithms within the OBSW like attitude control and thermal control.
- Simulate realistic orbit conditions, i.e. eclipse cycles, orbit propagation, magnetic field model, and air drag forces.
- Provide equipment simulation models for all onboard equipment of *Flying Laptop*. This means that the OBSW can interact with all units as if already in orbit.
- Possibility to trigger sequences of failures within equipment models, like e.g. defective units behavior to test failure handling.
- Debugging and inspection of the system at any given time and on a memory/instruction level of the OBSW and the simulation models.
- Uses the original mission control system and the actual command and parameter database which will also be used for mission operations.
- Scripting/procedure execution for automatic test execution.

Figure 3.4.2 shows the internal electrical block diagram of *Flying Laptop*. Each of the onboard equipment in Figure 3.4.2 is represented by a simulation equipment model except the OBC system.

In addition to the state of the STB presented in [Fritz 2013], additional elements were introduced to the STB as part of this work. As shown in Figure 3.4.1, these additional elements include a more powerful telemetry and command routing system which brings more protocol flexibility and also allows the seamless connection of the STB to the mission control network for operator training.

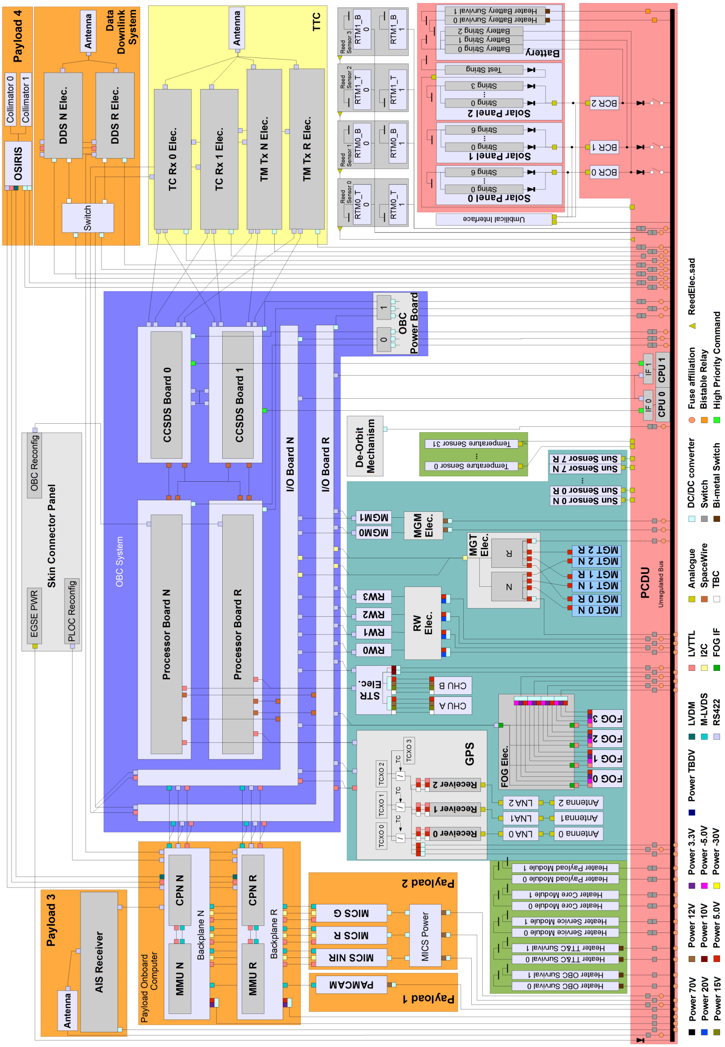


Figure 3.4.2: Electrical block diagram of Flying Laptop; all equipment, data lines, power lines have software simulation models in the STB except the OBC system (purple)

Figure 3.4.1 contains a picture of the STB setup along with a schematic overview of its core elements. Elements marked green are regular workstation computers connected to the institute's network. Elements in blue represent interfaces between onboard equipment and the external infrastructure like real-time simulator, mission control system and power supply. The onboard equipment hardware is marked yellow; these are OBC and other units connected as hardware-in-the-loop.

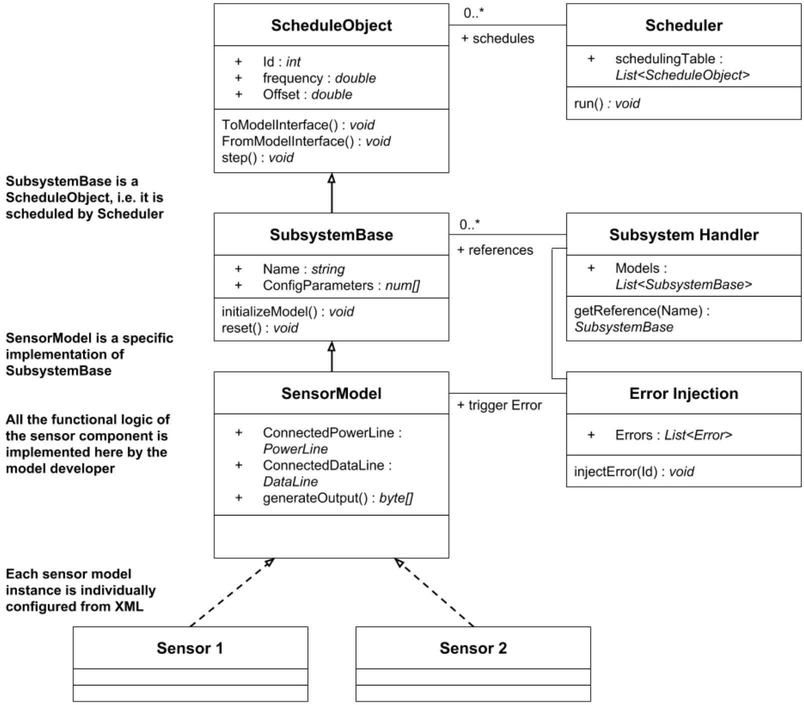
Ground communication in the real satellite system is done using TT&C radio equipment over S-band antennas. For this purpose, the CCSDS boards are connected to a radio receiver and transmitter which modulate and demodulate the CCSDS data stream. Unlike in the real satellite, the STB has no radio transceiver equipment. Instead, the CCSDS data stream is directly fed into a TM/TC frontend unit which processes the data stream and forwards it to the TM/TC routing system as shown in Figure 3.4.1.

3.4.1 Simulation Models

In model-based simulation technology, all logic that defines a system's behavior is implemented in simulation models. A spacecraft simulator typically consists of models for spacecraft dynamics, space environment, thermal conditions, and equipment models. Typically, numeric models like dynamics and environment models integrate all internal and external influences on the spacecraft and numerically solve the corresponding differential equations to determine the current state values of the simulation for each time step.

When modeling satellite equipment, a distinction between passive and active equipment is made. Influences of passive equipment like structure, mechanical parts, thermal insulation, or antennas can be neglected for functional modeling except for their characterizing parameters like mass, moment of inertia, or conductivity. Functionalities of active equipment, however, have to be modeled because active equipment influences the overall system behavior through its interaction with other equipment and the environment. Active equipment can either be modeled as transfer functions if their functional behavior is static or as state machines with temporal dependencies. For active equipment, to reflect the correct functional behavior of a satellite, the functionalities of the equipment, its interfaces and the interaction with the environment need to be taken into account [Eickhoff 2009].

Figure 3.4.3 shows the classes implemented for the *Flying Laptop* simulator. A sensor model, e.g., is an instance of the *SensorModel* class with a set of unique configuration values loaded from an XML data file. The *SensorModel* class inherits from the *SubsystemBase* class which provides methods to initialize and to reset the model. The Subsystem Handler manages references to all models. The Subsystem Handler can be used to obtain references to other models. *SubsystemBase*, in turn, inherits from *ScheduleObject*. A *ScheduleObject* provides methods that are called by the Scheduler after a configurable scheduling table.



SubsystemBase is a ScheduleObject, i.e. it is scheduled by Scheduler

SensorModel is a specific implementation of SubsystemBase

All the functional logic of the sensor component is implemented here by the model developer

Each sensor model instance is individually configured from XML

Figure 3.4.3: UML class diagram of a Flying Laptop sensor model showing dependencies and inheritance relations between the involved classes

The methods always called by the Scheduler are:

- **FromModelInterface**: Fetch all required external data from data lines, power lines, environment model and other equipment models.
- **step**: Process the internal state-logic and prepare model output data.
- **ToModelInterface**: Update all data lines, power lines, produced torque, change of velocity and consumed power.

The typical process of running the simulation is to first load the simulator which initializes all simulation models and then start the simulation by periodically running the scheduler.

The development of the equipment models of the *Flying Laptop* simulator was started by [Falke 2008] and [Fritz 2013]. For the purpose of this thesis, the models were improved and refactored in numerous student works to meet the following baseline criteria.

- The behavior of attitude control and power supply had to be simulated precisely as these subsystems are identified to be critical for mission success.
- The thermal state of the system was simulated rudimentary. The goal was that temperature sensor data does not hinder other testing activities, i.e. the overall simulated temperatures shall not be too hot or too cold.
- There was no further effort in the simulation of payload behavior other than the payload OBC's request/response protocol represented by valid payload dummy-data.

The decision to focus model development on these basic criteria was made to streamline model development and to be able to start OBSW testing as early as possible. A final list of the used simulation models for the *Flying Laptop OBSW* tests can be found in Appendix B.

Since *Flying Laptop* has a dedicated onboard computer for the payload subsystem (PLOC), the interaction with payload units from the viewpoint of the OBSW is always through the PLOC. This also means that the OBC has no direct connections to payload instruments. All payload simulation models have to provide characteristic parameters like power consumption and heat generation in different modes but do not produce any payload data for simulation purposes because this data is not handled by the OBSW. The PLOC model implements the full commanding protocol which is used to control payload activities from the ground, but all-device specific responses are only implemented as dummy data without any sophistication.

3.4.2 Error Injection

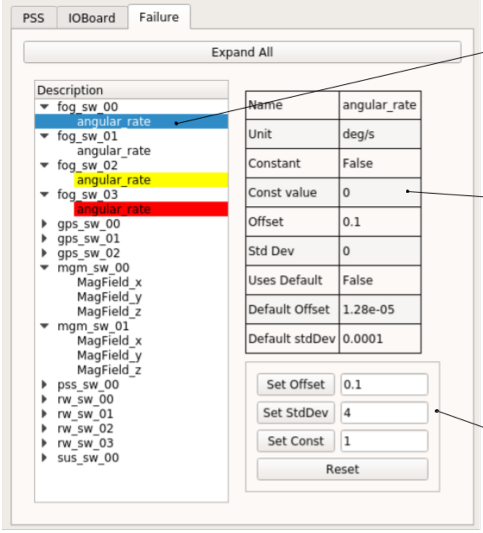
To simulate errors in onboard equipment, it is crucial to be able to modify the model's behavior and to inject errors at runtime. This way the OBSW's reaction to such non-nominal events can be tested. For this purpose, an error injection model has been developed for the *Flying Laptop* simulator which enables to

- activate and deactivate fuses and switches within the PCDU model
- activate and disable power and data harness lines
- manipulate sensor data

Sensor data can be modified in different ways. It is possible to override the current sensor value with a given constant value, increase or decrease the random noise of the value or to apply constant bias which is superimposed on the actual sensor value. This is realized through the use of a special class called *SensorBias*. Each sensor value is a member of the *SensorModel* class (see Figure 3.4.3). Upon initialization, each *SensorBias* object (each sensor value) registers itself at the central error injection model. The error injection model keeps track of all sensor values and provides an external interface that can be used to trigger any of the above sensor value modifications. An additional scheduler model activates the injected errors at a given point in time.

The tester can use a convenient tool to access the interface of the error injection model. Figure 3.4.4 shows the user interface to modify sensor values during simulation time. It lists all registered sensor values and also indicates if currently, any modifications are active. The three different tabs visible in Figure 3.4.4 are used to disable/enable power and data lines through the PCDU and I/O board models in a similar way.

In the left diagram of Figure 3.4.5, the effect of setting a *SensorBias* to a constant value is visualized. From the time on when the user clicks the 'Set Const' button in the graphical user interface, the chosen sensor value remains at the given level until the reset. The lower right diagram in Figure 3.4.5 visualizes increased noise on a *SensorBias* value. The amount of noise can be adjusted by setting the standard deviation for the value.



Description
- fog_sw_00
  - angular_rate
- fog_sw_01
  - angular_rate
- fog_sw_02
  - angular_rate
- fog_sw_03
  - angular_rate
- gps_sw_00
- gps_sw_01
- gps_sw_02
- mgm_sw_00
  - MagField_x
  - MagField_y
  - MagField_z
- mgm_sw_01
  - MagField_x
  - MagField_y
  - MagField_z
- pss_sw_00
- rw_sw_00
- rw_sw_01
- rw_sw_02
- rw_sw_03
- sus_sw_00

Name	angular_rate
Unit	deg/s
Constant	False
Const value	0
Offset	0.1
Std Dev	0
Uses Default	False
Default Offset	1.28e-05
Default stdDev	0.0001

Set Offset: 0.1
Set StdDev: 4
Set Const: 1

List of registered *SensorBias* values and their parent model instances in a tree view

Current state of selected SensorBias value including standard deviation of the applied noise, constant offset, unit type and the corresponding default values in a table view

User input to override *SensorBias* value properties

Figure 3.4.4: Graphical user interface of the error injection model; the graphical user interface communicates with the *ErrorInjection* model via ethernet connection; the current state of the model is transmitted to the graphical user interface application

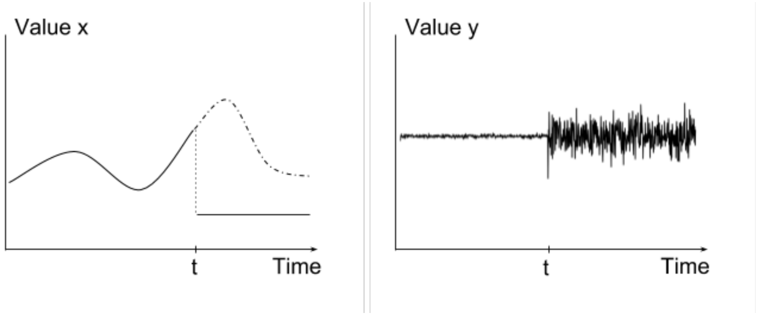


Figure 3.4.5: Two ways of influencing a simulation SensorBias value through failure injection; (left side) effect of setting SensorBias value to constant value; (right side) increasing sensor noise

4 Development Process and Testing of Flying Laptop OBSW

This chapter describes the applied methodology for developing the *Flying Laptop OBSW*. As already mentioned in section 3.1, because of special project conditions, an agile development approach was realized for the *Flying Laptop OBSW* where the design is driven by system tests rather than derived from requirements. Some formal requirements may be imperative, as in all other comparable approaches, but these formal requirements are deliberately not used to steer the testing/development process. Since the classic requirement based approaches as described in section 2.1.1 struggle to support dynamic design changes, a more suitable approach was elaborated to support the constant refinement of the design. The resulting approach is especially suited for small dynamic teams that want to quickly start working and see results rather than performing long planning phases.

In contrast to the classic system testing approach described shortly in section 4.1, section 4.2 gives a detailed description of the novel approach and also provides the reasoning for the selection of its building blocks based on project-specific circumstances. Section 4.5 described the toolchain and release cycles used for OBSW releases. Section 4.6 goes into detail about how regressions are handled.

4.1 Classic System Test Approach

In satellite development, system tests are performed with the assembled satellite to show that the built system is working as intended, to verify flight procedures and that the satellite is ready for launch. Three kinds of system tests are distinguished:

- System characterization tests to determine attributes of the satellite like mass, center of gravity, or moment of inertia
- Environmental tests like thermal vacuum tests or mechanical load tests which demonstrate the resilience and suitability of the system for space environment
- Functional tests to demonstrate the correct implementation of functionalities and flight procedures in an end-to-end manner

In traditional space projects, the flight software is already finished prior to the start of system tests. The reason for this is that the flight software is responsible for or at least involved in most tested functionalities and therefore also part of the system under test. Usually, significant effort is put into making sure that the risk of mission failure is minimized. Testing the already finished flight software during system testing is one aspect of this strategy. In traditional space projects modifying the flight software after system testing has begun would invalidate previous tests and would require repeating system tests over and over again [ECSS Committee 2018].

In classical industry projects, the verification approach is formulated in a verification plan document. This verification plan is created upfront by analyzing requirements like design peculiarities, qualification status of candidate solutions, availability of verification tools, available test methodologies, programmatic constraints, and cost considerations. The final verification plan then states *what* is to be verified, *how* it should be verified and *when* (meaning in which test environment) [ECSS Committee 2018].

Functional system testing in classical space projects structured in different test types [Eickhoff 2018]:

- **Integrated subsystem tests** (ISTs) test subsystem functionalities of each subsystem after integration.
- **System function tests** (SFTs) are large-scale tests using the manufacturer's own checkout equipment.
- **System validation tests** (SVT) are large-scale tests where the satellite remains at the manufacturer but is controlled using the ground segment of the customer.
- **Operational system timeline tests** (OST) include closed-loop simulation to operate the satellite at the manufacturer. The idea is to perform realistic operational scenarios usually over the duration of half a day or longer.

OSTs are sometimes part of **SFTs** and are always included in **SVTs**. Additionally, robustness tests are performed to test the system under load, e.g., to demonstrate data throughput, parallel activities, etc. [Eickhoff 2018].

4.2 Agile System Test Driven Development Approach

Since the development of the *Flying Laptop OBSW* was behind schedule (for reasons described in section 3.1), a traditional functional system test approach, where the OBSW development is completed and the source code is frozen before any system test activities begin, was impractical. Instead, the idea was born to merge OBSW development and functional system testing, starting with a bare OBSW version without any of the required functionalities and from that point on to always deliver the minimum viable product (MVP), as shown in Figure 4.2.1, in a Scrum like process (see section 2.1.2.2).

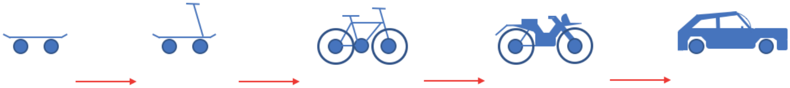


Figure 4.2.1: Principle of the minimum viable product (MVP) as basis for the OBSW development approach; licensed under Creative Commons © User:Teemu/Wikimedia Commons/CC-BY-SA-4.0

E.g., electromagnetic compatibility (EMC) tests could be performed months before essential parts of the OBSW like attitude control, thermal control or payload functionalities were available since the MVP for this test was an OBSW that only featured low-level device handling.

This iterative and incremental strategy served two purposes: First, to streamline the OBSW development process which could thus be organized in an agile and iterative way with fast iterations towards the next system test. And second, to enable the end-to-end functional testing of the system although the OBSW was not yet finished by adding the required features incrementally. Iterative means to continuously improve the OBSW in response to feedback from the testers. Incremental means to build the OBSW feature by feature instead of building functional blocks separately and then integrating them in a big bang integration effort at the end.

Not all system tests which are usually performed for a satellite are relevant for flight software development. System characterizations like RF compatibility tests and antenna pattern analysis are usually part of a system test campaign but do not contribute to the advancement of flight software development. Mechanical tests which are usually performed to demonstrate the capability of the satellite to withstand the mechanical and acoustic stress generated during launch are irrelevant for flight software development. The details of these system tests are not explained in great detail in this work but were of course also performed as part of the *Flying Laptop* system test campaign (see [Lengowski et al. 2015] for details).

In consideration of the fact that many system tests involve OBSW functionalities it is reasonable to approach functional system testing and OBSW development as a unified activity and to increase the interactions between both participating teams during phases C and D. By the application of the presented approach which combines aspects of TDD methods (see section 2.1.3), Scrum (see 2.1.2.2) and agile other concepts (see section 2.1.2) the *Flying Laptop* team was able to quickly respond to changes rather than following an inflexible development plan.

During system tests, the OBSW development team exercises a close collaboration with the system test engineers as illustrated in Figure 4.2.2. The testers constantly request new functionalities for the OBSW to be able to fulfill their testing goals and this way bring the OBSW closer to completeness. Therefore, the goals of the OBSW iterations are always to add the functionalities necessary for the next system test activity. In this relationship, the tester acts like an internal stakeholder, similar to an on-site customer (see section 2.1.2.1). Findings during test activities are directly reported to the OBSW developers. This way problems or bugs can efficiently be solved without interrupting the testing activities.

At the beginning of the development activities, the preparation phase, a development roadmap along with a product vision and an architecture vision is defined as depicted in Figure 4.2.2. This preparation phase is especially important for creating a common vocabulary among the team members. Creating visions about the product and the architecture as described in [Linz 2017] is essential for this approach because OBSW developers need to understand the terms used by system testers and vice versa. System testers need to be able to describe problems to developers by referring to a common nomenclature. Without this common nomenclature, conversations about the detailed OBSW architecture become chaotic and lead to problems because of misunderstandings. At the same time, the process of naming identifying and naming objects is already an important step in designing object-oriented software. By giving objects names, an underlying sense about the purpose of the software object and its relation to other objects is created, implicitly laying out the foundation of the software's architecture.

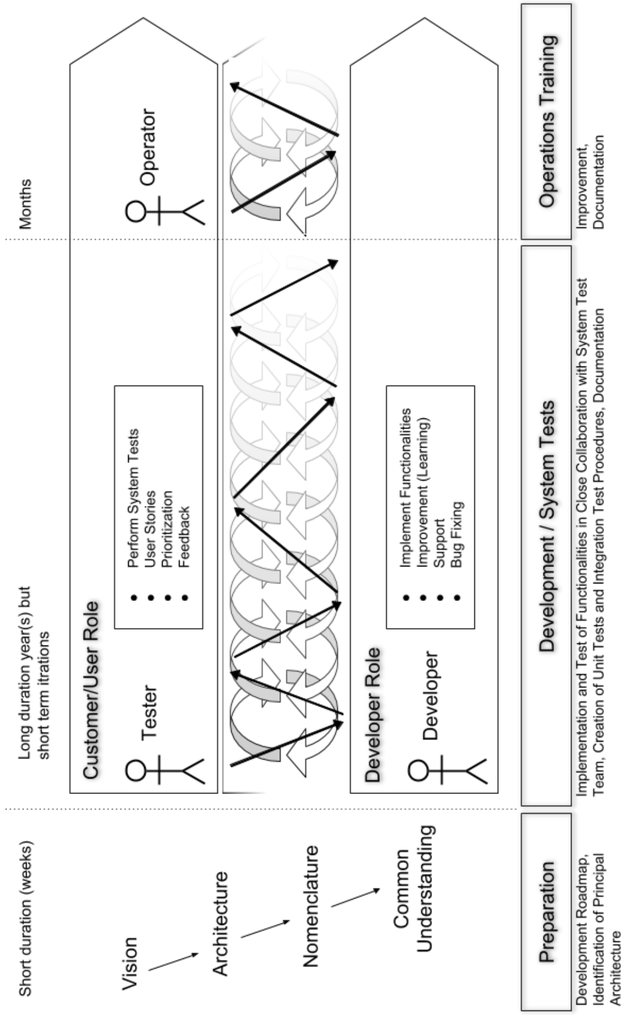

Short duration (weeks)
Vision
Architecture
Nomenclature
Common Understanding
Preparation
Development Roadmap, Identification of Principal Architecture
Long duration year(s) but short term itrations
Customer/User Role
Tester
Perform System Tests
User Stories
Prioritization
Feedback
Developer Role
Developer
Implement Functionalities
Improvement (Learning)
Support
Bug Fixing
Development / System Tests
Implementation and Test of Functionalities in Close Collaboration with System Test Team, Creation of Unit Tests and Integration Test Procedures, Documentation
Months
Operator
Operations Training
Improvement, Documentation

Figure 4.2.2: Interactions between tester and developer over the system testing campaign as applied in the system test driven OBSW development process

The development roadmap shown in Figure 4.2.3 serves as a rough means of orientation for the team about which functionalities to implement next and replaces a traditional development plan. For *Flying Laptop* the primary goal of functional system testing was to ensure the general robustness of the platform rather than excessively testing higher-level functionalities. The basic functionalities of the platform include:

1. power supply
2. communications
3. basic safe-mode attitude control

These three functionalities were targeted deliberately more than others because they are key to maintaining stable conditions in orbit in case something goes wrong.

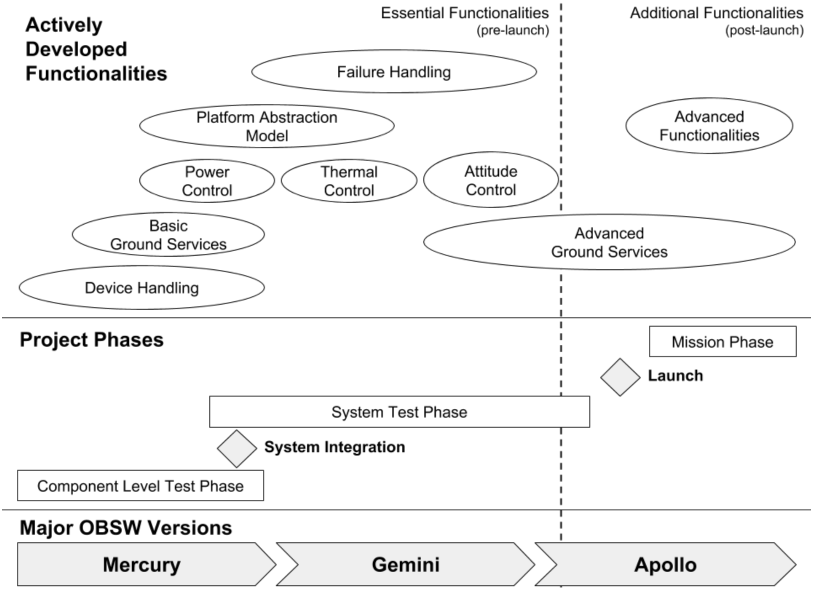


Figure 4.2.3: OBSW development roadmap; the roadmap was devised by the OBSW developers by distinguishing and prioritizing essential vs. optional functionalities as essential functionalities bring more value to the OBSW with the goal of achieving mission success

As Figure 4.2.3 shows, the three major OBSW versions, Mercury, Gemini and Apollo indicate these different phases in the project.

Mercury releases contain only device handling functionalities and some basic ground services to enable device-level compatibility tests. Gemini releases served as the OBSW which was used for incremental adding further required functionalities like thermal- and attitude-control, failure handling and the *platform abstraction model*. Apollo was the designated release for launch. It is the OBSW version which comprises all essential functionalities and can support the complete mission.

About a month before shipping, no further features were added to the OBSW. The current release at that point was used for mission preparations and last-minute pre-flight testing to ensure the actual flight release has matured sufficiently. Only critical bug-fixes were allowed to be added to the codebase during that period.

Figure 4.2.4 shows the different levels of test activities that were used during the system test campaign. Modified versions of Figure 4.2.4 will be used in chapter 5 to point out the corresponding levels for each section. Figure 4.2.4 also visualizes the incremental and iterative way of development within each level.

The approach diverges from a classic industry approach after the preliminary design phase and PDR. Since no CDR took place for *Flying Laptop* as a result of the application of the agile approach, the PDR was given increased significance. The PDR was only preliminary in the sense that the design decisions that followed were resulting from the agile approach. It was, however, final as it represented the last design review before production and testing began. Through constantly updating the test documentation and status reports using the online project management web platform Redmine[14], the current status of the test progress was visible at any point of the process.

With the approach of the launch date, an operational readiness review (ORR) of the ground segment and a flight acceptance review (FAR) of the space segment were conducted. In these reviews, the readiness of both ground and space segments was assessed based on the test results of the performed system tests.

[14] See www.redmine.org for information

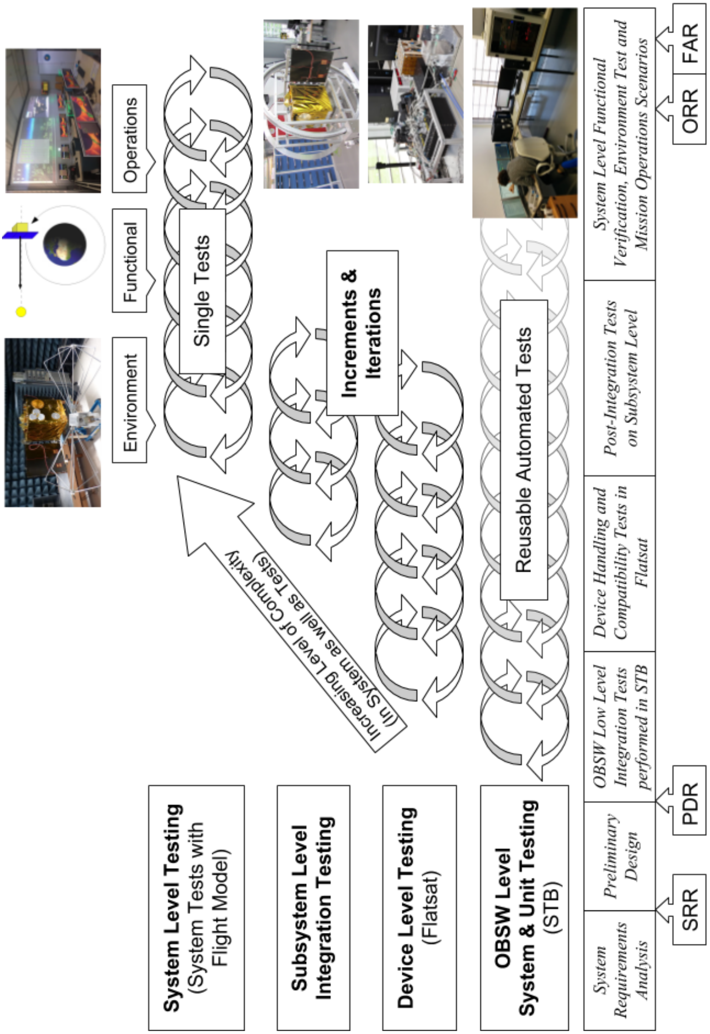
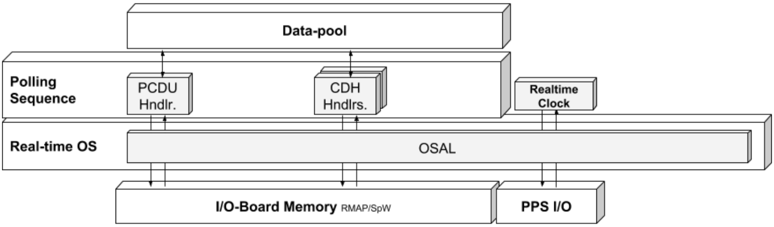

System Level Testing (System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing (Flatsat)
OBSW Level System & Unit Testing (STB)
Environment
Functional
Operations
Single Tests
Increments & Iterations
Reusable Automated Tests
Increasing Level of Complexity (In System as well as Tests)
System Requirements Analysis
Preliminary Design
OBSW Low Level Integration Tests performed in STB
Device Handling and Compatibility Tests in Flatsat
Post-Integration Tests on Subsystem Level
System Level Functional Verification, Environment Test and Mission Operations Scenarios
SRR
PDR
ORR
FAR

Figure 4.2.4: Different levels of test and development activities; lowest level is pure software testing while the highest level is representing test performed with the fully assembles satellite

4.3 Phases of OBSW Development

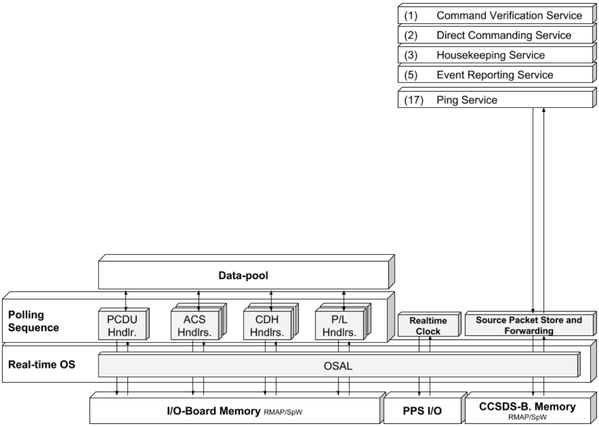
The eight steps visualized in this section should give the reader a better overview of the phases of OBSW development alongside system tests. It is important to point out, that over the whole system testing campaign up to the very end of the Apollo phase, an incomplete OBSW was used. Phases 1 through 8 are only rough guidelines and should help the reader identify what blocks of functionality were added to the OBSW in the various phases. However, the whole development of the OBSW between these phases was in constant iteration; even back-steps were sometimes required to deal with an unexpected problem or if it was recognized that a certain solution was not suited. The presented pictures for each phase are reduced versions of Figure 3.3.1. The reader can compare them with the full-size Figure 3.3.1 to get a complete picture.

Phase 1 - Basic Device Handling **OBSW Prototypes - M.1.0**



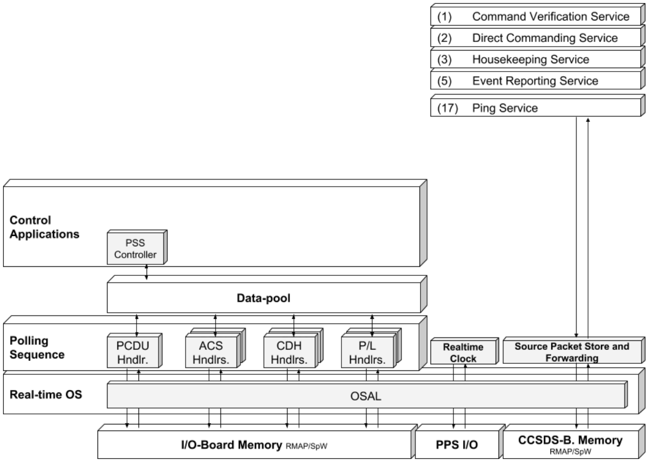
- PCDU device handler
- OBC boards handlers
- Data pool
- RMAP driver
- SIF
- Basic polling sequence

Phase 2 - Basic TM/TC **M.2.0 - G.12.0**



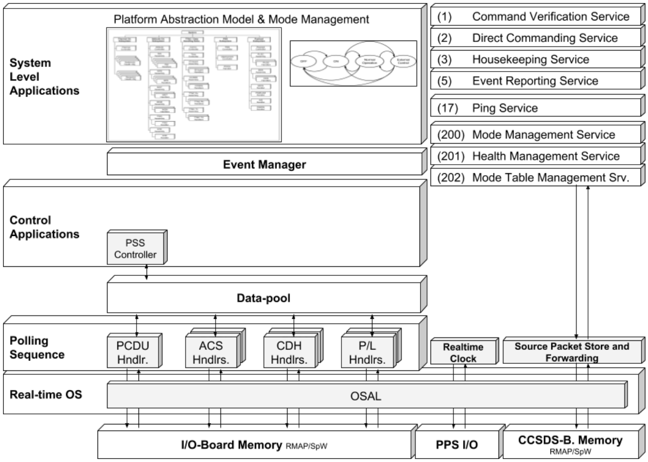
- Implementation of basic TM/TC services
- Basic PUS services for interaction with OBSW now through TM/TC link
- Commanding from mission control system using full CCSDS protocol
- Completion of device handlers

Phase 3 - PSS Controller G.13.0 - G.27.0



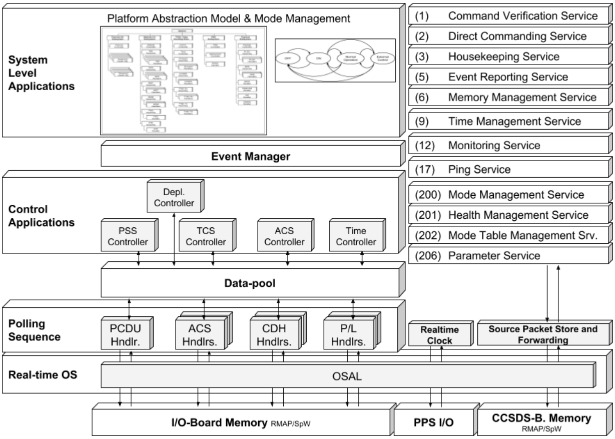
- Introduction of first control application
- Battery state of charge estimation

Phase 4 - Platform Abstraction G.28.0 - G.30


Platform Abstraction Model & Mode Management
System Level Applications
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
Event Manager
Control Applications
PSS Controller
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Realtime Clock
Source Packet Store and Forwarding
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW

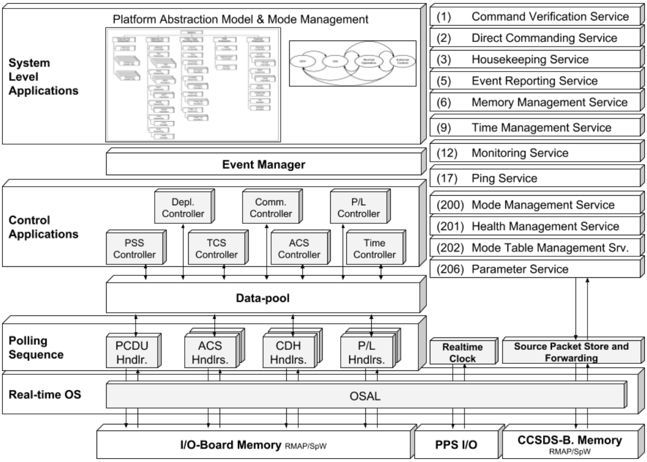
- Implementation of system-level functions
- Higher-level objects
- Event manager
- Platform abstraction model
- Mode tables and additional PUS services for mode & health management

Phase 5 - TCS & ACS G.31.0 - G.40.0


Platform Abstraction Model & Mode Management
System Level Applications
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(9) Time Management Service
(12) Monitoring Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service
Event Manager
Control Applications
Depl. Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Realtime Clock
Source Packet Store and Forwarding
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW

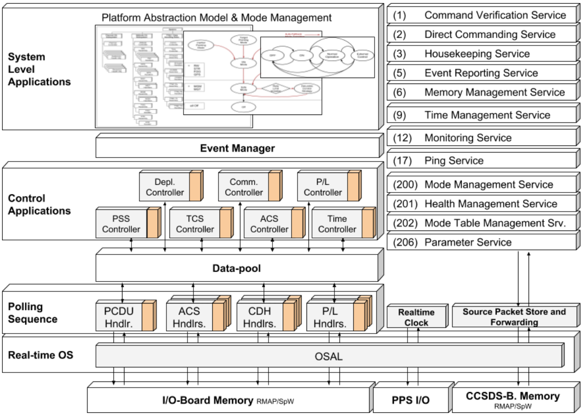
- TCS & ACS controller controller
- Time manager for precise onboard time
- Monitoring service
- Deployment controller (section 5.1.3.4)
- Memory management and parameter service to adjust onboard parameters

Phase 6 - Communication & P/L Control G.41.0 - G.43.0


Platform Abstraction Model & Mode Management
System Level Applications
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(9) Time Management Service
(12) Monitoring Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service
Event Manager
Control Applications
Depl. Controller
Comm. Controller
P/L Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Realtime Clock
Source Packet Store and Forwarding
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW

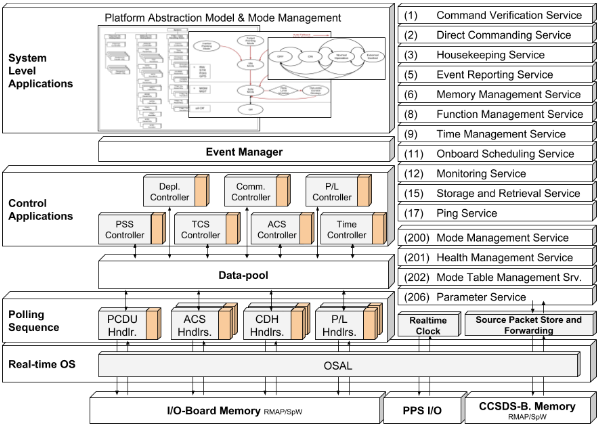
- Completion of control applications with automatic activation of downlink
- Payload controller for better control over Payload instruments

Phase 7 - Gemini Release Candidates G.43.0 - G.53.0-RC (final Gemini)


Platform Abstraction Model & Mode Management
System Level Applications
Event Manager
Control Applications
Depl. Controller
Comm. Controller
P/L Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Realtime Clock
Source Packet Store and Forwarding
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(9) Time Management Service
(12) Monitoring Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service

- Candidates for Apollo increment
- Failure detection
- Low-level and high level-failure handling
- Assemblies perform redundancy mgmt.
- System-level application FDIR triggers fallbacks based on events

Phase 8 - Advanced TM/TC (Apollo) A.1.0 - A.11.0


Platform Abstraction Model & Mode Management
System Level Applications
Event Manager
Control Applications
Depl. Controller
Comm. Controller
P/L Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Realtime Clock
Source Packet Store and Forwarding
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(8) Function Management Service
(9) Time Management Service
(11) Onboard Scheduling Service
(12) Monitoring Service
(15) Storage and Retrieval Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service

- Advanced TM/TC services
- Functions encapsulate complex onboard procedures (service 8)
- Onboard scheduling enables time tagged commanding
- Finalization of onboard TM stores

4.4 The OBSW Development Approach Summarized

In summary, the applied approach is based on a few fundamental principles: OBSW development was structured as an incremental and iterative approach that produces a minimum viable product for each system test. The features to be implemented next were dictated by the upcoming system test. Tests were prioritized according to a distinction between essential and optional functionalities where essential functionalities were implemented first because they brought more value; in the sense that they represent the indispensable functions of the system, i.e. power supply, communications, and safe-mode attitude control.

In some areas where ground-based system tests with the flight model were difficult to realize, tests were performed on the digital twin; the STB which is described in section 3.4. Cross checks were performed with the flight model if ever possible to increase the plausibility of test results gathered from STB tests.

In phases 1, 2 and 3, described in section 4.3, where low-level device handling with Mercury OBSW releases was tested, a flatsat was built using flight equipment prior to system integration. The flatsat is further described in section 5.1.

From phase 4 to 8, system tests were performed with the wholly assembled flight model using OBSW version Gemini.

Once the final Apollo OBSW was released after phase 8, operational scenarios and flight procedures were rehearsed both using the flight model and STB. These final tests served two purposes, flight acceptance and operator training.

In addition to the tests performed for the system test campaign, a continuous testing process that focused more on finding bugs, regressions, and general problems in new releases was carried out. This continuous process is a combination of automated test procedures with the goal of maximizing code coverage and an exploratory testing effort that was meant to identify problems in usability. Test automation and exploratory testing for *Flying Laptop* is described in chapter 6.

4.5 Release Life Cycle, Toolchain and Versioning

Figure 4.5.1 visualizes the proposed development and release process for *Flying Laptop OBSW* including release testing. At the beginning of each release cycle, the development team planned the upcoming release by taking into account the development roadmap and feature requests from the testing and operations team. As described in section 4.1 the system test campaign was the main driver for the *Flying Laptop OBSW* development.

The release schedule is a short-term summary of all development activities required for the upcoming release. Depending on the selected features the development team began implementing and bug-fixing while continuously performing developer tests with the current codebase. Tests which each developer performed in self-responsibility were mandatory, i.e. tests of new features or bug-fixes, unit tests and integration tests performed on the STB.

4.5.1 Release Life Cycle

During the entire OBSW lifecycle configuration parameters, mission parameters, command definitions and event definitions were subject to change. To produce comparable test results and also to find solutions to problems based on historical data, it was essential to track these changes throughout the complete lifecycle. This was done by versioning OBSW releases but also the mission database containing onboard parameter, command and event definitions. This ensured that for each OBSW release a compatible set of mission parameters was available and test results could be reproduced at any time.

Fortunately, the days in which tracking changes to source code was complicated are in the past. Version control systems like Git[15] are quickly set up and can be used in a flexible way, either using a central repository server or on a local machine. Also moving repositories from local machines to central servers or merging of development branches is handled efficiently by any modern version control system.

4.5.2 Toolchain

For the OBSW development, the bug tracker software Bugzilla[16] was used. After a bug was found, the tester or developer who identified the bug created a new issue in Bugzilla. If the

[15] Git is recommended as a code repository as it provides all the required capabilities, is widely used among software development teams and open source. However, any modern version management tool will be equally sufficient.

[16] Bugzilla website: www.bugzilla.org

bug was critical, it was prioritized and if it was so critical that it needed an immediate solution, a hotfix release was scheduled which is a revision of the most recent version containing only this necessary bug-fix.

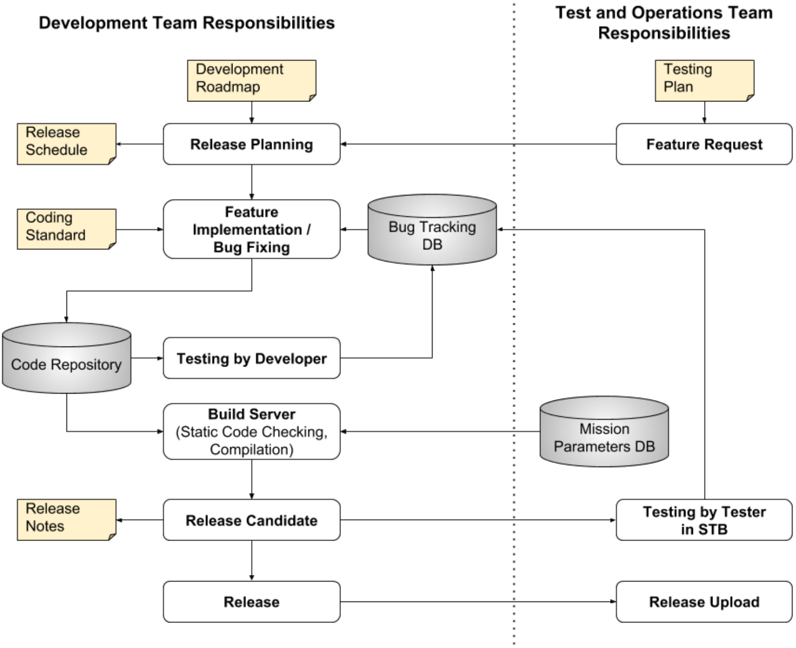


Figure 4.5.1: Flying Laptop onboard software release process and build toolchain

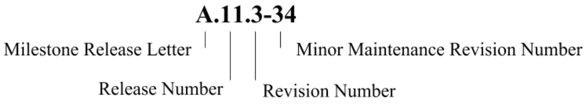
For each feature and bug-fix, the assigned developer created a new branch in the code repository and started working on its implementation. After the implementation was complete and tested by the developer, the feature or bug-fix branch was merged into the OBSW master branch. After tagging a specific commit with a release version-number, the build-server compiled the master branch and automatically generated the parameters initialization code from the mission database.

The output of the compilation step is an OBSW image which is ready to be uploaded to and executed on the OBC.

For the *Flying Laptop OBSW* cppcheck[17] was used as static code checker. Cppcheck is executed by the build-server in an automated step in the integration process *build server* of Figure 4.5.1. Cppcheck automatically generates reports about findings and also tracks which problems were solved or added compared to the release before.

4.5.3 Versioning

The versioning scheme that is used for OBSW development is explained here because later on in this work, there will be references to OBSW version numbers especially in chapter 5 where different system test activities are linked to specific OBSW versions. This is done to give the reader an overview and that it is possible to follow the context of OBSW development alongside system testing.



As already mentioned, the OBSW releases are grouped into three milestone release categories: Mercury (M), Gemini (G) and Apollo (A). In the OBSW versioning scheme, this is indicated by the first letter of the version number.

The milestone release letter is followed by a release number which is incremented with each release. A release typically introduces new functionalities and is officially scheduled using a release plan.

The third position in the OBSW version number indicates the revision number. Revisions contain bug fixes and are released as required without following a release plan. Sometimes, if a bug fix is so minor that no actual revision is made another incremental number is added.

[17] Cppcheck website: cppcheck.sourceforge.net

4.6 Handling of Regressions

As described in section 2.3.3, the practice of determining if a change to the codebase causes problems in other parts of a program as a result of interaction between different software parts is called regression testing. If undetected, regressions can cause significant problems in software over its lifecycle. Testing for regressions is crucial to prevent the overall software quality from declining from release to release.

The strategy to address those issues for the *Flying Laptop OBSW* was to implement automated test procedures for all major functionalities (see section 6.5). In contrast to unit tests, these regression tests are performed on system-level and by using a black-box approach. The successful execution of these automated test procedures certified the integrity of an OBSW release. This implied that the release had to provide the same functionality as any previous release which had undergone the same regression tests and that none of the previously implemented functionalities were broken. If a new functionality was introduced, a new test module was implemented along with it to demonstrate the functionality in future executions.

To provide retrospective for test results, each data set produced during regression test runs was archived for the specific release. This way it was possible to reconstruct exactly when a change introduced a problem that might only have become apparent later on.

4.7 Event Reporting Service for Satellite Introspection

In order to evaluate OBSW tests efficiently, a tester needs to know what is happening onboard the satellite. One means of introspection is the ECSS PUS event reporting service (as part of the TM/TC ground services mentioned in section 3.3.2). In contrast to periodically updated reports of onboard parameters received through the housekeeping service, events are unique instances generated sporadically by OBSW when an error is detected, autonomous onboard actions are performed or progress is made in normal onboard activity. When an OBSW object generates an event, a TM source packet containing a corresponding event report is generated. The source packet is then either stored or directly downlinked to the ground through the telemetry link.

Table 4.7.1: Characteristic parameters contained in an OBSW event

Parameter	Description	Required by ECSS PUS
Timestamp	The generation timestamp of the source packet is also the event timestamp	yes
Event Id	The identifier if the event, sometimes also called report Id	yes
Component Object Id	The OBSW object address which generated the event	no
Severity Level	Signifies the severity of the occurred event - Four levels are recommended by [ECSS Committee 2003]: **Info, Low, Medium, High**	yes
Event Parameters	Additional set of auxiliary parameters containing information about the cause or context of the event	optional

Source packets containing an OBSW event report have the characteristic parameters listed in Table 4.7.1. According to the packet utilization standard [ECSS Committee 2003], onboard events are classified in different severity levels identified by the service subtype: Info event, low severity event, medium severity event, high severity event.

A specific combination of events over time is like a signature for a specific activity. E.g., events generated for system mode from *Safe* mode to *Idle* mode are unique for that particular activity and can be used to track changes to the OBSW by comparing their timely sequence between tests.

Because of their infrequent one-time occurrence, onboard event reports require a different kind of visualization compared to regular parameter based telemetry data. It turned out that attributing events to their OBSW components which generated them was beneficial to clearly trace what was happening during test activities. This way, also component interactions can be visualized effectively. Figure 4.7.1 shows an example of recorded events generated by the involved components after an unresponsive GPS unit (GPS unit 1) triggers a recovery.

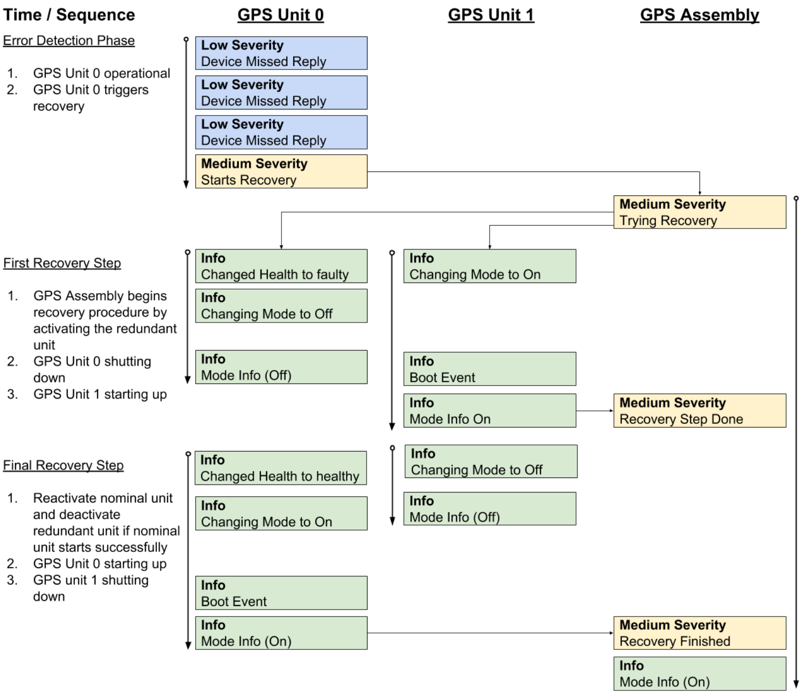
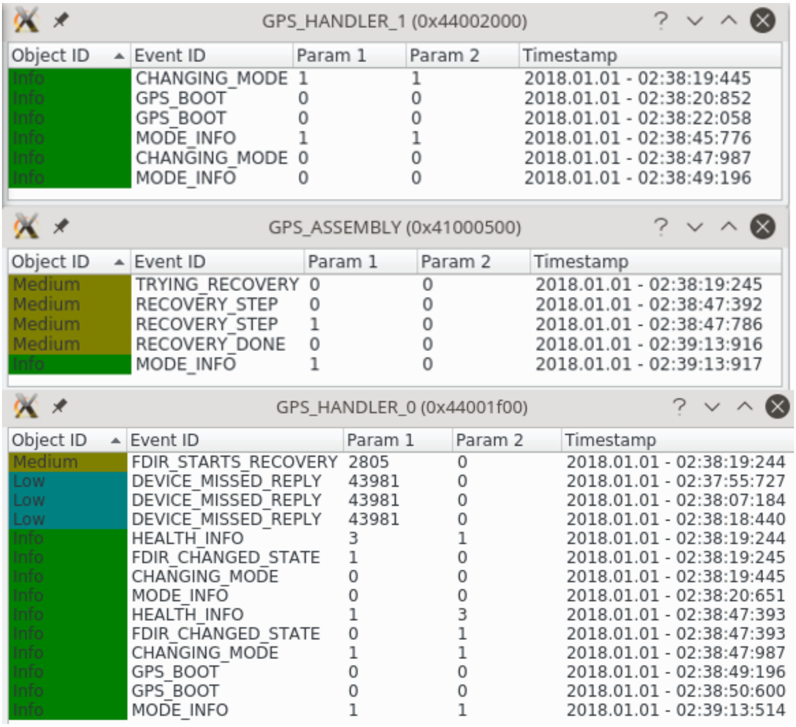

Time / Sequence
GPS Unit 0
GPS Unit 1
GPS Assembly
Error Detection Phase
1. GPS Unit 0 operational
2. GPS Unit 0 triggers recovery
Low Severity Device Missed Reply
Low Severity Device Missed Reply
Low Severity Device Missed Reply
Medium Severity Starts Recovery
Medium Severity Trying Recovery
First Recovery Step
1. GPS Assembly begins recovery procedure by activating the redundant unit
2. GPS Unit 0 shutting down
3. GPS Unit 1 starting up
Info Changed Health to faulty
Info Changing Mode to Off
Info Mode Info (Off)
Info Changing Mode to On
Info Boot Event
Info Mode Info On
Medium Severity Recovery Step Done
Final Recovery Step
1. Reactivate nominal unit and deactivate redundant unit if nominal unit starts successfully
2. GPS Unit 0 starting up
3. GPS unit 1 shutting down
Info Changed Health to healthy
Info Changing Mode to On
Info Boot Event
Info Mode Info (On)
Info Changing Mode to Off
Info Mode Info (Off)
Medium Severity Recovery Finished
Info Mode Info (On)

Figure 4.7.1: Example recovery of GPS unit; events produced during automatic onboard recovery procedure; color code: **green**=info event, **blue**=low severity event, **yellow**=medium severity event

When viewing Figure 4.7.1 and Figure 4.7.2 it is clear that a component-based view of the sequence of events is superior to a simple list view of the received events because it is possible to follow the event stream of a component individually. The timely interaction of OBSW components gets much clearer once the components can be viewed side by side. To allow for such a component-based display, the event report has to contain also the object identifier.

This way of visualizing event reports and attributing them to their OBSW components has been realized in a software tool TMInspector[18] which significantly improves the satellite's observability during testing activities. Figure 4.7.2 is depicting how the same example sequence of events introduced in Figure 4.7.1 can be visualized using the TMInspector software.



GPS_HANDLER_1 (0x44002000)

Object ID	Event ID	Param 1	Param 2	Timestamp
Info	CHANGING_MODE	1	1	2018.01.01 - 02:38:19:445
Info	GPS_BOOT	0	0	2018.01.01 - 02:38:20:852
Info	GPS_BOOT	0	0	2018.01.01 - 02:38:22:058
Info	MODE_INFO	1	1	2018.01.01 - 02:38:45:776
Info	CHANGING_MODE	0	0	2018.01.01 - 02:38:47:987
Info	MODE_INFO	0	0	2018.01.01 - 02:38:49:196

GPS_ASSEMBLY (0x41000500)

Object ID	Event ID	Param 1	Param 2	Timestamp
Medium	TRYING_RECOVERY	0	0	2018.01.01 - 02:38:19:245
Medium	RECOVERY_STEP	0	0	2018.01.01 - 02:38:47:392
Medium	RECOVERY_STEP	1	0	2018.01.01 - 02:38:47:786
Medium	RECOVERY_DONE	0	0	2018.01.01 - 02:39:13:916
Info	MODE_INFO	1	0	2018.01.01 - 02:39:13:917

GPS_HANDLER_0 (0x44001f00)

Object ID	Event ID	Param 1	Param 2	Timestamp
Medium	FDIR_STARTS_RECOVERY	2805	0	2018.01.01 - 02:38:19:244
Low	DEVICE_MISSED_REPLY	43981	0	2018.01.01 - 02:37:55:727
Low	DEVICE_MISSED_REPLY	43981	0	2018.01.01 - 02:38:07:184
Low	DEVICE_MISSED_REPLY	43981	0	2018.01.01 - 02:38:18:440
Info	HEALTH_INFO	3	1	2018.01.01 - 02:38:19:244
Info	FDIR_CHANGED_STATE	1	0	2018.01.01 - 02:38:19:245
Info	CHANGING_MODE	0	0	2018.01.01 - 02:38:19:445
Info	MODE_INFO	0	0	2018.01.01 - 02:38:20:651
Info	HEALTH_INFO	1	3	2018.01.01 - 02:38:47:393
Info	FDIR_CHANGED_STATE	0	1	2018.01.01 - 02:38:47:393
Info	CHANGING_MODE	1	1	2018.01.01 - 02:38:47:987
Info	GPS_BOOT	0	0	2018.01.01 - 02:38:49:196
Info	GPS_BOOT	0	0	2018.01.01 - 02:38:50:600
Info	MODE_INFO	1	1	2018.01.01 - 02:39:13:514

Figure 4.7.2: Example recovery of GPS unit as displayed to the operator in TMInspector software

[18] TMInspector on GitHub: github.com/nicobucher/TMInspector

A second concept introduced during system testing has later been adopted for the *Flying Laptop* ground segment: The data storage concept by which historical telemetry data is archived. It facilitates lookups for archived event reports but also telemetry parameters and space packets. A dedicated software written in Java called *tmarchive* stores received telemetry packets in a MySQL database [Klemich et al. 2016]. Also, *tmarchive* extracts contained parameters and events from the corresponding reports source packets and stores them in separate tables for later retrieval. The database supports large quantities of parameters and events to be stored.

For the retrieval of historical telemetry data through *tmarchive*, a REST API[19] was developed which allows efficient lookup of source packets, parameters, and events for a specific timeframe. This API can be flexibly used by other applications such as web pages or plotting tools to provide historical telemetry data to interest groups but also by the TMInspector software to reproduce test data and comprehend historical sequences of events.

[19] Representational State Transfer Application Programming Interface

5 Flying Laptop System Test Campaign

In this chapter, the application of the software development approach which was outlined in section 4.2 is described. The approach is exemplified for the development of the *Flying Laptop OBSW* which took place at the University of Stuttgart between the years 2013 and 2017. The OBSW development was guided by the system test campaign for the satellite.
Figure 5.0.1 shows the chronological sequence of system tests that were performed for *Flying Laptop*. The following sections describe each incremental step that was performed until the final OBSW was finished.
At the end of each section of this chapter, the benefits of the presented approach will be summarized in comparison to a fictional classical approach which is relying on the traditional methodologies of flight software development and functional satellite system testing as described in sections 2.1.1, 2.2, and 4.1.

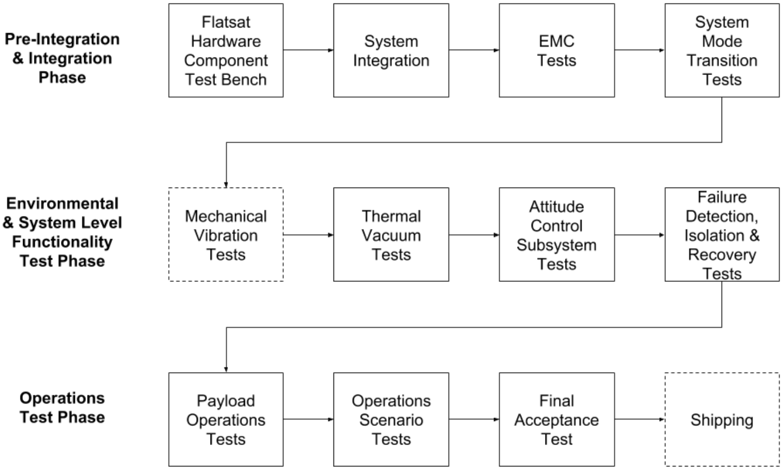

Pre-Integration & Integration Phase
Flatsat Hardware Component Test Bench
System Integration
EMC Tests
System Mode Transition Tests
Environmental & System Level Functionality Test Phase
Mechanical Vibration Tests
Thermal Vacuum Tests
Attitude Control Subsystem Tests
Failure Detection, Isolation & Recovery Tests
Operations Test Phase
Payload Operations Tests
Operations Scenario Tests
Final Acceptance Test
Shipping

Figure 5.0.1: Chronological sequence of Flying Laptop system tests; full line: Tests involving OBSW development; dashed line: Activities without participation of OBSW

5.1 From Flatsat to Fully Integrated Satellite

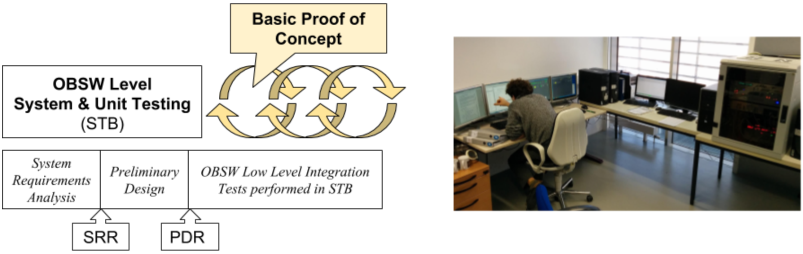

Basic Proof of Concept
OBSW Level System & Unit Testing (STB)
System Requirements Analysis
Preliminary Design
OBSW Low Level Integration Tests performed in STB
SRR
PDR

Figure 5.1.1: Context of first proof of concept work with the STB and OBC engineering model prior to flatsat and system integration

First device level compatibility tests were performed with a proof of concept version of the OBSW. This OBSW version used simple periodic tasks and basic I/O which was offered by the RTEMS operating system. This served the purpose of trying out the basic functionalities on which later, more complex OBSW components were built. Using the STB, engineering models of onboard equipment were connected to the I/O board and first communication tests were performed. This kind of device-level testing represented the first level of complexity for functional tests as shown in Figure 5.1.4.

A flatsat is a particular form of satellite test bench where the individual onboard equipment units are laid out flat on a table. Sometimes this test bench is also called *electrical functional model* (EFM) [Eickhoff 2012]. Because of their special arrangement, each piece of equipment can be accessed conveniently by test personnel to better work with the equipment and harness. This way, a more dynamic iterative way of working with the equipment is made possible without the risk of having to disassemble the whole satellite if, e.g. a connector needs to be detached.

Since the OBSW team had little experience in this field, the trial and error methodology presented an excellent way to familiarize the developers with the intricacies of the system.

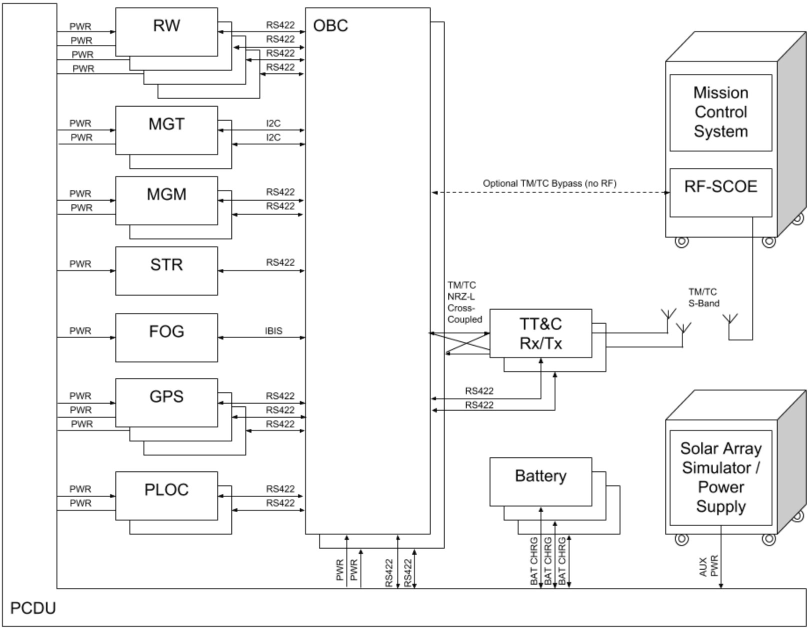


Figure 5.1.2: Schematic overview of the flatsat test setup including OBC, PCDU, TT&C unit and battery.; PWR = Equipment power line; BAT CHRG = Battery charging line; RS422, LVDS, I2C and UART describe the underlying digital signaling scheme

For *Flying Laptop* an extensive flatsat phase was performed before system integration. The device handling of the OBSW was refined along with the performed tests. All onboard equipment was mounted on aluminum plates and connected by a specially repurposed harness which was previously used for mockup purposes. This way, the preparation of the main structure and the flight harness could already be started while the flatsat tests were still ongoing. Figure 5.1.2 shows all flatsat equipment and their interconnection which were realized by the flatsat harness (compare also Figure 5.1.3). Although the test harness had good similarity to the flight harness (identical types of connectors, wires and shielding) minor modifications like pin allocation of connectors were still required to realize all flatsat tests.

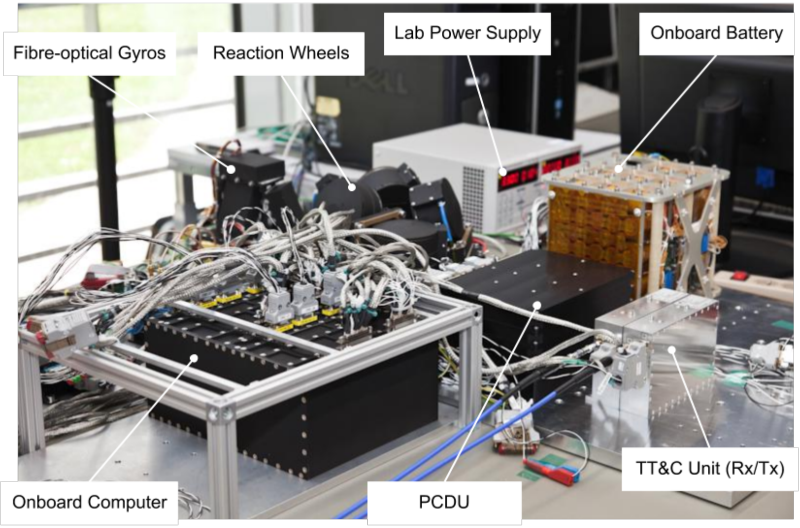


Figure 5.1.3: Impression of the Flying Laptop flatsat; actual flight hardware in on-desk arrangement using a dedicated test harness

The main purpose of a flatsat test phase within a satellite project is to demonstrate the compatibility between onboard equipment on both electrical and data levels before their final integration into the satellite structure. To perform data-level compatibility tests, the OBSW must provide the corresponding device handler components for each device. The *Flying Laptop* flatsat test phase was used to incrementally implement each device handler as new devices were added to the flatsat. This method is drastically different from the classic method applied for space projects where the complete software must be ready before. The method offered a unique opportunity for the OBSW developers to directly interact with the device as they developed each device handler.

5.1.1 System Startup Procedure and Basic Communication

Before any functional flatsat tests could begin, the OBC needed to be brought into operation. To achieve this, a number of internal OBC functionalities needed to be tested. These functionalities included loading the OBSW binary file, communicating with all peripheral boards over SpaceWire (see section 3.2.1 for details on OBC interfaces) and logging debugging output over the SIF. All of those functionalities were tested in the early stages of the flatsat campaign. As the first equipment, the OBC box was integrated into the flatsat and connected to the test harness. At this point, a lab power supply was used to provide power to each of the core and peripheral boards individually as shown in Figure 5.1.3.

After powering the core CPU boards, a bootloader software was installed using the JTAG interface. After the next power cycle, the OBSW starts automatically from the bootloader. In its startup procedure, the OBSW checks the availability of the peripheral OBC boards and initializes tasks for executing the polling sequence, basic ground communications, and the SIF.

The next steps were to access all available OBC memory regions both internal and external using RMAP. At first, this was done using the JTAG interface. Later, the ECSS PUS memory management service (service 6, see section 3.3.2) which enables memory access through the TM/TC interface replaced memory access over JTAG whenever direct memory read and write had to be performed. The memory management service also enabled OBSW updates through the command link. Figure 5.1.5 shows the basic TM/TC ground services that were available in this phase.

5.1.2 Power Supply and Distribution

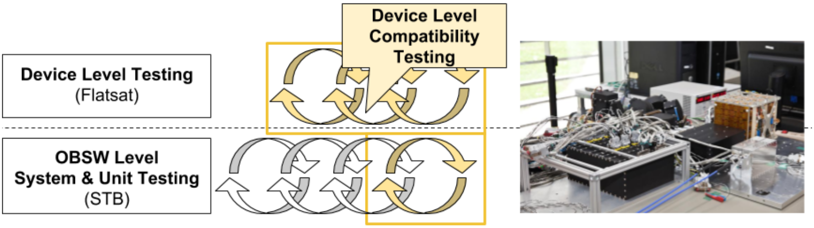


Figure 5.1.4: Context of device level in OBSW development process; compatibility tests in flatsat and development of device handling

The power supply subsystem (PSS) consists of PCDU, solar panels, and battery. Power is generated from the solar panels when being illuminated by sunlight proportional to the elevation angle of the sun. The generated power is then distributed to the active satellite equipment over an unregulated main power bus. Unconsumed energy is used to charge the batteries. If no power is generated from the solar panels, the battery cells provide the required power.

5.1.2.1 Power Supply Subsystem Tests

The first functional system tests after taking the OBC into operation were related to the PSS. These tests were essential for every following test activity because they ensured the power supply of all onboard equipment so that they can be operated safely.

Instead of the actual flight battery, an engineering model made from spare battery cells was used to prevent deterioration of the original battery cells used for flight. This engineering model remained the energy storage for almost all system test activities until it was exchanged with the flight model shortly before launch. Likewise, throughout all further test activities, a solar array simulator unit was used to recharge the battery cells since the real solar cells can not generate enough power from artificial light and thus could not be used in ground tests (see Figure 5.1.2). For this purpose, the PCDU offers auxiliary solar panel input lines to provide solar panel current from an external power source.

To pass the PSS test, the system had to show the following capabilities:

- PCDU device handling
- battery charging/discharging and battery string balancing
- solar panel voltage control
- equipment safety features like over-current protection

In the OBSW design iteration which took place before the PSS test, the OBSW developers had to implement additionally required components. These OBSW components included the PCDU handler, the housekeeping service, the commanding services, and the event reporting service. The OBSW components available for the PSS test are shown in Figure 5.1.5.

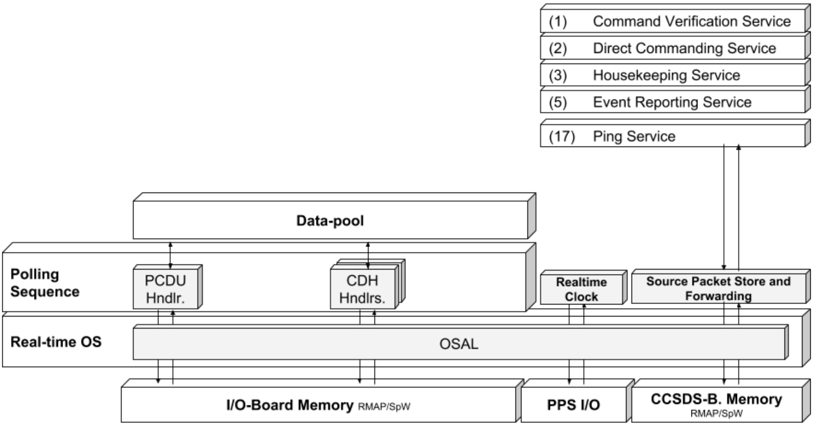


Figure 5.1.5: OBSW components available at the beginning of PSS test

To perform PSS tests, the minimal arrangement of units that constitute the PSS had to be assembled. This was done in three steps:

1. The solar array simulator was connected to the auxiliary PCDU solar panel inputs
2. The OBC was connected to the corresponding PCDU connector
3. The engineering model of the battery was connected to the PCDU providing power over a switchable line

After providing power to the PCDU, the startup process of the system began. After performing a built-in self-test, the PCDU entered operational mode and immediately activated the power outlets of the nominal OBC boards. This lead to the startup of the OBSW which in return constantly polled the PCDU for updated information. The periodic polling of PCDU data by the OBC signals the PCDU that the OBC booted successfully and is alive. This startup process represented the first goal of the PSS test, i.e., to show that the PCDU is capable of starting up on its own if power is provided via battery input lines.

Further in the test process, each PCDU equipment outlet was activated by direct commands (service 2) to the PCDU handler. The measured voltage at each outlet testifies that the PCDU was providing power to the correct equipment.

As a final proof that the PSS is capable of recovering from total power loss, a power outage scenario was simulated after system integration by cutting off all power sources from the system. While the battery cells were slowly drained, the system remained stable until the supply voltage of the PCDU reached the point where the PCDU shuts down. Some minutes after the shutdown occurred, external power supply was re-enabled. After a short period of hysteresis, the PCDU restarted and brought the OBC and the rest of the system back up again. From that moment on, the recharging of the battery was monitored. Throughout this whole scenario, the system remained stable except for the expected shutdown which demonstrated the robustness of the PSS.

This test was deliberately postponed in the overall functional system test plan to a point where the satellite was already fully integrated. Instead of running this scenario earlier in the flatsat environment, it was decided to include the OBSW PSS controller application into the test scope. The PSS controller was added in OBSW version G.13.0 and provides a better estimation of the state of charge of the battery. By including this controller component into the test scope, the test of this functionality could already be covered in one single functional test.

To summarize, Table 5.1.1 lists all tests performed for the PSS within the flatsat testbed.

Table 5.1.1: Overview of flatsat tests performed for PSS

Test Name	Subsystem/Type	Test Bench	OBSW Version
Installation of PCDU and Power Outlets Test	PSS	Flatsat	M.5.1
PCDU Over-current/Fuse Test	PSS	Flatsat	M.5.1
PCDU Command and Control	PCDU DH	Flatsat	M.10.2
PCDU High Priority Commands	PCDU DH	Flatsat	M.10.3
PCDU Housekeeping/Sensor Data Test	PCDU DH	Flatsat	M.10.5
Solar Panel Input Test	PSS	Flatsat	M.10.5
Battery Charging/Discharging Test	PSS	Flatsat	M.10.5
Battery State of Charge Estimation Test	PSS	Flatsat	M.10.5
Solar Panel Deployment Mechanisms Test	PSS	Flatsat	M.10.11
Full Recovery after Power Loss Scenario	PSS	FM	G.16.0

5.1.3 Device Handler Development and System Integration

As shown in Figure 5.1.6, the flatsat tests were used to jump-start the subsystem level integration process and later enable EMC tests as the first system-level test.

All onboard equipment was installed into the flatsat in the order given in Table 5.1.2. In principle, all redundant devices were installed as well not only one representative unit. The reason for this was to show that the complete system was working and not only partial installations. Moreover, redundancy switching between redundant units was tested that way.

Each time a new device was added, the OBSW team released a new OBSW version that supported the new equipment. The new OBSW release was rolled out to the flatsat OBC immediately for testing. An iterative process began in which the device handlers were refined step by step until all planned tests were passed. Each OBSW increment resulted in a new revision of the OBSW (signified by the last digit of the version number). The results of these tests were documented in a corresponding test report which at the same time served as the test procedure.

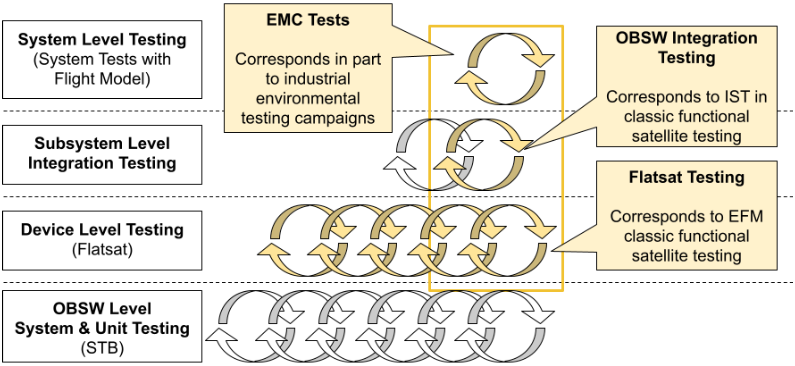

System Level Testing (System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing (Flatsat)
OBSW Level System & Unit Testing (STB)
EMC Tests
Corresponds in part to industrial environmental testing campaigns
OBSW Integration Testing
Corresponds to IST in classic functional satellite testing
Flatsat Testing
Corresponds to EFM classic functional satellite testing

Figure 5.1.6: Context of system integration and EMC testing in the OBSW development process

As described in section 5.5, the payload equipment was not part of this flatsat. For this reason, the payload tests as listed in Table 5.1.2 were performed with OBSW version Gemini.

Table 5.1.2: Overview of device handling tests performed in flatsat

Test Name	Subsystem /Type	Test Bench	Repeated Post-Integr.	OBSW Version
PCDU Basic Device Handling Test	PCDU	Flatsat	yes	M.5.1
Magnetometer Device Handling Test	ACS	Flatsat	yes	M.5.5
Magnetic Torquer Device Handling Test	ACS	Flatsat	yes	M.5.5
Star Tracker Device Handling Test	ACS	Flatsat	yes	M.6.5
GPS Receiver Device Handling Test	ACS	Flatsat	yes	M.8.1
TT&C Receiver/Transmitter Device Handling and RF Communication	CDH	Flatsat	yes	M.9.0
Pulse per Second (PPS) Characterization	CDH	Flatsat	no	M.9.1
Reaction Wheel (RW) Device Handling Test	ACS	Flatsat	yes	M.9.4
Fibre-optical Gyro Device Handling Test	ACS	Flatsat	yes	M.10.RC
PCDU Command and Control	PCDU	Flatsat	yes	M.10.2
PCDU High Priority Commands	PCDU	Flatsat	yes	M.10.3
PCDU Housekeeping/Sensor Data Test	PCDU	Flatsat	yes	M.10.5
Payload Onboard Computer (PLOC) Device Handling Test	P/L	Flatsat	yes	G.1.1
Payload Data Downlink Transmitter (DDS) Device Handling	P/L	Flatsat	yes	G.1.5

Since the first day of the flatsat test campaign, the mission control system (SCOS-2000) which was designated to be used for mission operations was used to command the flatsat. The TM/TC link was encoded using all CCSDS protocol layers. In times were no TT&C radio equipment was installed in the flatsat, the link was established by bypassing the RF modulation/demodulation and directly connecting the checkout equipment (RF-SCOE) to the OBC's CCSDS boards. Later on, as soon as the TT&C radio was available, the full S-band RF link was used for all tests (see also Figure 5.1.2).

After finishing all device handling functionalities for the tests listed in Table 5.1.2, the integration of equipment into the main satellite structure began. At this time the capabilities of the OBSW progressed as far as shown in Figure 5.1.7. This state represents the basic device handling and communication functions which were targeted by the "Mercury" releases. As this stage was now complete, all further releases used "Gemini" as the new major version identifier.

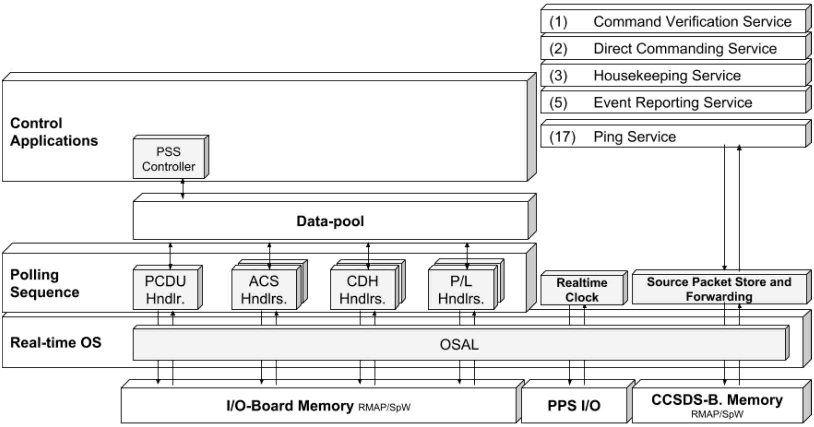


Figure 5.1.7: Functionalities of first Gemini release which featured all device handlers

The system integration process was divided into different stages. First, PCDU, battery engineering model and OBC were integrated into the main structure. Figure 5.1.8 shows the empty main structure of *Flying Laptop* with the prepared flight harness prior to integration of the PSS and the OBC. Each remaining piece of equipment of the flatsat was integrated, one at a time. For each piece of equipment, a reduced version of functional tests was repeated to check that the equipment was installed correctly. Table 5.1.2 shows which tests were repeated after integration. The performed integrated equipment tests served the following purpose:

- To briefly re-verify the functionalities already tested in flatsat tests, only this time with the flight harness and within the satellite main structure
- To check that the equipment was not damaged during the integration process
- To show the correct allocation of sensors and actuators as described in section 5.1.3.3

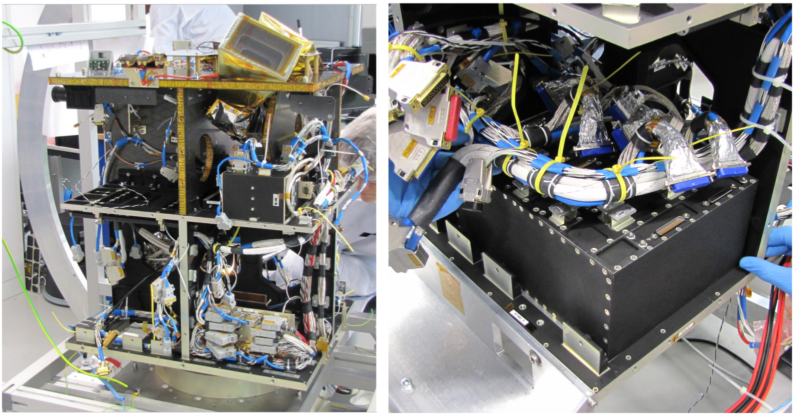


Figure 5.1.8: Preparation of the Flying Laptop main structure and harness (left) for the integration of the OBC (right)

5.1.3.1 Calibration of Equipment Simulation Models

Model calibration tests for the equipment simulation models were performed along with these flatsat tests to ensure that the simulation models sufficiently resemble their real counterparts. Table 5.1.3 lists each calibration test that was performed for each model.

For sensors and actuator models, noise figures and polarity were checked. For all power-consuming equipment, the simulation model's power usage was updated by the power consumption determined through the corresponding flatsat test.

Allocation checks were performed for all sensor and actuator models. Simulation parameters that determine the position and orientation of the sensor/actuator within the simulated satellite body frame had to match the specified position and orientation of the design.

For all simulated PCDU fuses, the activation currents were updated according to the overcurrent test listed in Table 5.1.1. The battery model, which is based on shepherd parameters was calibrated along the results of the battery charging and discharging test also mentioned in Table 5.1.1.

Table 5.1.3: Simulation model calibration tests performed in flatsat test bench

Model Name	Polarity	Noise	Power Cons.	Comm. Timing	Allocation	Additional Calibration
PCDU Model		X	X	X		Fuse-Current Limits
Sun Sensor Model		X			X	
MGM Model	X	X	X	X	X	
MGT Model	X		X		X	
Fibre-optical Gyro Model	X	X	X		X	
STR Model	X	X	X	X	X	
RW Model	X	X	X	X	X	
Battery Model					X	Shepherd-Parameters
Solar Panel Model					X	
TT&C Tx/Rx Model			X			
Temperature Sensor Model		X			X	
Payload Dummy Models			no calibration performed			

5.1.3.2 Tuning of Polling Sequence

As described in section 3.3.1 and visualized in Figure 3.3.2, the polling sequence table (PST) determines the underlying order in which the OBSW schedules device communication. To be more precise, the PST defines time-slots within a time interval in which the device handlers read and write to and from the I/O board buffers. The PST task repeatedly executes this polling sequence in a fixed frequency.

During flatsat tests, the polling sequence table was fine-tuned. Each onboard equipment's unique timing characteristics had to be respected. E.g., an RW needs cyclic commands and polling for housekeeping information in an alternating fashion. If the timing of these commands is skewed, the OBSW either misses the transmitted data or polls the device too soon. Both cases might lead to data loss because of missed device replies.

Since the PST contains time-slots and the number of available time-slots is limited, the PST needs to be arranged cleverly to meet the timing requirements of all individual onboard devices. For some devices, an interleaved scheme was necessary in which a device is polled only every other cycle to make room for larger data transmissions to finish on the data lines.

Since the arrangement of the PST influences the activities on the data harness lines, it was especially required to complete the fine-tuning of the PST before the EMC test took place because changing the PST could affect the outcome of the test due to interference.

5.1.3.3 Actuator Allocation and Polarity

The correct allocation of actuators and their polarity (the direction of their produced torque) is essential for attitude control. If these properties do not match the outputs of the attitude control software, i.e. the control algorithms, the attitude control loop is unstable or inverted which means it does not work properly.

[ECSS Committee 2012] defines polarity tests as a group of tests to verify the correct polarity of the functional chains, mainly A(O)CS, or equipment of the space segment element from sensors to actuators, through a number of interfaces and data processing nodes.

As described in section 3.2.2, *Flying Laptop* has two different actuator systems, magnetic torquers (MGTs) and RWs. One significant goal of flatsat and post-integration tests was to determine the correct allocation and polarity of all actuators through various methods. In general, there are a number of possible test benches or methods for allocation and polarity tests which are listed in Table 5.1.4.

Table 5.1.4: Possible means of determining actuator allocation and polarity through tests

Unit	Test Bench	Means of determining polarity	Means of determining allocation
MGT	Flatsat	• LED connector signaling positive/negative current	• LED connectors signaling on/off
MGT	S/C	• Compass/Teslameter • Auto-measurement MGM	• Compass/Teslameter
MGT	STB	• Auto-measurement MGM	• Simulation Parameter Check
RW	Flatsat	• Air-bearing table • Low drag bearing	• Vibration • Sound
RW	S/C	• Air-bearing table • Low drag bearing	• Vibration • Sound
RW	STB	• Observation of Reaction to torque-command • Auto-measurement w/ Fibre-optical Gyro	• Observation of Reaction to torque-command • Auto-measurement w/ Fibre-optical Gyro

Determining Reaction Wheel Allocation

The four *Flying Laptop* reaction wheels (RWs) are numbered from RW0 to RW3. The OBSW communicates with each wheel individually through a fixed address space within the I/O board memory (see Figure 5.1.10). Each wheel has a designated position and is mounted in the RW assembly as shown in Figure 5.1.9. The individual position within the assembly frame determines the effective vector of the generated torque. The polarity of each wheel is indicated by their direction of spin (clockwise or counter-clockwise) which is visible from an imprint on the housing of the RW unit itself. This means that from this visible information, the polarity of the RW itself could be determined. However, the end-to-end polarity including signs of commanded torque generated by OBSW ACS algorithms through the I/O board and the internal harness could not be tested with means available to the team.

To determine the correct RW allocation, a high wheel speed was commanded from the OBSW using the device handler which is assigned to a specific wheel and the reaction of the correct wheel was verified.

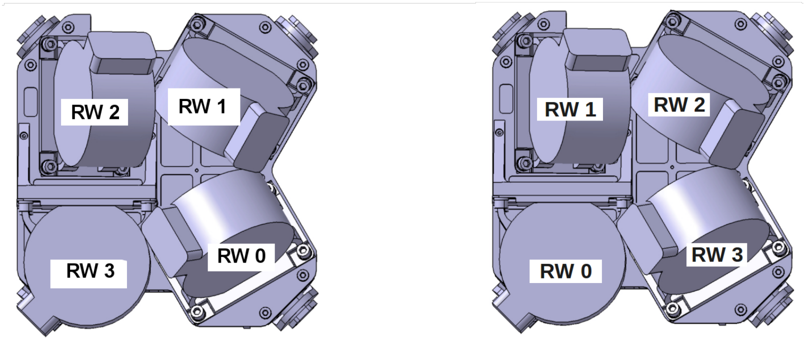


Figure 5.1.9: Discrepancy between physical allocation of RWs when mounted in RW assembly; Left: Allocation identified in flatsat test; Right: Allocation identified in post-integration flatsat test

The reacting RW could either be identified by touching the wheel's housing and finding the one that was vibrating or by identifying the wheel which produced sound. It turned out that the method of touching each wheel is unreliable for the given circumstances. The RW assembly uses dampeners to isolate the rest of the satellite structure from vibrations making it difficult to identify each wheel from touching it. For this reason, the method of identifying each wheel by sound was chosen for these tests.

Post-integration tests showed that the allocation of RWs, when commanded by the OBSW, changed in comparison to pre-integration flatsat tests. This was most likely caused by differences between test harness and real harness or confusion of connectors during integration. The discrepancy was found by spinning up one RW at a time and identifying the one that produces sound using a binaural stethoscope.

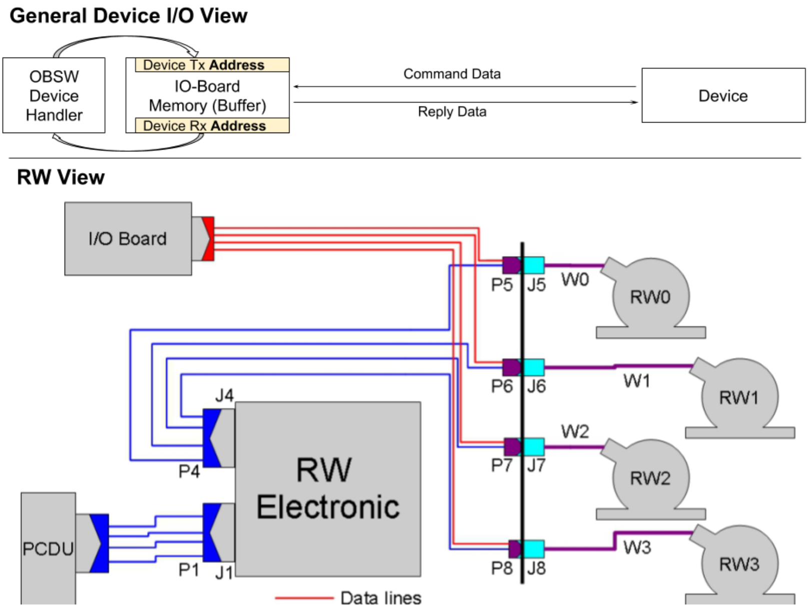


Figure 5.1.10: Wiring of individual RWs with the internal satellite harness (bottom); data flow of RW commanding from OBSW to RW unit (top)

As visible in Figure 5.1.10, the confusion must have occurred in the red data lines which are routed inside the satellite's harness and are used to command each RW from the OBC's I/O boards. Most likely this was caused by a manufacturing error of one of the harnesses or by connecting the wrong connectors P5 through P8 (see Figure 5.1.10) during the integration of the RW assembly.

The issue was solved by reassigning the RW handler addresses in OBSW to the updated allocation. With this workaround, a disassembly of the already integrated satellite and switching connectors could be avoided.

Allocation and Polarity of Magnetic Torquers

Even more critical than the RW allocation is the correct allocation and polarity of the MGT coils because they are essential for *Safe* mode attitude control. An error in the allocation of the MGTs could lead to early loss of mission because the satellite is unable to spin-down initial rotation rates. Supposing that spin down of initial rates worked, it is still necessary to correctly orient the solar panels to the Sun using only magnetic torquing.

Flying Laptop uses three redundant magnetic coils to produce a local magnetic dipole which interacts with the Earth's magnetic field to generate torque (two of the three coils are illustrated in Figure 5.1.12). Just as for commanding of the RWs, the OBSW can activate each coil individually by issuing the corresponding command with the correct I/O board addressing. For pre-integration flatsat tests, connectors with test LEDs as seen in Figure 5.1.11 were attached to the output of the MGT unit instead of the real magnetic coils. This offered the advantage to see the effect of a coil command visually.

As shown in Figure 5.1.11 each connector has two pairs of LEDs; one pair represents one redundant coil. The red LEDs indicate negative coil currents while the green LEDs indicate positive coil currents.

For post-integration tests, the LED connectors were replaced by the real magnetic coils and the correct output had to be determined by measuring the produced magnetic field of each coil using magnetometers. This was done by correlating two independent measurements. The first measurement was done using a probe-teslameter and the second by using the satellite's own onboard magnetometer sensors. Figure 5.1.12 demonstrates the orientation of the magnetic dipole and magnetic field generated by MGT1 and MGT2 (green fieldlines) and the location of the teslameter probe measurement (yellow). The purple arrow **m** in Figure 5.1.12 represents the corresponding dipole moment.

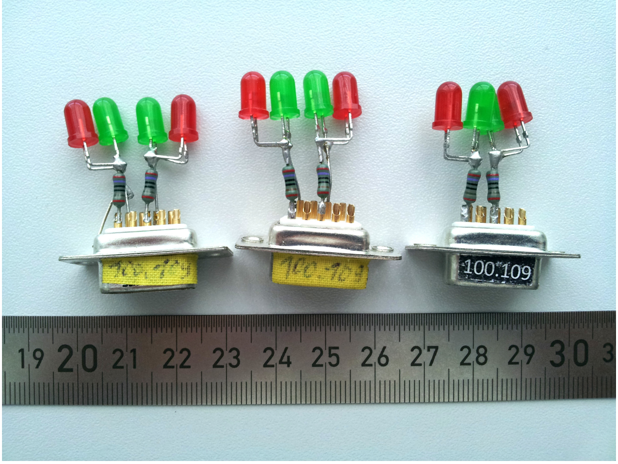


Figure 5.1.11: LED connectors used to indicate polarity of MGT commands during device handling tests

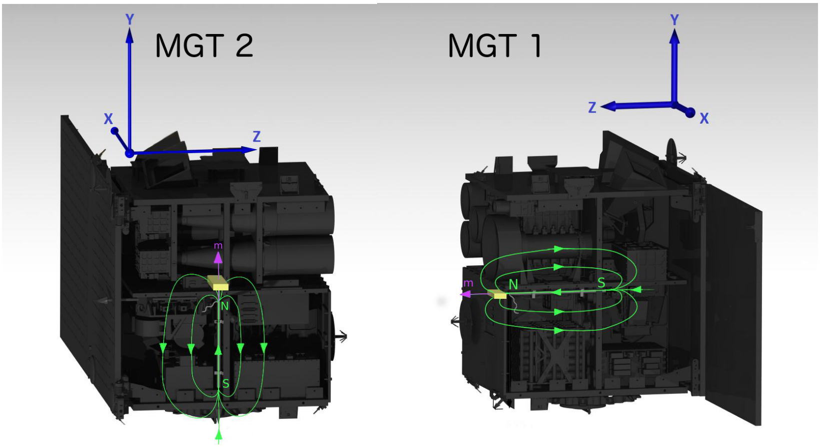


Figure 5.1.12: Magnetic field generated by MGT coils measured by probe-teslameter and internal onboard magnetometers

Cross-checking magnetic field measurements of the magnetic field produced by each coil with measurements from the internal magnetometers verified the correct coil assignment and at the same time increased confidence in the correct integration of the sensors themselves. The influence of an active MGT coil on the onboard magnetic field measurement is shown in Figure 5.1.12 and Figure 5.1.13.

In order keep this example simple, only MGT coils in z- and y-direction are shown in Figure 5.1.13. Of course, *Flying Laptop* also possesses torquer coils in the x-direction. The process for testing each coil direction is exactly the same as demonstrated below for the y-direction.

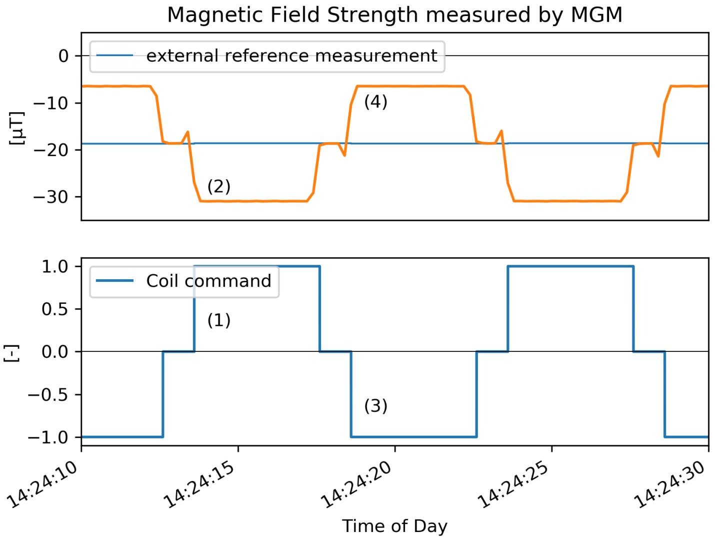


Figure 5.1.13: Influence of magnetic field generated by MGTs on onboard magnetometer measurement over a typical activation timespan (4,5 seconds) (top) Magnetic field strength as measured by onboard magnetometer and reference magn. field strength; (bottom) MGT command in Y-axes

Figure 5.1.13 shows the magnetic field strength in the y-direction as measured by the onboard magnetometer within the body-frame of the satellite and in addition an external reference measurement expressed in the same body-frame. It can be seen that in the phases when the torquer is inactive, the onboard magnetic field measurements match exactly the external reference.

As also visible from Figure 5.1.13 a positive coil commanded (1) leads to a distinct increase in the magnitude of the internally measured magnetic field (2) due to the superimposition of both the natural external magnetic field and the one generated by the torquer in the vicinity of the sensor. Likewise, a negative coil command (3) leads to a decrease in the magnetic field strength (4). This sequence was repeated for each coil one by one.

The sign of the superimposed field is the determining property for polarity tests. In an analysis step after the test, the correctness of the measurement had to be determined. This was done manually by tracing the direction of the magnetic field concerning the location of the sensor as shown in Figure 5.1.12 for each MGT coil and axis.

Summary

In summary, the actuator allocation and polarity tests performed for MGTs were sufficient to ensure that the commands generated by the attitude control software had the intended effect. The results of *the Safe* mode plausibility test (described in section 5.4.2) and data gathered shortly after launch in space also suggested the correct allocation and no further modifications had to be made to the actuator allocation and polarity.

For RW polarity, the picture is different. While RW allocation could be verified sufficiently by the use of stethoscopic inspection, the end-to-end polarity of RW commands remained unverified because these kinds of tests were regarded to be too costly.

The agreed strategy was to secure RW polarity by carefully commissioning them in orbit. Since *Safe* mode attitude control relies on MGTs as actuators, the RWs could be checked out, one by one, in orbit while the satellite remained in *Safe* mode. A short test was performed as part of RW commissioning in which only the polarity of each wheel was verified by looking at the reactions in spin rate produced by a single RW command. Once the polarity of each wheel was known, higher pointing attitude control could safely be enabled because the remaining functionalities were already verified in the STB.

5.1.3.4 Workaround for PCDU Deployment Functionality

The following example is intended to demonstrate the ability of the OBSW developers to respond to unexpected issues in a dynamic and creative way at a point in time where the addressing of similar issues would have caused major challenges in traditional satellite projects. The issue for this example arose during system testing of the PSS and specifically during the test for the solar panel deployment procedure with the fully assembled main structure.

Solar panel deployment for *Flying Laptop* works through a melting wire mechanism. The PCDU outputs a current which heats the melting wire through four dedicated heating resistors. Once the melting wire is ruptured, a prestressed spring brings the two solar panels into their final position similar to a swing-door mechanism. This procedure has to be performed once at the beginning of the mission to provide enough power for the satellite. In the original design of the *Flying Laptop*, the PCDU handles solar panel deployment autonomously based on internal timers.

A problem was identified during testing of the PCDU functionalities: The deployment timer of PCDU which determines how long the PCDU waits to begin the deployment procedure and sends current through the heating resistors was faulty because of a specification error.

The specification states that the deployment timer shall be active only during the first start-up of the PCDU. A factory reset is required to reset and re-enable the timer. This specification was, of course, meant to prevent the PCDU from trying to deploy the solar panels on every reboot. For *Flying Laptop*, however, it was not feasible to perform a factory reset just before launch because this would also remove vital, non-volatile settings within the PCDU.

To solve the problem, two options were considered. To return the unit to the manufacturer and change the firmware of the PCDU, or to implement a workaround solution for the OBSW. Returning the unit to the manufacturer would have been costly and also risky because this would have meant having to partially disassemble the satellite. Fortunately, the solar panel deployment procedure of the PCDU could also be remotely triggered by the OBSW using a specific PCDU command. By applying this trick, the deployment currents could reliably be controlled by the OBSW. The OBSW team quickly implemented a prototype of an additional, internal OBSW deployment controller which solves the issue and at the same time offers more flexibility and control over the deployment process to the operator compared to the PCDU functionality.

This additional controller OBSW component was not part of the original design. However, by shifting this logic away from the PCDU into the OBSW, the functionality also became more testable because it was now under full control of the OBSW. This redesign represented a late change to the overall functional architecture of the satellite system but also improved the design significantly. It was a result of the team learning throughout the testing process.

The example demonstrates the ability of the OBSW development approach to deal with unexpected late changes to the design and also the ability to draw benefits from it. While the problem was identified in system-level tests, the solution was the conceptual change of the OBSW but did not affect the other levels (see Figure 5.1.14).

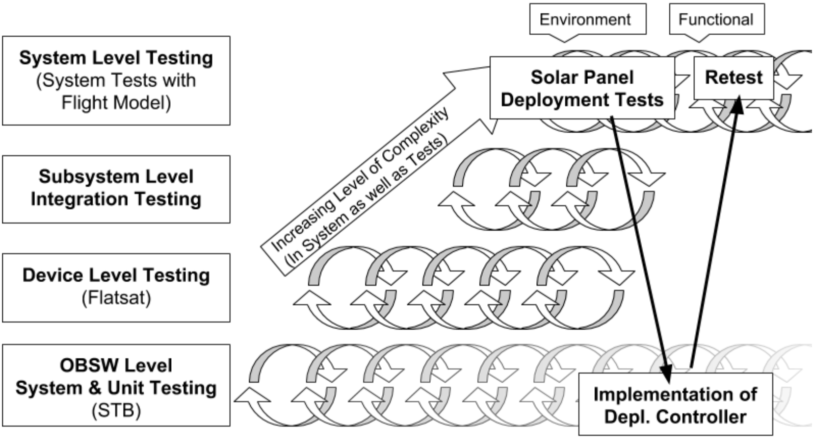

Environment
Functional
System Level Testing (System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing (Flatsat)
OBSW Level System & Unit Testing (STB)
Increasing Level of Complexity (In System as well as Tests)
Solar Panel Deployment Tests
Retest
Implementation of Depl. Controller

Figure 5.1.14: Context of the example of deployment controller implementation in OBSW development process demonstrates how late changes to the OBSW design are enabled with the presented approach

5.1.4 Electromagnetic Compatibility

Once the complete system was assembled, the electromagnetic compatibility (EMC) of all onboard equipment in their final arrangement needed to be demonstrated. EMC testing was also aiming to show that there are no effects of cross-talk between equipment through the satellite harness. For this purpose, tests described in [Lengowski et al. 2015] have been performed at Airbus facilities (depicted in Figure 5.1.15) but also in the University's cleanroom to measure:

- electromagnetic influence of devices communication on harness lines
- isolation of radio frequency spectrums used by onboard radio equipment

Table 5.1.5 shows an overview of all performed EMC tests.

Table 5.1.5: Overview of performed EMC Tests

Test Name	Type	Environment	OBSW Version
Identification of passive intermodulation products of onboard transmitters	EMC	Cleanroom	G.11.6
Harness radiation analysis	EMC	Cleanroom	G.11.6
Analysis of interference of TT&C transmitter and GPS antenna	EMC	Airbus DS	G.11.8
Electromagnetic auto-compatibility test	EMC	Airbus DS	G.11.8
Identification and measurement of static magnetic fields	EMC	Airbus DS	G.11.8

The harness radiation analysis was performed while the satellite was still in its final preparation phase. At this time, the multi-layer-insulation which usually covers most parts of the satellite main structure like visible in Figure 5.1.15 was yet to be attached. This way an electromagnetic probe could better reach critical locations of the harness. Various positions were probed to identify areas where emissions from harness lines were significant. This was especially required for *Flying Laptop* because some high-frequency harness lines were left unshielded. The impact of these radiated emissions on neighboring equipment was assessed and additional insulation was added if required.

For the electromagnetic auto-compatibility test (shown in Figure 5.1.15), all components needed to be actively communicating on their data lines to find overlapping frequencies of emissions. The test procedure for this test merely consisted of activating each single device handler by command and subsequently watching out for data transmission error events. Fortunately, no such events were seen during this test.

In addition, the static magnetic fields of the satellite were measured in the EMC test chamber. Onboard equipment may contaminate magnetic field measurements, required for *Safe* mode attitude control, with their static magnetic fields. This could be caused by either high internal currents or parts that have been magnetized through contact with permanent magnets. This measurement was deliberately performed inside the EMC test chamber because it provides isolation from external magnetic fields. However, in the case of *Flying Laptop*, no critical static magnetic fields were found.

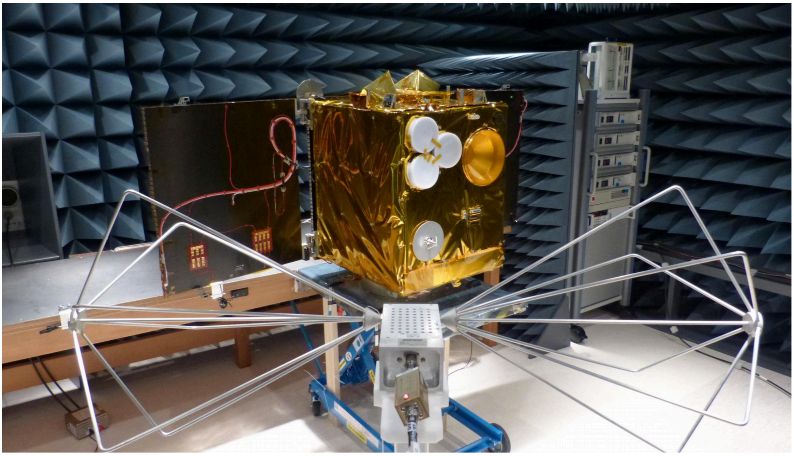


Figure 5.1.15: Flying Laptop inside electromagnetic compatibility (EMC) test chamber (Airbus DS facility); Dipole antenna in the front is used to measure radiated emissions in the near field and to detect transmitter intermodulation from different angles [Lengowski et al. 2015]

5.1.5 Benefits of System Test Driven Approach in early System Integration

The tests described in this chapter are common industry practice. Flatsats, also sometimes called electrical functional models (EFM), are common in classic industry practice as well. The main difference in this case, however, is that OBSW development was started together with system testing.

★ Building a flatsat is a mandatory part of this approach that guarantees electrical compatibility of equipment and enables untrained developers to familiarize themselves with the system.

In a classic approach, the flight software would be finalized before system testing begins. Developing device handlers while performing flatsat tests enabled the OBSW to grow gradually with each piece of equipment that was added. In contrast, the big bang integration approach applied in classic satellite development can lead to unexpected problems when the finished flight software is integrated for the first time in the built system. These problems can lead to costly rework iterations in the typically strongly departmentalized industry organizations.

Usually, all test chamber activities like e.g. EMC tests, TBTV and vibration tests are performed together in one big environmental test block and at one external test facility. The approach for *Flying Laptop* allowed splitting these tests in external test facilities into different singular tests like e.g. the EMC test described in 5.1.4.

- ★ The approach for *Flying Laptop* brought greater flexibility in choosing the appropriate test facilities for environmental tests because time constraints are less of an issue since the focus of OBSW development could be shifted at short notice

5.2 System Mode Transitions

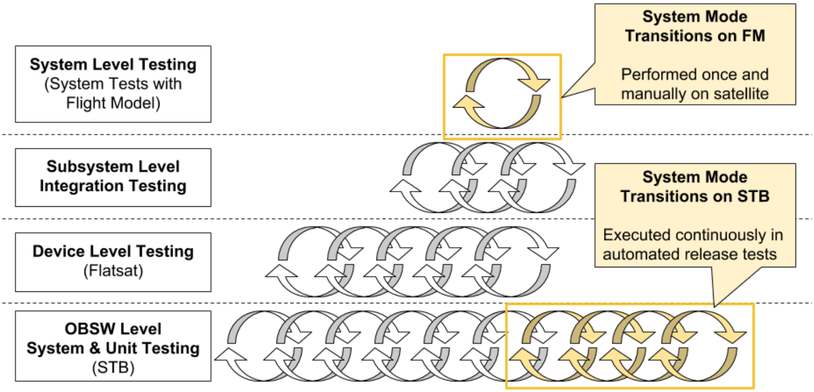

System Level Testing
(System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing
(Flatsat)
OBSW Level System & Unit Testing
(STB)
System Mode Transitions on FM
Performed once and manually on satellite
System Mode Transitions on STB
Executed continuously in automated release tests

Figure 5.2.1: Context of system mode transition tests with respect to test levels

To support the upcoming tests, controller components like the TCS controller were required. The control functionalities rely on system modes (see section 3.3.3) to be able to perform their control tasks. At least the system modes to operate the satellite during the thermal vacuum test were required next. As described in section 3.3.3, system modes define modes for every component in the system. The *platform abstraction model* allows defining the state of each component for a given system mode through mode tables. A list of defined system modes for *Flying Laptop* is provided in Appendix C.

Figure 5.2.2 shows the additional OBSW components that were implemented to enable system mode transitions. Besides the general *platform abstraction model* which holds the mode tree, the mode management logic and the event manager were added. The event manager was necessary at this point in order to enable OBSW components to broadcast their mode transition events to the rest of the system.

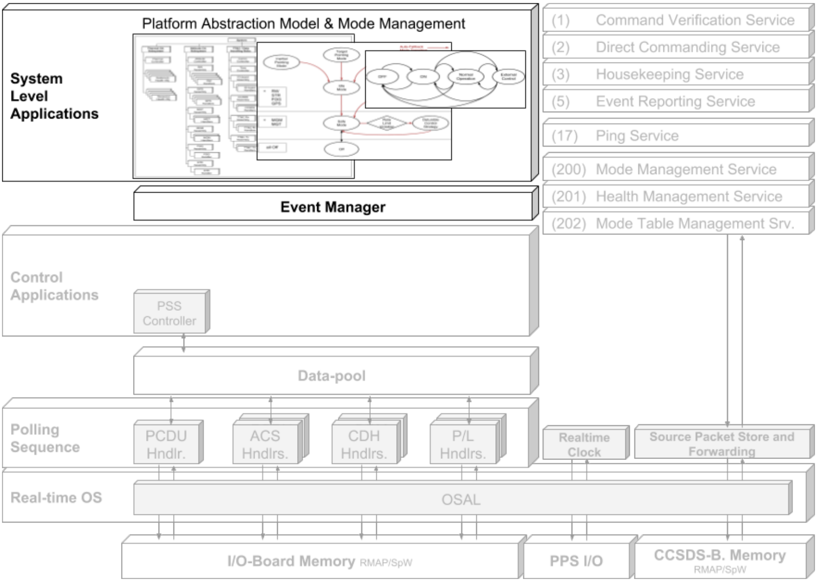


Figure 5.2.2: Addition of the system-level functionalities of the OBSW which include the platform abstraction model, event manager and mode tables

The goal of the system mode transitions test is not only to verify that the system can operate in all defined modes but also to demonstrate that the system can perform all mode transitions without errors. This is a significant difference because the number of all possible mode transitions, i.e. all allowed combinations of starting mode and target mode is significantly higher than just the number of defined system modes. It is required to test all possible combinations because the transitional logic within the components can produce unexpected effects in different combinations. Also, timing issues have to be considered here which makes performing these tests mandatory. To make these tests repeatable for all OBSW releases, they were implemented in MOIS procedures and later PyOps scripts (see section 6.5).

A simple example of such a test procedure and the involved steps is given in Figure 5.2.3. In Figure 5.2.3 the numbers next to the transition arrows going from one mode to another are the order in which the test sequence is implemented. The PSS in this example is in *Off* mode at the start of the procedure. The first test step is to command it into *Boot* mode, a special mode which is only required during the startup of the OBSW. From *Boot* mode, the next transition will go to the *Default* operational mode and then back to *Off* mode. The test procedure has now gone through every existing PSS mode, but this is not the end of the procedure. As explained above, there are a lot more transitions which also require testing. E.g., transitions from *Off* directly to *Default* or from *Default* to *Boot*, etc.

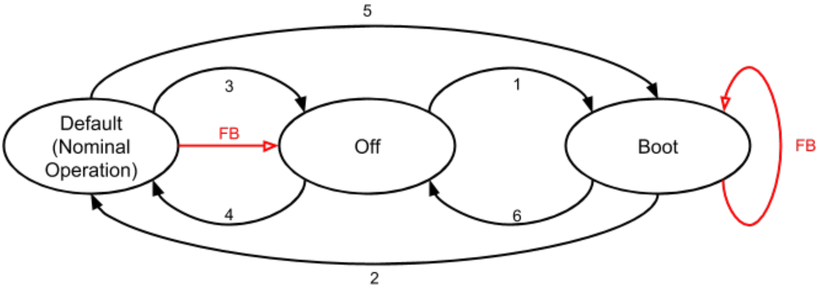


Figure 5.2.3: Mode transitions test procedure for the PSS; **FB**= fallback transition

Table 5.2.1 is a list of all performed mode transition tests and the number of single test steps needed in total. This emphasizes the scale of these automated mode transition tests which are executed for each release.

Mode transitions of controllers are more straightforward than, e.g. device handler mode transitions. Their mode transitions do not affect the outside world, in other words, their transitions have no side-effects other than interaction with the data-pool. They merely execute different control algorithms based on their current mode. A controller mode transition is merely a switch of control algorithms. This is why the transitions of controller components received less attention during OBSW testing and are not included in Table 5.2.1.

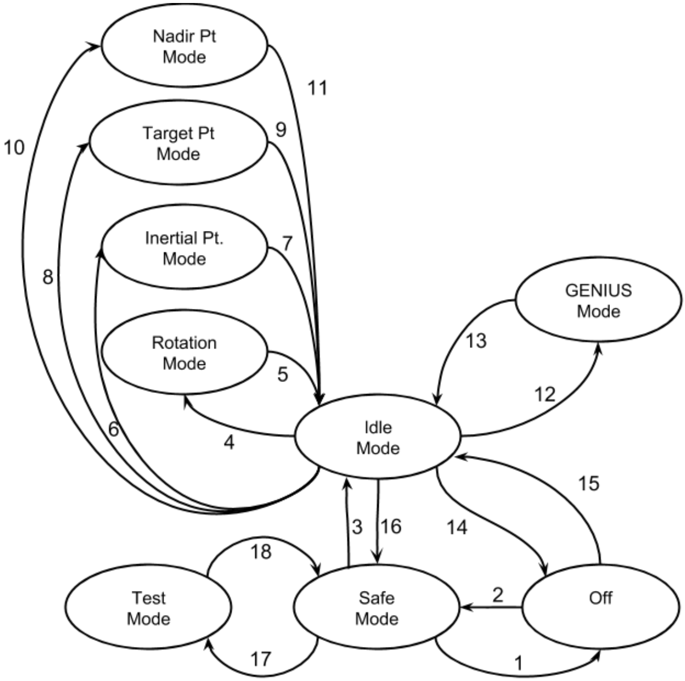


Figure 5.2.4: Sequence of mode transition test procedure for the ACS

Another more complex example is the ACS since the ACS has more modes than the PSS the amount of possible mode transitions is significantly higher. Note that the mode transitions given in Figure 5.2.4 do not necessarily correspond to the ACS modes given in section 3.2.2 and Figure 3.2.3. Modes like *Test* mode or *Rotation* mode are not used operationally but are defined for testing purposes. The regular system mode tables used for *the Flying Laptop* mission do not contain these testing modes.

Just as for the PSS, all possible mode transitions of the ACS are visible in Figure 5.2.4. For the ACS, the test procedure starts in *Safe* mode. Note that some mode transitions are not possible which limits the total number of transitions per test listed in Table 5.2.1. An attempt to command the ACS from *Off* to *Nadir Pointing* mode e.g. would be rejected by the mode management service. Similar procedures exist for all tests listed in Table 5.2.1. Likewise, all fallback mode transitions need to be tested as well. This, however, is part of the tests described in section 6.2 since fallbacks need to be triggered by failures and cannot be simply commanded.

Table 5.2.1: Amount of system mode transition tests summarized

Test Name	Test Targets (Components)	Transitions per Component	Modes p. Comp.	Total No. of Transitions
Device Handler Mode Transitions	17	9	4	153
Assembly Mode Transitions	12	6	3	72
PSS Mode Transitions	1	6	3	6
ACS Mode Transitions	1	18	9	18
TCS Mode Transitions	1	6	3	6
P/L Subsystem Mode Transitions	1	35	13	35
TTC&CDH Subsystem Mode Transitions	1	6	3	6
System Mode Transitions	1	231	25	231

In traditional flight software architectures, mode transitions are performed by flight procedures. That means either through command stacks issued by mission control or in the form of onboard control procedures. The OBSW uses an unconventional approach to mode changes where a lot of the logic that is traditionally performed in the form of commands is hidden within the mode transition logic of the OBSW components.

- ★ This makes a testing mode transitions with high coverage mandatory. On the other hand, the overall number of flight procedures is reduced through the special OBSW design. That means that testing of regular flight procedures is simplified.
- ★ Since mode transition tests are based on mode change commands, they can easily be automated through the technology described in 6.5.

5.3 Thermal Vacuum Tests

The next step in the development of the Flying Laptop OBSW was to implement thermal control capabilities to support the upcoming thermal vacuum test. The thermal vacuum test (also called thermal balance/thermal vacuum or TBTV test) was performed in the thermal vacuum chamber of DLR in fall of 2014. Figure 5.3.1 shows Flying Laptop inside the thermal vacuum chamber.

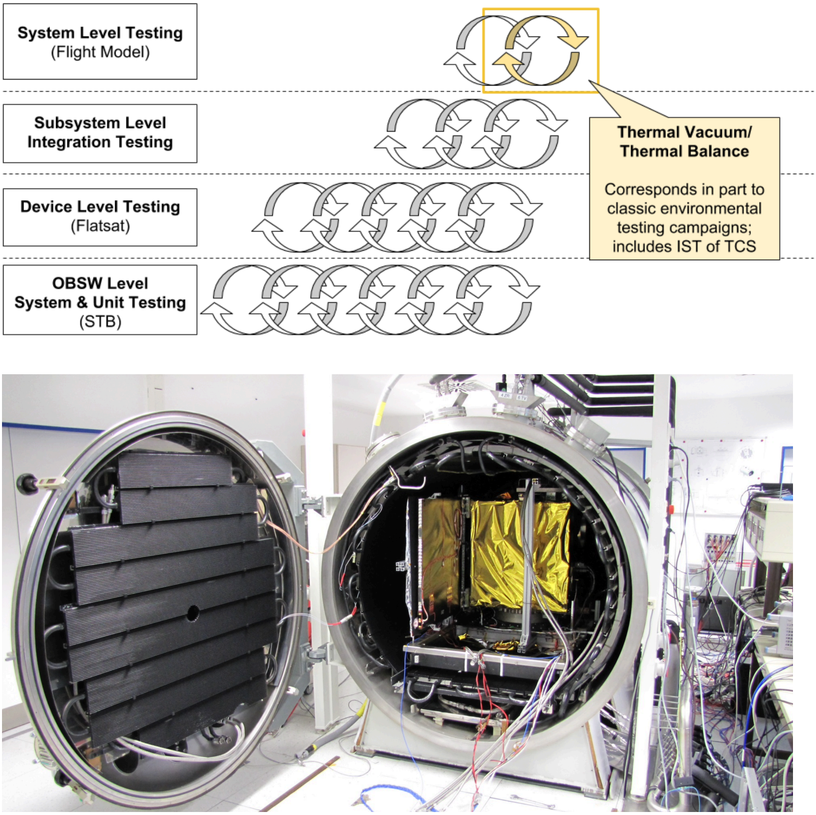

System Level Testing
(Flight Model)
Subsystem Level Integration Testing
Device Level Testing
(Flatsat)
OBSW Level System & Unit Testing
(STB)
Thermal Vacuum/ Thermal Balance
Corresponds in part to classic environmental testing campaigns; includes IST of TCS

Figure 5.3.1: Context of environmental tests in OBSW development process (top) and *Flying Laptop* inside thermal vacuum chamber at DLR facilities for TBTV tests (bottom)

The TBTV test was divided into two parts which were both performed in the thermal vacuum chamber but under different testing conditions:

- The thermal balance test
- The functional thermal vacuum

For each part, the test chamber was slightly modified. Temperature-controlled metal plates (or shrouds) were used for controlling the internal temperature during functional thermal vacuum testing. Thermal insulation was used during thermal balance tests where the goal was to create conditions close to the real radiative environment as encountered in orbit. The functional thermal vacuum test was performed with active thermal control because its purpose was to test the TCS, i.e. temperature sensors, heaters, and the thermal controller. The operational and non-operational temperature limits of the onboard equipment defined the test cases for the functional thermal vacuum test. Of course, each piece of equipment was thermally qualified individually over the complete temperature range which means that the equipment can operate at both temperature limits [Steinmetz 2016], [Lengowski et al. 2015].

Table 5.3.1: Different scopes of thermal balance and thermal vacuum tests

Thermal Balance	Functional Thermal Vacuum
• Determine static settling temperatures when the satellite is in thermal equilibrium • Validation of thermal simulation results and thermal models by comparing static settling temperatures and durations • TCS is passive • Temperature is determined by satellite (simulation of space radiative environment) • Hot and cold test cases (all equipment active/ only *Safe* mode equipment active)	• Show that temperature sensors and heaters are designed correctly • Functional testing of the OBSW TCS controller • Test the performance of the system at complete temperature range, e.g. temperature dependency of radio equipment, frequency stability, lubrication of RW bearings • TCS is active • Temperature is externally controlled via thermal vacuum chamber heating and shrouds

The thermal vacuum test is relevant to functional OBSW testing. For the thermal vacuum test the following OBSW functionalities were added (cf. phase 5 described in 4.3):

- Device temperature monitoring
- TCS controller

The job of the TCS controller is to keep the module temperature of a satellite within the allowed range, i.e. the range defined by each equipment's operational and non-operational limits. Since the *Flying Laptop* is a cold biased system, which means that it is thermally designed to tend to cool out instead of overheating, a passive TCS design was chosen. Passive TCSs are only capable of heating, not actively cooling equipment.

From the viewpoint of the TCS controller, *Flying Laptop* is divided into three separate thermal modules as shown in 5.3.2. The core module at the bottom, the service module in the middle and the payload module on top. The units located in each module share one redundant module heater and are usually similar with respect to the operational temperature ranges. Only the especially isolated battery has a different thermal temperature range than the rest of the satellite. The module temperature is the mean value of each temperature sensor allocated to the module. Instead of controlling each equipment temperature individually, the TCS controller controls the module temperatures by activating and deactivating the module heaters if the temperature gets close to the lower operational limit of active equipment within a module.

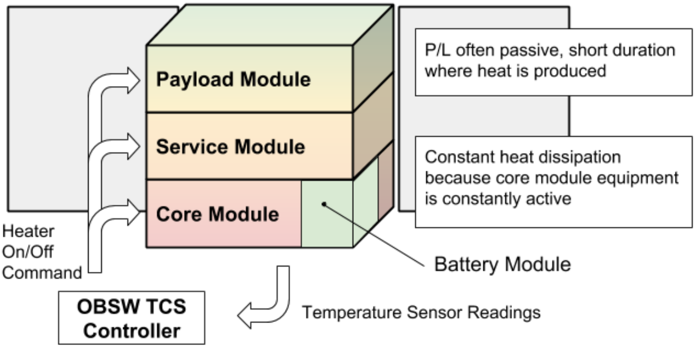

Payload Module
Service Module
Core Module
P/L often passive, short duration where heat is produced
Constant heat dissipation because core module equipment is constantly active
Battery Module
Heater On/Off Command
OBSW TCS Controller
Temperature Sensor Readings

Figure 5.3.2: Thermal design of Flying Laptop which uses three separate thermal modules

To monitor temperatures during TBTV tests, the monitoring service (ECSS PUS service 12) was added to the stack of OBSW components. The monitoring service enables monitoring of values and gradients in the data pool. For TBTV tests only temperature values sampled from temperature sensors were monitored. Later, the monitoring service was, of course, used to monitor all data pool variables. If a monitored data pool parameter or its gradient exceeds or falls below the configured limits the monitor of the parameter issues an event with the corresponding severity level. This event is sent to ground by the event reporting service but also broadcasted to subscribed OBSW components through the event manager in order to notify, e.g. failure handling (see section 6.2 for further details).

5.3.1 Wrong Calibration of a Single Temperature Sensor

The multispectral camera system MICS is the primary camera system of *Flying Laptop*. Each of the three individual cameras has its own built-in temperature sensor. The readout of the sensor value is contained in the housekeeping information that the camera provides to the OBSW and is measured in counts (cts). One count is equivalent to the value expressed by the least significant bit (LSB).

To convert counts to a human-readable temperature, e.g., degree Celsius, a calibration curve is required. This calibration curve is determined by the linear or non-linear characteristics of the sensor and expressed by a polynomial function [Eickhoff 2016]. For the MICS, such a calibration curve was available since the manufacturer provided it as part of the documentation.

However, in post-integration tests, it was discovered that one of the three cameras, MICS near-infrared (NIR), provided a temperature value that was way off compared to the other sensors within the vicinity as clearly visible from the data given in Table 5.3.2. The proprietary test-software of the camera manufacturer did not show this behavior which is why this problem was not apparent in equipment-only tests. It was assumed that either the sensor was damaged during the integration of the camera or the test-software uses different calibration curves depending on camera type.

Table 5.3.2: Temperatures and raw values provided by each MICS camera in post-integration tests

	MICS Green	MICS Red	MICS Near-Infrared
Temperature	8.42 °C	9.0 °C	-7.0 °C
Counts (LSB)	211 cts	222 cts	38 cts

From the limited amount of samples that could be collected within the University cleanroom (where controlled room temperature is maintained), a correct calibration curve for the MICS NIR sensor could not be obtained. Also, the unit could not be brought into a thermal test chamber for this purpose because it was already integrated into the satellite. Disassembly, in order to perform single equipment analysis with the camera, was risky and would have caused significant delay and cost.

Fortunately, because of the agile nature of the system test planning and OBSW development, the TBTV test (described in section 5.3) could be used to collect enough sample points and to reconstruct an estimated calibration curve for the MICS NIR temperature sensor.

As demonstrated in Figure 5.1.14, the solution to this problem demonstrates that the presented approach enables the implementation of late solutions to problems that occur in functional system testing through OBSW changes that were relatively easy to implement.

5.3.2 Benefits of System Test Driven Approach

The TBTV tests themselves, as performed for *Flying Laptop*, did not differ from typical environmental satellite system tests. The difference in the system test driven approach was that the tests could be performed earlier. As mentioned in section 5.1.5, usually satellite manufacturers book environmental test facilities way in advance and in one block after the complete software integration process is finished. It would have been difficult for the *Flying Laptop* team to react to problems like the one described in section 5.3.1 if discovered in a late, monolithic TBTV test block.

- ★ Due to the application of the system test driven approach and as already described for EMC tests in section 5.1.5, no full OBSW was required to perform the TBTV test. Only the mandatory thermal control and monitoring functionalities of the OBSW were required which were implemented as needed for the test.
- ★ The test results could then be used to improve simulation models to further improve the TCS controller in separate STB simulation sessions.
- ★ The solution to the problem described in section 5.3.1 demonstrates how the approach enables to directly and efficiently handle unexpected difficulties.

5.4 Attitude Control Subsystem

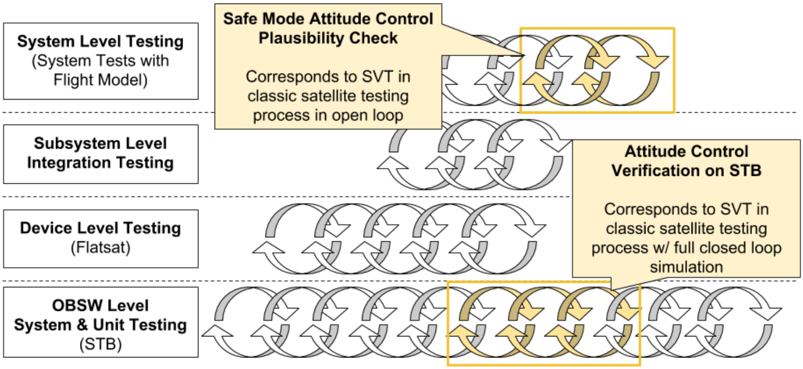

System Level Testing
(System Tests with Flight Model)
Safe Mode Attitude Control Plausibility Check
Corresponds to SVT in classic satellite testing process in open loop
Subsystem Level Integration Testing
Attitude Control Verification on STB
Corresponds to SVT in classic satellite testing process w/ full closed loop simulation
Device Level Testing
(Flatsat)
OBSW Level System & Unit Testing
(STB)

Figure 5.4.1: Context of attitude control subsystem test in the overall system testing process

System tests of the attitude control subsystem are amongst the most challenging tests to realize for a satellite, yet the correct functionality of attitude control is crucial to mission success. A failing attitude control system can lead to mission loss if the satellite is not designed to survive without proper orientation like *Flying Laptop*. Likewise, the proper operation of payloads can only be realized with a stable and controlled satellite attitude for most mission types.

Difficulties arise due to the lack of a space-like environment for ground tests which is required to generate suitable test conditions. Casually speaking, unlike a test car, a satellite cannot be taken for a test drive. This problem is even more significant the larger the satellite is. For smaller satellites, affordable mechanical test stands exist that use, e.g. air-cushion technology to create a semi drag-free environment. With larger satellites, these test stands are more difficult to realize and thus testing gets very expensive. Table 5.4.1 gives a comparison of available test strategies for ground-based attitude control tests and explains in which mission types these test strategies are usually applied.

Table 5.4.1: Comparison of different methods of attitude control verification methods

	Simulation	Complexity	End-to-End Verification	Usually applied in Mission Types
Test Stand/Chamber	no	medium/high	yes	• CubeSats & low budget
Closed-Loop	yes	high	yes	• high stakes missions • agency missions • novel systems
Open-Loop	yes	low	no	• low budget • systems with heritage

5.4.1 Flying Laptop Attitude Control Subsystem Tests

For the functional verification of *Flying Laptop* attitude control subsystem, an open-loop test approach was applied. Closed-loop simulation tests were additionally performed on the STB. With this combined approach, full ACS functionality could be verified without complex closed-loop test setups and SCOE for the FM. A strong emphasis was laid on making sure that crucial *Safe* mode would provide a reliable fallback in any emergency. To achieve this, *Safe* mode simulation tests were augmented by additional plausibility tests with the real spacecraft.

Although for *Flying Laptop* open-loop plausibility tests were performed, it is, in principle, possible to completely renounce testing attitude control with the real satellite and instead perform verification completely in simulation. Especially if the system and simulation models have reasonable heritage, this is a valid yet risky verification strategy to pursue. While the overall goal of simulation tests was to show that

- the satellite provides detumbling capabilities from spin rates exceeding nominal limits.
- the satellite reliably orients the solar panels to the sun in *Safe* mode using only magnetic actuation.
- the satellite reliably orients the solar panels to the sun in *Idle* mode using RWs.
- the satellite is capable of fine pointing to any given ECEF coordinates or aligning to any inertial reference.
- the pointing error in pointing mode is below a maximum pointing error of 150 arcseconds for any (moving) earth target.

The goal of the additional plausibility tests is to boost the confidence gained by the results of simulation runs. The main purpose of these tests is to demonstrate that magnetic torque commands are generated correctly and show that sun direction determination is reliably working. Table 5.4.2 lists the performed open-loop tests for the attitude control subsystem performed with the flight model. They are further described in the following sections.

Table 5.4.2: Open-loop tests performed with flight model for attitude control verification

Test Activity	Test Bench	Chapter
Verification of magnetic commands in detumble attitude control	FM	5.4.2
Verification of magnetic commands in sun acquisition	FM	-
Detection of sun direction using sun sensor mockup	Mockup	5.4.3
Simulation of GPS scenario using GPS network simulator	FM	5.4.4
Simulation of star tracker star pattern using star simulation unit	FM	5.4.5

5.4.2 Verification of Magnetic Commands

To demonstrate the generation of correct MGT commands, the satellite was rotated while mounted in its rotatable test stand as shown in Figure 5.4.2. For simplification, only the rotation around the x-axis is explained in this two-dimensional example. However, as part of the full test, the other body axes were treated identically.

While manually rotating the satellite within the test stand the MGT commands generated from the active detumbling *Safe* mode controller were recorded for later evaluation. This section will provide information about how the conclusion that the generated MGT commands were correct was justified.

Since the detumbling control algorithm is comparably simple, it is possible to predict the expected commands if the local magnetic field is known, static, and sufficiently homogeneous. The local magnetic field around the satellite was measured before the test was started. The coordinate system visible in Figure 5.4.2 is not aligned with this local magnetic field. That means the local magnetic field **B** has components in all three spatial axes. The components in lateral x- and z-direction cause compass needles to point north, but a significant proportion of **B** was measured along the y-axis in Figure 5.4.2. For this reason, **B** is tilted in Figure 5.4.4 where the resulting MGT commands are visualized in a plane projection.

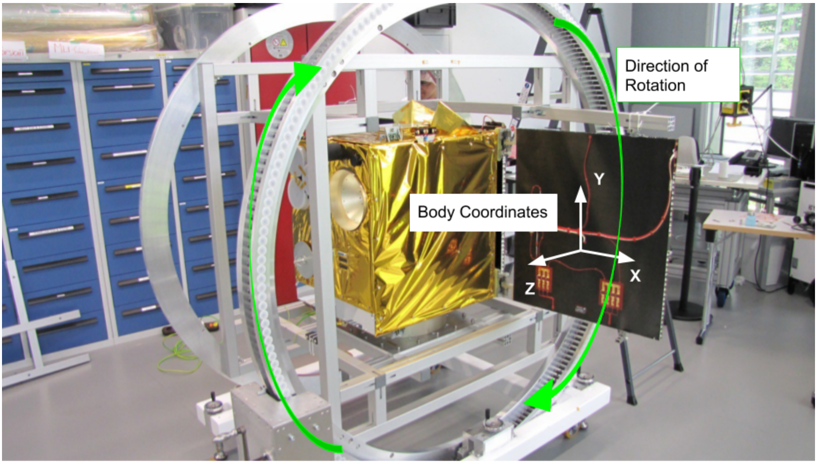


Figure 5.4.2: Flying Laptop mounted in rotatable test stand including coordinate system of body frame and direction of rotation for this example

The detumbling controller as part of *Safe* mode generates MGT commands which always counteract the derivative of the currently measured magnetic field (B-dot control law) [Stickler et al. 1976]. The magnetic field derivative is calculated by the difference between two consecutive magnetic field measurements.

The B-dot control law after [Stickler et al. 1976] uses the derivative of the measured external magnetic field $\dot{\mathbf{B}}$ which is always perpendicular to the magnetic field vector **B** and proportional to the rotational rate. In Eq. (1), **m** is the command vector that is generated by the control algorithm and commanded to the MGTs. It uses discrete saturated values determined by the sign of $\dot{\mathbf{B}}$. Eq. (1) and (2), $\mathbf{B}$, $\dot{\mathbf{B}}$, $\boldsymbol{\tau}$, and **m** are vectors expressed in the body frame of the satellite.

$$m_{x,\,y,\,z} \quad = \quad \begin{array}{rl} -1 \Rightarrow & \dot{B}_{x,\,y,\,z} > 1 \\ 0 \Rightarrow & \dot{B}_{x,\,y,\,z} = 0 \\ 1 \Rightarrow & \dot{B}_{x,\,y,\,z} < 1 \end{array} \qquad \text{Eq. (1)}$$

The resulting torque vector generated by MGTs is given in Eq. (2). It is the cross-product of **m** and the Earth's magnetic field vector **B**.

$$\boldsymbol{\tau} = \mathbf{m} \times \mathbf{B} \qquad \text{Eq. (2)}$$

In the example result of the test below, the satellite was rotated in the test stand shown in Figure 5.4.2 around its X-axis for 2 minutes. Rotating around one body axis simplifies the evaluation of test results since only two dimensions have to be considered.

Figure 5.4.3 shows the magnetic field strengths in three dimensions measured by the on-board magnetometers during the test. These measurements are the input of the B-dot controller which applies the control law Eq. (1). Also visible in Figure 5.4.3 is the absolute rotational rate derived from the magnetic field measurements by the attitude control software. For the whole duration of the test, a constant rate of the satellite was maintained of approximately 1°/s or ~0,017rad/s. However, the rate was not stable nor precise since the test stand was operated manually. Nevertheless, this test demonstrates that the rate estimation of the ACS software is correct. Figure 5.4.3 shows the produced MGT commands as generated by the B-dot law controller. In a manual evaluation step after the data was recorded, the MGT commands were plotted in a flat projection as shown in Figure 5.4.4 to verify the direction of **m**.

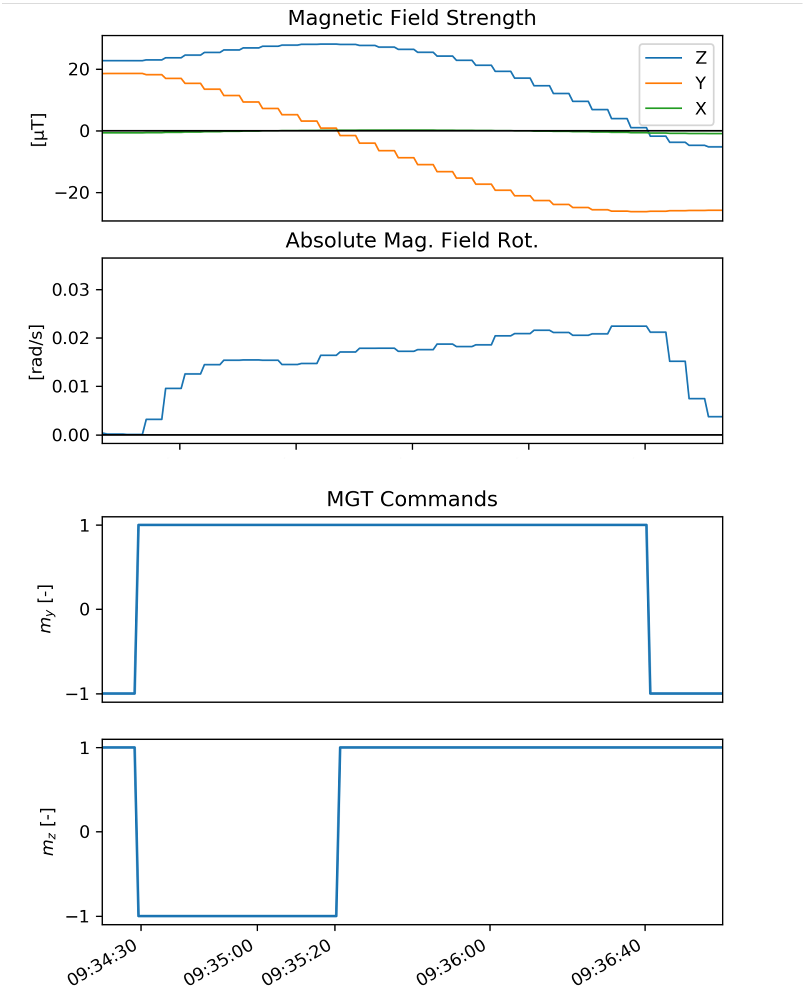


Figure 5.4.3: MGT commands produced by ACS controller (bottom) and measured magnetic field (top) in response to constant rotation around X-axis while in *Safe* mode (detumble strategy); MGT commands are individual for each MGT coil in X-, Y- and Z-direction within the body-frame of the satellite; note the distinct points where MGT coil commands Y and Z change in response to sign change in measured magnetic field

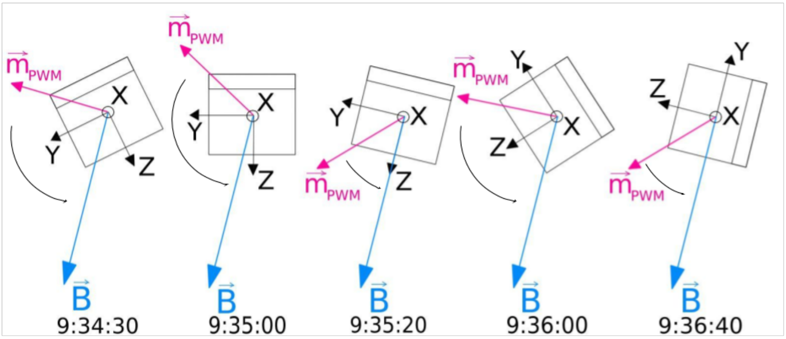


Figure 5.4.4: Orientation of satellite at different local times throughout the verification of MGT commands; Blue vector represents the Earth's magnetic field (from external measurement); Pink vector is the magnetic dipole moment resulting from torque commands; The satellite is slowly rotated clockwise, while the resulting torque would rotate the satellite counter-clockwise (right-hand rule)

In Figure 5.4.4 **m** was reconstructed from the recorded test data of Figure 5.4.3 for five distinct points in time during the test. By application of the right-hand rule, it can be seen that while the rotation of the satellite within the test stand was clockwise, the resulting torque Eq. (2) would rotate the satellite counter-clockwise. This is the expected result for a correct B-dot controller to reduce rotational rates of the satellite (detumbling).

In Figure 5.4.4, **m** suddenly jumps at 9:35:20. This jump is clearly visible in the MGT commands shown in Figure 5.4.3 where the z-axis command changes from -1 to 1. This jump is a result of discrete -1/0/1 commanding which is expressed in Eq. (1). The B-dot law controller always tries to apply the maximum achievable torque in the given orientation.

This end-to-end test demonstrated that the *Safe* mode ACS controller produces the correct commands when the satellite is slowly rotating. It is important to note that for this test no shortcuts were taken and no assumptions made which makes this test very important for boosting confidence in the correct implementation of the *Safe* mode ACS.

5.4.3 Sun Sensors

To demonstrate the suitability of the sun sensor design and the processing chain of sun sensor readings into a sun direction by the attitude control software, a portable testbed which comprises a 3D-printed mockup in combination with spare sun sensor cells representing the arrangement of the sun sensors and a portable STB was developed as part of a student thesis [Becher 2017].

The testbed was used in a field experiment where it was exposed to real sunlight rather than artificial light. This solution is more feasible than putting the whole satellite in a sun simulation test chamber. With this mockup not only the sun sensor system and sun direction determination algorithm could be verified in an isolated end-to-end test but also the sun sensor simulation model could be calibrated with the collected real-world data. Figure 5.4.5 shows the arrangement of the spare sun sensor cells within the mockup. Note that for a sufficient representation only the sensor's orientation but not the relative distances between sensor cells is relevant.

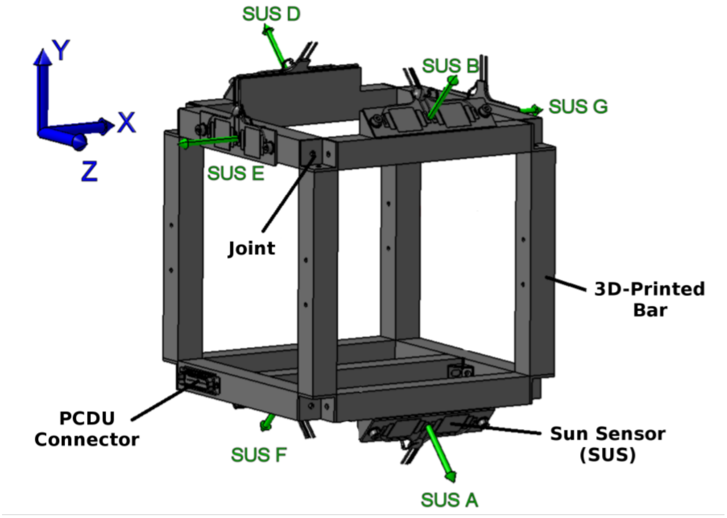


Figure 5.4.5: Sun sensor three-dimensional mockup for end-to-end test of sun detection algorithm

The mobile testbed is a modified version of the STB. It uses a workstation computer to execute the satellite simulator and a Xilinx ML505 development board which runs the OBSW. Also, a portable installation of the mission control system SCOS-2000 was installed to control the OBSW via TM/TC link. This way, real-world sampled data is processed by the OBSW and an end-to-end data processing chain is realized without exposing the *Flying Laptop* FM to an unclean environment.

Different scenarios were tested including the determination of the static sun vector from various angles but also the extraction of the angular rate by rotating the mockup using a small motor. The deviations of the detected sun vector by the OBSW and the chosen reference lies between 0.6° and 6.5° depending on the selected scenario. The scenario in which the sun direction is identical to one of the main body axes (x, y, and z) is found to be the least accurate [Becher 2017].

In summary, by experiments conducted with the sun sensor mockup, it was shown that

- the sun sensors were correctly assigned in OBSW.
- the computation of the sun vector by the OBSW with respect to the body coordinates of *Flying Laptop* was sufficiently accurate.
- the derivation of the angular rate from the sun vector was also sufficiently accurate.

The sun sensor simulation model which was calibrated by data collected from the mockup experiments produces a sufficient match of simulated sun sensor current and expected real current (see sections 6.1.2 for a detailed evaluation of the sun sensor model). The increased effort to improve the sun sensor model by constructing the three-dimensional mockup was deliberately invested to increase confidence in the ability of the OBSW to reliably find the Sun as a key capability for *Safe* mode attitude control.

5.4.4 Testing the GPS Receivers

GPS receivers have become standard equipment for LEO satellites since the GPS network has been established. With the addition of other navigation networks like GLONASS and Galileo, satellites can rely on a broad variety of navigation solutions. The *Flying Laptop* carries three GPS receivers which serve different purposes:

- They provide an accurate time source for onboard data-handling and precision attitude control.

- They provide position and velocity information for the satellite's attitude control system which, in combination with time information, enables to perform precise target pointing but also orbit determination.
- For *Flying Laptop*, the GPS receivers are also part of an experiment called GENIUS which aims to extract satellite attitude information from three different GPS antennas.

To synchronize the time across different onboard equipment, the three onboard GPS receivers generate a pulse-per-second (PPS) signal which is received and redistributed by the OBC.

The synchronization of time is essential for precision target pointing. Onboard sensor values like attitude, position, and velocity are measured by different sensors at different points in time. For this reason, sensor values are timestamped so they can later be fused correctly. The time-series of sensor values need to be extrapolated by the attitude control software using time-stamping on the basis of a common time reference.

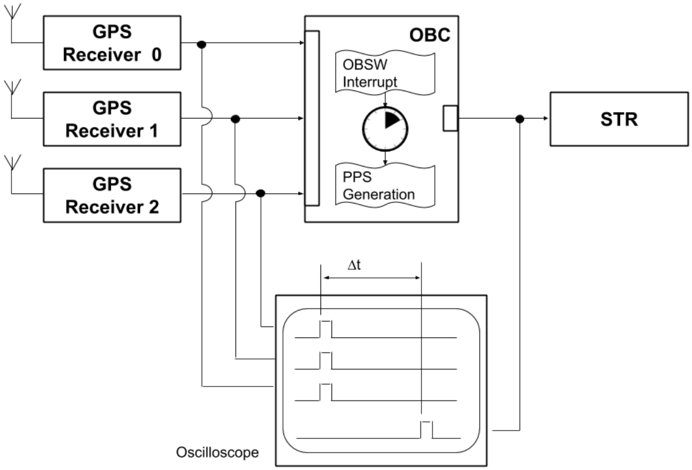


Figure 5.4.6: Measuring end-to-end PPS drift and jitter by wiretapping the PPS signals generated by GPS receivers and OBC using an oscilloscope

The GPS PPS signal enables the OBC to synchronize its internal time to the GPS provided time. On the other hand, the OBC sends out a similar PPS signal to the star tracker unit to also synchronize timestamps generated by the star tracker to the common time source.

A dedicated test to characterize both jitter and drift of the PPS signals was performed early within the flatsat test environment. The basic setup of this test is shown in Figure 5.4.6. The PPS lines from all GPS receivers to the OBC and the PPS line from the OBC to the star tracker unit were probed using an oscilloscope. As long as a GPS receiver has a valid GPS solution it generates the PPS signal which is then used by the OBSW to adjust the internal onboard time. From an OBSW perspective, the forwarding of the PPS comes down to handling an interrupt generated by the incoming GPS pulses and issuing the outgoing pulse to the star tracker. This process generates the delay Δt which was measured and is listed in Table 5.4.3 for each combination of GPS receiver and OBC (nominal and redundant). Table 5.4.3 also lists the average period and jitter of the pulses which were measured as a result of this test.

Table 5.4.3: Pulse per second characterization test results

	GPS Rec. 0	GPS Rec. 1	GPS Rec. 2	STR
Avg. Period	1.000002 s	1.000003 s	1.000002 s	1.000005 s
Std. Deviation (Jitter)	5.7 µs	5.8 µs	5.3 µs	7.1 µs/9.2 µs OBC N/OBC R
Δt OBC (N/R)	0.463/0.299 s	0.549/0.297 s	0.533/0.297 s	-

The PPS characterization test showed that the incoming PPS signal is correctly received from the GPS receivers and the outgoing PPS signal to the STR is generated correctly. The test also provides important figures required to increase attitude control accuracy.

In addition to the PPS characterization tests, a series of tests with the fully integrated satellite have been performed which had the goal to demonstrate, that the receivers could all receive the GPS signal properly. For this purpose, two methods were applied to feed a valid GPS signal to the satellite's antennas.

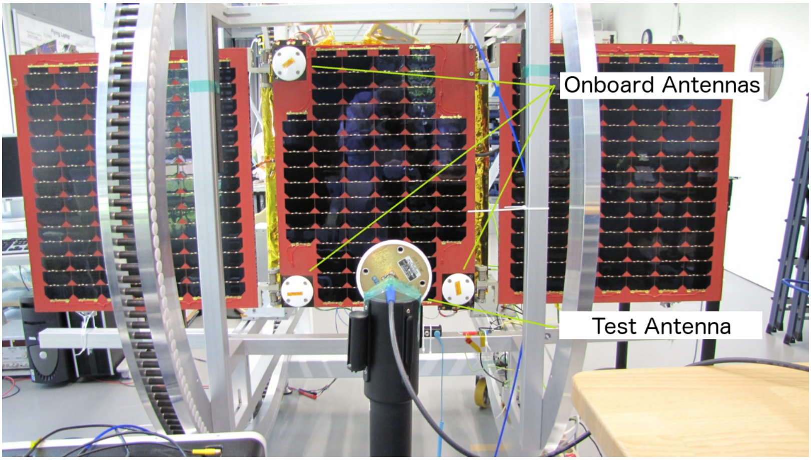


Figure 5.4.7: Flying Laptop solar panel side and onboard GPS antennas; test antenna for GPS signal repeated from outside (front of picture, outside antenna was placed on institute rooftop)

The first method was to retransmit the real GPS signals which were received through an outside antenna into the clean-room where testing took place. The second method was to use a GPS network simulator capable of simulating various GPS scenarios. Figure 5.4.7 shows the satellite inside the clean-room during a GPS retransmission test. The three onboard antennas are arranged in an L-shape and each feed to a single onboard GPS receiver.

Figure 5.4.8 shows a similar test setup. The difference is that the GPS signal is now generated by a GPS network simulator. With GPS retransmission, the GPS solution generated by the GPS receivers is static because the satellite does not move during ground tests. This problem can be solved by the usage of the mentioned GPS network simulator. The GPS network simulator generates a GPS signal identical to the one the satellite would encounter in orbit where high relative velocities between receiving and transmitting antenna occur. This test enables tests with more realistic scenarios compared to retransmitting the signal of a stationary antenna.

By the GPS tests described above, it could be shown that *Flying Laptop* was capable of receiving the GPS signal from all three GPS antennas on ground and in simulated orbit conditions. The GPS receivers were capable of compensating the Doppler shift caused by high relative orbit velocities and also provided a stable GPS solution through GPS satellite handover.

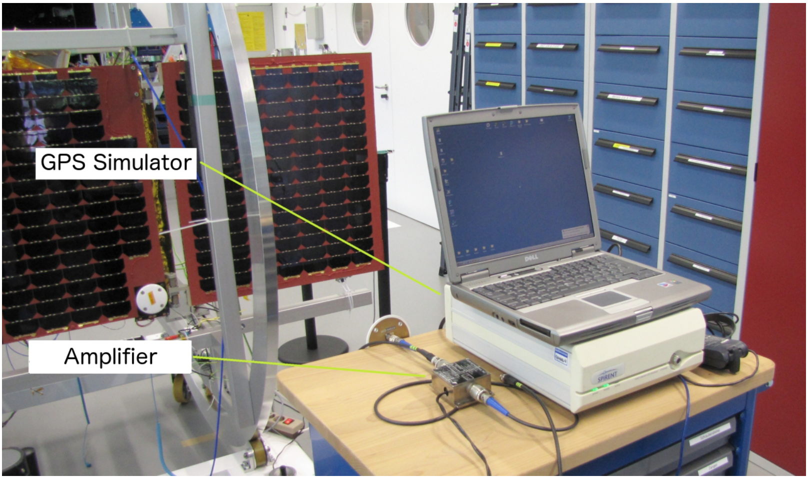


Figure 5.4.8: GPS network simulator unit, signal amplifier and sender antenna for simulation of different GPS scenarios

5.4.5 Difficulties with Testing Star Tracker

Flying Laptop uses two star tracker cameras whose independent solutions are merged by the OBSW. The star trackers of *Flying Laptop* showed unexpected behavior in orbit. During commissioning of the star cameras, it became clear that merging both attitude solutions generated by the two different star tracker cameras failed sometimes. This resulted in an invalid attitude. The problem only occurred if both cameras were used simultaneously and was never observed in ground tests. After some investigation, the problem could be attributed to an OBSW bug which only occurred with a statistical likelihood if both cameras take different durations to acquire a solution from their star images. If this is the case, both solutions get transmitted at a varying time but to the same I/O board buffer. Because of the way the buffer is implemented there were difficulties separating both messages under these conditions.

The *Flying Laptop* star tracker is a smart sensor which means it is a complex unit with its own sophisticated processing software and which uses the ECSS packet utilization standard protocol to communicate with the OBC. The two camera head units (CHUs) are connected to the central data processing unit (DPU) which performs all image processing and communicates with the OBC. The duration which the DPU requires to process a star image can vary between 553 and 653 ms depending on image complexity and how blurry the image is as a result of motion (i.e. how much filtering is required).

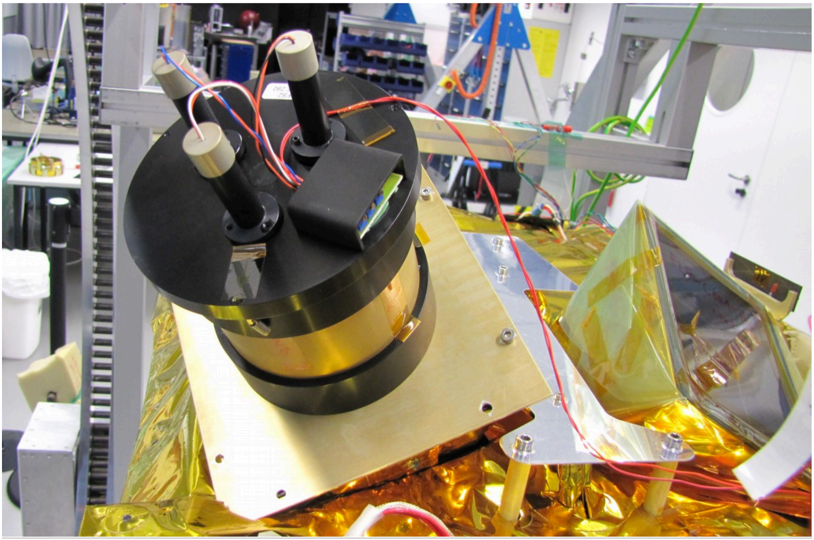


Figure 5.4.9: Star simulation unit used to project a static test star map; artificial star map is produced from three LEDs to a single star camera; the unit is attached to the rectangular baffle

As soon as the processing is finished, the solution is transmitted over the data line to the OBC's I/O board. Transmission happens unprompted without the OBSW polling for a reply within the limits of the minimum and maximum processing duration. This applies to both cameras individually if both are active at the same time. Thus, from an OBC point of view, the star tracker is essentially two sensors communicating over the same line without fixed polling intervals which creates a certain complexity in sorting out the received messages from the receive buffers.

For ground testing of the star cameras, a projector unit, depicted in Figure 5.4.9, was available which allowed projecting a simple star image consisting of three groups of stars to one camera. An impression of these three groups of stars is visible in Figure 5.4.10 along with real pictures from the star sky collected in orbit for comparison. It is clear that the fake images present a much more deterministic solution and is likely that the fake images cause less variation in processing time.

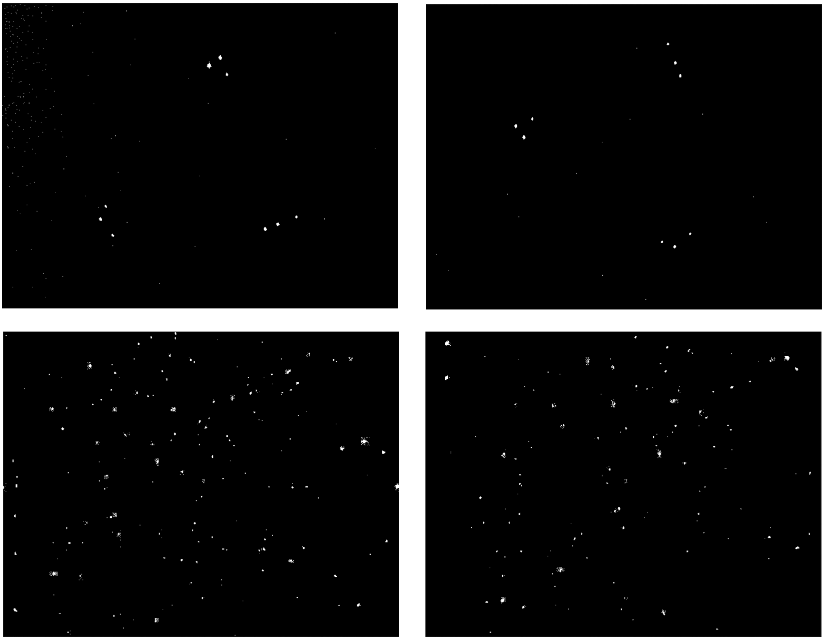


Figure 5.4.10: Static picture of the star map projector used for testing the star tracker system compared to real star images; Test pictures produced by star tracker CHU A (top left) and CHU B (top right) in cleanroom ground tests using star projector; Real sky map picture taken by CHU A (bottom left) and CHU B (bottom right) in orbit

In ground tests, only one camera head unit could be tested at a time. To test the other camera, the projector had to be removed from the first camera baffle and attached to the second CHU. Because of these limitations, it was not possible to notice the problem in functional system tests.

The inability to test both cameras simultaneously was classified as acceptable when the test methods were evaluated because it was planned to test the merging of both solutions in simulation. Simulation tests, on the other hand, did not produce any irregularities and always delivered a valid merged result. The used star tracker simulation model was too deterministic with respect to timing to reproduce the problem.

To summarize, the real star tracker's integration time fluctuates more than the simulation model's integration time. That means the simulation model's messages were received more uniformly by the OBC. The variations in integration time have been accounted for in the simulation model. However, there was an overlooked issue with the generation of randomness in the model thus the variation was not as significant as required to reproduce the real unit's behavior.

The conclusions which can be drawn from this example are that it is important for the approach presented here that simulation models of complex onboard equipment require a lot of attention to detail. Especially timing issues in communication are difficult to analyze and even more challenging to model because simulation models are normally scheduled at fixed intervals and cannot resemble timing behavior precisely. One method to improve model quality is to work closely with the manufacturer when creating a model in order to understand the internal software better. An alternative method of improving simulation model accuracy would be to use only proven models supplied by the manufacturer if available or models which were already used in other missions where the same equipment was used (this could be achieved by the simulation portability standard mentioned in section 3.1).

However, not being able to use both star cameras simultaneously during the beginning of the *Flying Laptop* mission did not have a major impact on mission success. Although the bug had an effect on the availability of the star tracker sensor system, all mission goals could be fulfilled even by using only one star camera. Since the bug was solved later on, both cameras could be used normally during the rest of the mission.

Safe mode attitude control was not affected by this problem which was the primary goal of functional system testing described for this approach. This is another argument for the efficiency of the presented approach. The problem was later solved by an OBSW patch which could be tested before upload to the spacecraft by using a refined star tracker simulation model.

5.4.6 Evaluation of System Test Driven Approach for Verification Attitude Control

As laid out in section 4.1, the classic approach towards attitude control verification is to perform costly closed-loop tests with the real satellite FM. These closed-loop tests involve sensor stimulation EGSE in order to achieve the required feedback from co-running virtual simulations. Usually these tests are planned as part of SFTs. Low budget missions cannot rely on these costly closed-loop setups and use drag-free test stands to verify attitude control. Due to the mentioned limitations of these test stands, these verification results have limited meaningfulness.

In the presented approach, almost all ACS tests were performed in full simulation using the STB. Only simple open-loop checks were performed with the real satellite FM for *Safe* mode attitude control.

- ★ The strong simulation focus for attitude control verification relieved the project from affording costly sensor stimuli EGSE and other SCOE.
- ★ By the early use of simulation tests for attitude control, the simulation models were improved and the refinement of the attitude control algorithms could continue until the launch of the satellite even when the FM was not available for testing anymore. Even while in orbit, the STB was used to further improve ACS pointing accuracy.

However, problems with the STR unit as described in 5.4.5 show that the approach of only using simulation the verify attitude control has limitations. Especially for complex equipment, the simulation models fail to fully resemble the real functional behavior. This was especially noticed when timing constraints have to be met.

In order to prevent such problems, it might be rewarding to invest increased effort in making sure that either the simulation models of complex equipment are excellent or to afford additional, more sophisticated SCOE.

5.5 Payload Tests

For the payload subsystem of the *Flying Laptop*, a different test approach was chosen because the development of payload software and systems was largely decoupled from OBSW development. Usually, as in many satellite designs, the payload is separate from the bus or platform system. This enables reusing satellite platforms for multiple missions and therefore saving development costs. However, the OBSW needs to control payload activity and thus at some point in the system testing campaign, the marrying of bus and payload segment is required.

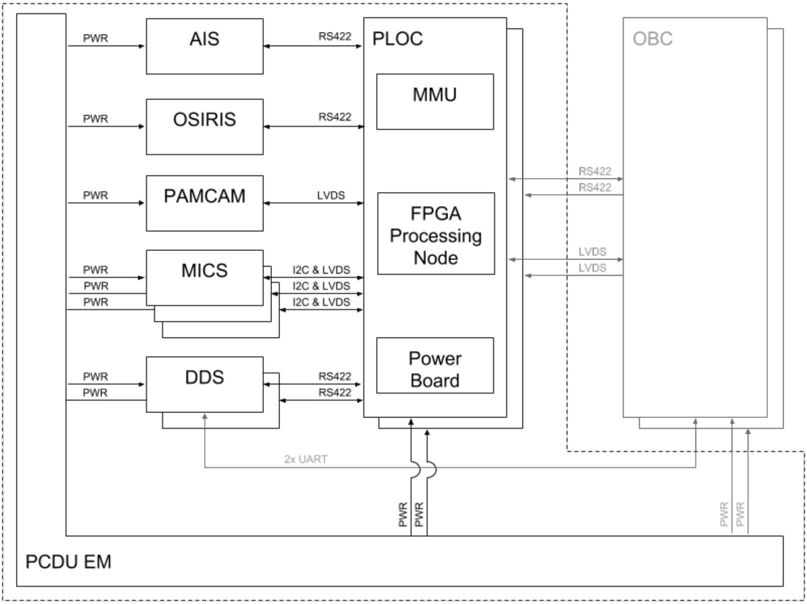


Figure 5.5.1: Flying Laptop payload subsystem/payload flatsat and OBC interfaces overview; PWR = power line; RS422, LVDS, I2C and UART describe the underlying digital signaling scheme

As visible in Figure 5.5.1, which shows the multiple payloads of *Flying Laptop* and their interfaces, the main interface between payload and bus segment is the command and control link between the payload onboard computer (PLOC) and the OBC. Almost all payload functionalities are commanded by the OBSW via the PLOC command and control line and the PLOC in this situation acts merely as a relay for the OBSW's commands. Yet, the PLOC maintains its own command and control protocol (see [Hagel 2018] for details). From the standpoint of OBSW testing, the PLOC is considered an external component just like any other onboard equipment.

As for all onboard equipment, the payload units have been individually qualified and functionally tested prior to integration into the satellite main structure. For optical instruments, i.e. the multispectral camera system (MICS) and panoramic camera (PAMCAM), characterization tests have been performed on the unit level which included recording of dark field, flat frame, spectral response, and noise figures. These unit level tests were performed using a custom payload testbed and custom software which did not include the real interfaces and processing software that will be used inside the satellite and for ground data processing. As a result, tests that include representable interfaces and software were still outstanding [Böhringer 2014].

Compared to the rest of the satellite, the payload subsystem was lagging behind in maturity. At a point where the remaining subsystems were already integrated and functionally tested the tests listed in Table 5.5.1 were still to be performed. For these tests the following bullets served as the baseline verification strategy:

- A separate payload flatsat was established because the payload subsystem was finished later than the already integrated rest of the platform. The payload flatsat served the purpose of compatibility tests between payload equipment and PLOC just like the platform flatsat served for compatibility tests between OBC and platform equipment. The payload flatsat is described in section 5.5.1.
- Payload equipment was simulated only as dummy equipment models for the STB. This was done to enable OBSW tests including payload mode-switching (see section 5.2) and failure handling (see section 6.2) but not fully simulated payload operations.
- Payload data handling and payload operations tests were performed in a semi-integrated state: The rest of the satellite and the PLOC were already integrated into the main structure while the payload units were mounted in the optical bench module which was not yet attached to the main structure.

- Payload data downlink and ground processing tests were performed with the fully integrated satellite, once the payload ground segment was ready.
- In parallel, alignment tests for MICS, PAMCAM, and laser terminal OSIRIS were performed.
- For troubleshooting and general support of payload development during mission time, an expansion of the STB with PLOC and payload EM hardware was planned (see section 5.5.2).

Table 5.5.1: Overview of performed payload tests

Test Name	Subsystem/Type	Test Bench	OBSW Version
OBC/PLOC functional test	Platform	Flatsat	G.1.1
OBC/DDS functional test	Payload	Flatsat	G.1.5
MICS functional test	Payload	P/L Flatsat	G.10.1
PAMCAM functional test	Payload	P/L Flatsat	G.11.8
OSIRIS functional test • Transmission of bit pattern	Payload	P/L Flatsat	G.20.2
AIS functional test using AIS signal generator	Payload	P/L Flatsat	G.20.2
Payload data handling test • Data take with all instruments • Storage of P/L data in MMU • Direct P/L data forwarding	Payload	FM	G.20.2
Payload data downlink test • Routing of P/L data to DDS Tx • Routing of P/L data to OSIRIS	Payload	FM	G.53.1
Camera alignment	Payload	FM	-
Full payload operations scenario	Mission Scenario	FM	G.52.3

The reader should bear in mind, that this shift in testing baseline occurred right in the middle of the already ongoing system testing campaign as a reaction to delays in payload subsystem development. It is the strength of this agile approach to enable such drastic replanning of integration and test on relatively short notice.

5.5.1 Payload Flatsat

As mentioned above, the progress of payload development was lagging behind due to technical difficulties which made the separation of platform integration and payload integration necessary. Integration of platform equipment was already started before payload equipment was ready for flatsat tests to buy more time for payload technology to mature.

Only the device handling of the PLOC handler was tested in the flatsat as described in section 5.1. This was done to make sure that once the OBC is integrated, it has already demonstrated to be compatible with the PLOC. Since only the PLOC communicates directly with the individual payload equipment, the payload flatsat could be built individually without concerning the rest of the platform. Once the payload subsystem was ready and internally tested through the payload flatsat, the rest of the satellite and the payload subsystem were married.

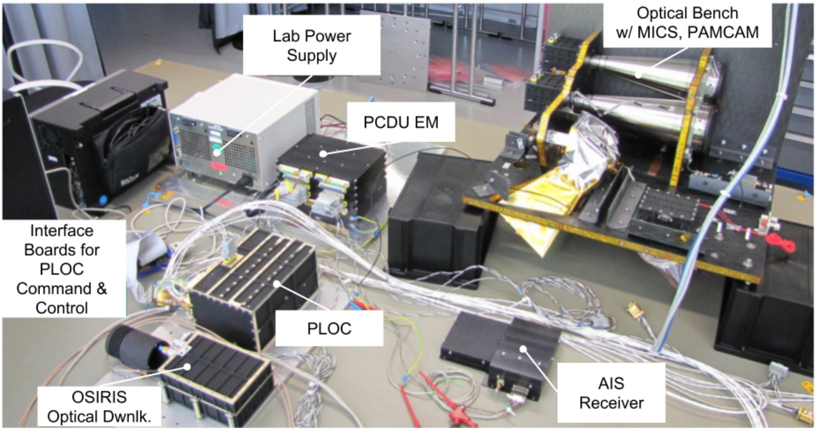

Lab Power Supply
Optical Bench w/ MICS, PAMCAM
PCDU EM
Interface Boards for PLOC Command & Control
PLOC
AIS Receiver
OSIRIS Optical Dwnlk.

Figure 5.5.2: Payload flatsat including optical bench, PCDU EM and PLOC [Hagel 2018]

As a result of this strategy, the PCDU flight model was obviously also already integrated into the satellite structure and therefore not available for the payload flatsat. Because a flatsat resembles all electrical interfaces and not only data lines realistically, a replacement unit for the PCDU flight model was required in order to provide accurate power supply conditions. For this purpose, the PCDU engineering model (EM) which had an almost identical electrical layout was repurposed and used as payload flatsat PCDU. Figure 5.5.1 shows the interfaces and payload equipment that comprised the payload flatsat. Figure 5.5.2 shows the original arrangement of payload equipment including the partly integrated optical bench.

The main challenge of the payload data handling subsystem compared to the rest of the satellite data handling were the high data rates that are produced by some types of payload instruments. Especially for camera systems, the amount of generated instrument data is significant. Usually, as is the case for *Flying Laptop* this data is stored in a mass memory unit (MMU). The MMU of *Flying Laptop* is integrated into the PLOC as visible in Figure 5.5.1 [Hagel 2018].

All optical payloads of *Flying Laptop* are hosted on the so-called optical bench which provides enough stiffness to tolerate the shrinking and expansion of the surrounding material due to temperature changes.

5.5.2 Payload Testbed for Mission Support

Because the simulation equipment models of payload equipment were implemented as dummies that could be switched on and off and provided some fake housekeeping data but no real payload data, a separate payload testbed was required for substantial payload software development. This additional software development is especially important to test OBSW and PLOC software updates during the mission because no other means of testing payload activity was available after the satellite was launched.

The payload testbed was added to the simulation environment which already hosted the STB. It consists of payload engineering models that were previously used for qualification tests. The separate payload testbed was attached to the STB and is connected to the STB's OBC via its hardware I/O board EM. That means, while using the STB, the tester can dynamically switch between using either simulation models or real payload hardware in the loop as the representation of the payload subsystem [Hagel 2018].

Unfortunately, the payload testbed was a late addition to the simulation environment and could not be used for OBSW development during system testing. However, it will be a vital part of OBSW development later in the *Flying Laptop* mission.

5.5.3 Possible Benefits for Payload Instrument Development

In regular industry projects, the development of the satellite platform and the development of the payload instruments are usually decoupled. Sometimes, the payload instrument is even supplied by different vendors because the technology used in these instruments is highly specialized. For that reason, the integration of the payload instrument usually happens late in the project; sometimes it is the very last integration activity after the platform is already fully tested.

The readiness of the payload segment of the *Flying Laptop* was also late but in this case for different reasons. The effect, however, was the same. *Flying Laptop* payload equipment was not available for functional testing early on.

★ Through the application of the agile system test driven approach, it was possible for the payload subsystem to catch up in maturity with the rest of the platform subsystems.

Although the testing of payload for *Flying Laptop* happened late, the approach in principle supports both modes of integration (early or late integration). Early integration means to include the payload development into the agile testing and development process early on to take advantage of the same benefits described in earlier chapters.

Another positive aspect is the capability to exchange payloads flexibly because the system design is less static as compared to traditional satellite missions.

★ As demonstrated for the *Flying Laptop*, the approach makes late changes to the composition of the payload segment possible, even if the platform is already fully tested.

5.6 Operations Scenarios

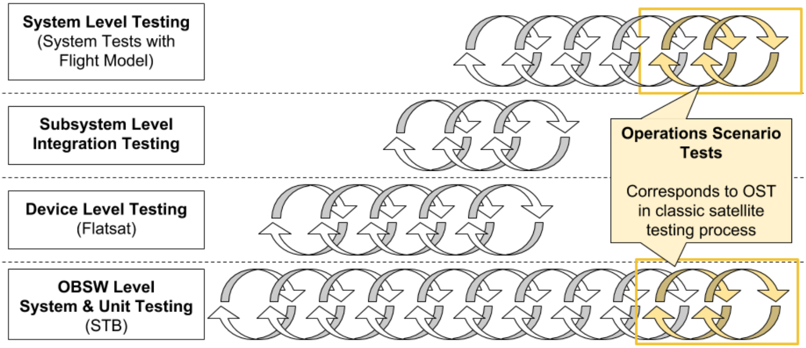

System Level Testing
(System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing
(Flatsat)
OBSW Level System & Unit Testing
(STB)
Operations Scenario Tests
Corresponds to OST in classic satellite testing process

Figure 5.6.1: Context of operations scenario tests in OBSW development process

As the OBSW steadily proceeded towards the final release version (Apollo), full operations tests became possible. The goal of full operations tests was to operate the satellite as it would already be launched into orbit using the entire ground data segment and without any shortcuts (cf. [Wertz et al. 2011]).

The idea behind the full operations test was to combine many disciplines in one big test prior to launch. Not only were these kind of tests demonstrating that real mission scenarios could be performed with the developed system but it was also seen as an opportunity for the operations team to get to know the satellite. Furthermore, the flight procedures of the operations team were verified in these tests.

While all system tests until this point in time focused on specific topics, subsystems or functionalities, the full operations test now covered all integral aspects of daily operations. The tasks performed during these tests were ranging from processing housekeeping data, operating the payloads to handling unexpected failure events. Because of this holistic approach, these tests were also an excellent opportunity to further improve the OBSW's usability.

To operate the satellite without any shortcuts meant that all involved systems (not only the OBSW) needed to work as expected. It was not allowed to hot-patch the OBSW, bypass any rules or restart the satellite while the test was ongoing. Any action that is not possible in orbit could not be used as a means to solve problems encountered during this test. If the full operations test could not be continued because of such a problem, the test had to be aborted.

Table 5.6.1: Advantages of performing operations tests in simulation (STB)

Simulation Tests	Real Satellite Ground Tests
• Full closed-loop attitude control simulation possible • Test of attitude control maneuvers	• No use of SCOE which stimulates attitude control sensor input • Attitude control is silenced in order to prevent deterioration of actuators
• Simulation of thermal parameters and effects of thermal control through thermal model	• Temperature sensor measurement values will remain at room temperature • Thermal control is enabled but will not actively heat equipment because of controlled environmental conditions
• Availability of error injection to cause e.g. device errors (described in section 6.2)	• Generation of device errors is not possible or advisable (see section 6.2)

Table 5.6.2: Advantages of performing operations tests in the real satellite

Simulation Tests	Real Satellite Ground Tests
• Only payload dummy models available • No real payload data generated	• Full payload operation including data takes with payload data downlink
• There are no perfect simulation models • Small deviations cause effects which distinguish the simulation from the real satellite from the point of view of an operator	• The operator experiences the real system • Problems which only present themselves on the real system caused by, e.g. timing deviations, side-effects, cross-influence of onboard equipment can only be encountered with the real system

Two separate operations tests were planned which cover different phases of the mission. One test which simulates the launch and early orbit phase (LEOP) and the second which simulates one week of nominal mission operations including payload activities. Before each test with the real satellite, a simulation run of the complete test run was performed using the STB. As described in section 3.4 it was possible to either control the STB from the mission control room or the real satellite within the cleanroom due to flexible TM/TC routing.

Certain aspects of operations could better be realized in full simulation with the STB rather than actual tests with the real satellite. Table 5.6.1 and Table 5.6.2 provide a comparison between the advantages and disadvantages of simulation and real-world ground tests. Clearly, both variants were required to cover all operational aspects of the mission.

This was the first point in the development process where the OBSW presented itself like it will be used in flight. As already mentioned, this milestone was also implied by a version change from Gemini to Apollo (see section 4.5.1). Every aspect of the full operations tests was thus also an implicit re-test of an OBSW functionality which has been already covered by a previous test. Up to this point commanding and telemetry functionalities of the OBSW have been added as needed and were only tested to some extent in previous system tests and in operational tests with the STB but never entirely with the real system. Especially the onboard storage of housekeeping telemetry and its management using service 15 (storage and retrieval) was heavily used in the full operations test but did not have much test time before. Also, PUS service 11 (onboard scheduling) was not used before except in exploratory simulation runs.

To simulate realistic orbit cycle conditions with the real satellite the following infrastructure was used to get lifelike telemetry data from the satellite in the integration cleanroom:

- Eclipse and sun phases were simulated by a solar array simulation power supply which enables scripted control over power output and solar array characteristics.
- Scripted activation and deactivation of RF-SCOE to simulate ground station contacts.
- RF signal quality decrease and simulation of interference by manipulation of RF-SCOE.
- Payload data downlink via dedicated downlink system and payload SCOE.

5.6.1 Launch and Early Orbit Phase Test

The LEOP test was a simulation of the exact sequence of events as expected to occur once the satellite gets separated from the launch vehicle. The goal of LEOP is to establish reliable communications with the satellite and to reach a safe state in which power supply is ensured.

It was decided to perform a dedicated LEOP test to better prepare the team for this critical task so that the operators are familiar with all procedures but also to increase test hours of the relevant OBSW functions. The test was conducted after the principles described in section 5.6. The detailed objectives of LEOP which are thus part of the test procedure were:

- To ensure the reduction of residual angular velocity after launcher separation. This is an implicit test of the detumbling control strategy of the attitude controller using only magnetic actuation which was already mentioned in section 5.3.4.
- To establish contact with the satellite, test commanding and check the overall state of the system including health flags, equipment temperatures and error events.
- To check that *Safe* mode attitude control is working reliably and orients the solar panels to the Sun.
- To deploy the solar panels. This is an implicit test of the OBSW deployment controller as described in section 5.1.3.4.
- To determine precise orbit data for pass prediction by activating the satellite's GPS receivers.

Table 5.6.3 lists the sequence of events during the LEOP test which comprises the steps to achieve the above-mentioned objectives and also the simulated passes in which these steps were performed.

Table 5.6.3: Sequence of event of LEOP test

Pass No.	Actions
1	• **Establish TM/TC communication** • Set onboard time • Read/reset DOM
2	• Verify finished detumbling
3	• Activate GPS0 receiver (adjust onboard time)
4	• Verify GPS • **Deployment decision**
5	• Solar panel deployment
6	• Command slow *Safe* mode rotation
7	• Verify rotation and speed up *Safe* mode rotation
8	• End of LEOP • Reset of config variables

During the LEOP test on STB, a few issues occurred which required to repeat the test several times. Among these issues were problems like:

1. Bad standard configuration of the used OBSW in which too much housekeeping information was produced.
2. Error in calculation of modified Julian date which caused the onboard time to jump 12h ahead when GPS receiver was activated
3. Usability issues of onboard TM housekeeping stores regarding erasure of housekeeping data memory, checking for completeness of received data and TM bandwidth allocation
4. Some limits of monitored data pool parameters had to be adjusted to reduce warning events
5. Minor tweaks to housekeeping service frequency of delivered TM packets and command definitions
6. Time synchronization within the ground segment of mission control did not work correctly

After these issues were resolved, the test could successfully be executed on the flight model.

5.6.2 Week-in-the-life Test

In the same way as the LEOP test, a one week-long operations test was conducted in a realistic setting with both, the real satellite and STB. In this case, the focus of the test was on the nominal operational aspects of the mission. Because the duration of this test was around one week, it was called the *week-in-the-life test*. The goal was to have the operations team operate the satellite over a significant time.

Most functional tests performed so far had a very limited scope and took only a short duration which is problematic because the confidence gained by short-term tests over the long-term performance is deceptive. It must not be neglected that a dedicated class of software and hardware problems only manifest after some time. Within one week of operation, most of these long-term effects should get exposed. Passing the week-in-the-life test brought a valuable increase in confidence in the system and the OBSW's stability. The objectives of the week-in-the-life test for the *Flying Laptop* were:

- to verify all nominal flight procedures
- to operate the satellite in all defined system modes

- to demonstrate management of onboard telemetry storage and ground telemetry processing
- to demonstrate payload operation and payload data handling

A full test procedure of the week-in-the-life test as performed on STB and real satellite is provided in Appendix D.

5.6.3 Evaluation of Operational Tests in the Context of the System Test Driven Approach

Just as for *Flying Laptop*, operational tests are regularly performed in traditional industry projects. As already described in 4.1, OSTs are used to operate the satellite in closed-loop together with the operational ground segment. This implies that already mission control, sometimes located in other facilities, is partaking in OSTs.

Figure 5.6.2 illustrates how the system test driven approach loses agility over the course of the system testing campaign and more and more resembles a classical testing approach. The benefits that the system test driven approach brings can be exploited most efficiently at the beginning, i.e., in the flatsat phase. The strengths of the method are best utilized early in the project.

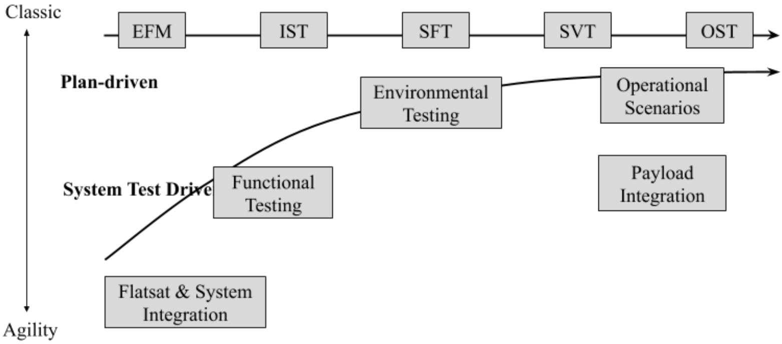


Figure 5.6.2: Agility of system test driven approach compared to a classic plan-driven method

- ★ In the system test driven approach, the classic OST is not performed with the satellite in closed-loop but divided into two different tests. One that is performed in the STB and one that is performed with the real satellite. This is done to cover all operational aspects and not only the ones that are representable with the different test environments.
- ★ This division has a positive effect on later operational tests performed with the STB. It helps to bring the results of the simulation closer to the real system since the STB is improved along the way.

5.7 Final Acceptance Test

Before shipping the satellite to the launch site, final acceptance tests were performed with the flight model. Approximately one month before shipping, a final OBSW release candidate was chosen. The final candidate represented the most advanced state of the OBSW to this day. With this release candidate, the final acceptance tests were performed which thoroughly scrutinized every aspect of the functionalities OBSW once again. The final acceptance tests focused on:

- Abbreviated functional tests (AFTs) (see section 6.3)
- Final sensor, heaters and actuator checks
- Retest of the solar panel deployment
- Functionality of all TM/TC ground services
- Commanding and operation of all system modes

Only minor bug-fixes were accepted for revisions of this release to not void the already performed tests through the made changes. If a significant problem with the release appeared during final acceptance tests, the release would have been discarded, the problems solved and in theory, the next release would undergo the same thorough testing from the start as the next release candidate for flight. Because of this strict process, sufficient time had to be allocated in advance in order to be able to react to problems. Fortunately, for *Flying Laptop* no such show-stoppers were discovered during this critical phase.

Table 5.7.1: Final configuration of OBC PROM for launch

PROM	primary	secondary
OBC Nom.	A.11.3-34	A.11.3-34
OBC Red.	G.53.1-104	A.11.3-34

Table 5.7.1 shows the final configuration of the persistent OBC memory (PROM). As a backup, in case there is a problem occurring in space right after launch, an older but well-tested release with reduced functionality was stored in the primary PROM of the redundant OBC. The OBC would first try to start OBSW image that is located in the primary PROM of the nominal OBC but if this fails it would after some time switch over to the redundant OBC and be able to boot a very settled and proven OBSW version. With this precaution, a one-time human error during final checkout, like e.g. selecting the wrong image, could also be prevented.

Although OBSW version G.53.1-104 (see Table 5.7.1) is missing a lot of features compared to A.11.3 it still had all the functionalities which were required to support LEOP, i.e. *Safe* mode attitude control, TM/TC services. It was the most mature and well-tested release of the Gemini OBSW phase and was thus chosen to be the ultimate fallback-version in case none of the other versions stored in PROM could be started.

6 Pure Software Tests in System Testbed

While the previous chapter describes the tests that were performed manually as functional system tests with the *Flying Laptop* flight model, this chapter describes the pure OBSW tests that were either performed once or as automated test procedures in the STB environment described in 3.4.

Section 6.1 evaluates the used equipment simulation models and compares the output of the models to real data gathered in orbit. The equipment models of the power supply subsystem, the sun sensor model, and the RW model were selected to give the reader an idea about the degree of resemblance that was achieved with the used equipment simulation models.

Section 6.2 describes how failure detection and failure handling was tested by pure software tests. Section 6.3 explains the automated test procedures which were created for *Flying Laptop* to test the overall health state of the satellite. Section 6.4 gives an introduction to PyOps, a satellite operations script module created in Python which was used for OBSW test automation. Section 6.5 describes automated tests which were introduced to test every individual OBSW release continuously and, finally, section 6.6 describes the experiences made with the application of exploratory testing.

6.1 Evaluation of Simulation Equipment Models

Once *Flying Laptop* was launched, the data gathered from telemetry was used to evaluate the precision of the used simulation models. This section provides comparisons of real mission data with data produced by simulation models in order to judge the required amount of realism for simulation models for potential future applications.

6.1.1 Evaluation of Power Supply Unit, Battery and Solar Panel Model

A stable power supply was one of the main goals of the system tests performed for *Flying Laptop*. The developed equipment models had to reflect key characteristics relevant to the OBSW PSS controller and failure detection with respect to voltage levels, the power consumption of components, and the functional behavior of the PCDU. The characteristics of the PSS which have been taken into account for the relevant equipment models to achieve this are given in Table 6.1.1.

Table 6.1.1: Characteristics of the PSS simulation models

Modeled Eq.	Characteristics to be modeled for Simulation
Solar Panels	• Power U/I curve • Orientation and moment of inertia change upon deployment • Degradation BOL/EOL
Batteries	• Characteristics of Li-Ion cells (charge/discharge curve) • Shepherd model parameters[20] • Capacity • Configuration (number of cells, series or parallel arrangement)
PCDU	• Voltage control of solar arrays • Battery charge control • Dynamics of the unregulated power bus • Equipment switches, fuses, relays

The interaction between incoming current generated by the solar panels and current going into or out of the three battery strings is a fundamental mechanism which is essential for modeling the power subsystem.

Figure 6.1.1 shows the solar panel voltage of each solar panel recorded over one orbit gathered from a simulation run compared to real-world data collected in orbit. In this example, the satellite was constantly pointing its solar panels to the Sun which means that if it was in sunlight, the batteries were charged with maximum efficiency. Figure 6.1.2 shows the battery string currents that were measured at the charging lines of each battery string. It can be seen that if solar panels were illuminated during sunlight phase of one orbit, which starts at approximately minute 32, they assumed a voltage according to their characteristic current-voltage curve. If the battery strings were not at 100% state of charge, the solar panels assumed a voltage close to the maximum power point (where the generated power is high). However, if the battery strings were fully charged the solar panel voltage jumped to a higher value at which less or even no power is generated due to the shape of the characteristic current-voltage curve in order not to overcharge the battery cells. At the same point, as visible in Figure 6.1.2 the battery string current dropped to zero. The PCDU is responsible for

[20] A shepherd model is a generic battery model which is commonly used to model the charge and discharge curves of Lithium-Ion batteries. The charge/discharge curve in Shepherd's battery model is based on Shepherd parameters which determine the shape of the curve [Shepherd 1965].

this mechanism and to adjust the solar panel voltage to regulate the generated power [Wertz et al. 2011], [Eickhoff 2016].

The simulation model of the PCDU uses a simple algorithm to adjust the voltage of each solar panel to approximate the behavior of the real PCDU because implementing an exact match would involve solving differential equations that express the behavior of the PCDU's electronic circuitry by the model. The deviation that is a result of this simplification leads to the higher voltage of the solar panels for the simulation model if a battery string is fully charged compared to the voltage shown in real-world data.

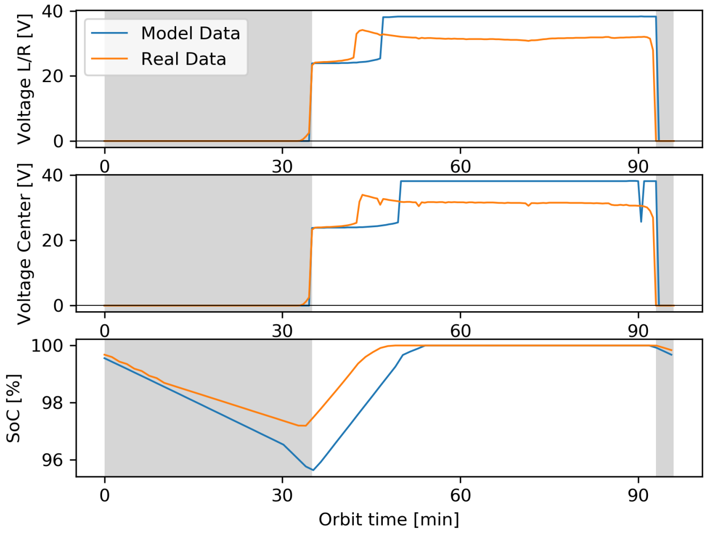


Figure 6.1.1: Comparison of solar panel voltages; top shows left/right solar panel strings (identical); middle graph shows center panel; bottom graph shows the corresponding overall battery state of charge; data is from simulation model and real data recorded in orbit during *Flying Laptop* mission

As a result of conservative assumptions about the power consumption of equipment in the equipment models, the overall power consumption in simulation is higher than the power consumption of the real satellite. For this reason, the negative battery string current during eclipse phases (negative means that the battery string is being discharged) is lower for real-world data. For the same reason, the real satellite battery strings are recharged more quickly after the satellite enters sunlight.

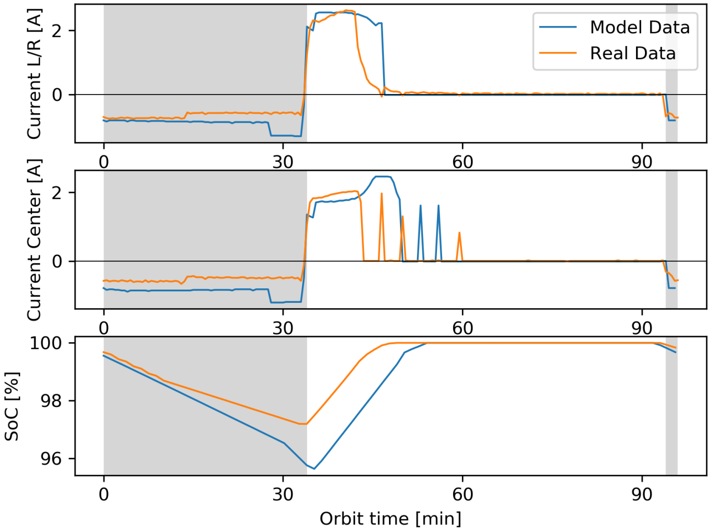


Figure 6.1.2: Comparison of battery string charge/discharge currents; top shows left/right battery strings (identical); middle graph shows center battery string (less capacity); bottom graph shows corresponding overall state of charge; data is from simulation model and real data recorded in orbit during *Flying Laptop* mission

More focus was deliberately laid on the discharge characteristics of the battery strings. A shepherd charge and discharge model was implemented to achieve results close to the real-world discharge characteristics which are also temperature-dependent. This was deliberately done because the OBSW uses a custom algorithm for battery state of charge determination. This state of charge determination algorithm has implications on automatic countermeasures performed by the OBSW (like deactivation of non-essential loads) when low state of charge levels are reached. For that reason, a good resemblance to real-world discharge curves was aimed for.

6.1.2 Evaluation of Sun Sensor Model

To orient itself to the Sun, *Flying Laptop* uses six out of eight available sun sensors (SuS) at a time distributed over the satellite body. These particular sun sensors are small solar cells which produce current when sunlight shines on them proportional to the Sun's elevation angle. By evaluating the current generated by six of the eight sun sensors, the OBSW is able to determine the direction of the Sun within the coordinates of the satellite's structure. In addition to finding the sun direction, the sun vector is used by the OBSW to derive the current angular rate from the sun vector's rate of change. By combining the angular rate derived from the sun vector with the one derived from the magnetic field (see section 5.4.2), the OBSW can further improve the estimation of the angular rate without using higher-level sensors like gyros or the star tracker which is especially required for Safe mode attitude control. The multipurpose use of the sun vector makes it an essential input for the OBSW's attitude control algorithms and special attention was paid to make sure that the sun vector determination is flawless.

In LEO, the reflected sunlight (earth albedo) is causing sun sensor cells to produce an additional current which is superimposed on the current generated from sunlight alone. Although the proportion of current generated from albedo radiation is smaller than the current produced by direct exposure to the Sun, it is still required to perform filtering on the sun sensor signal to avoid ambiguity. Because the Earth's albedo effect can reduce the accuracy of the sun direction, this might be problematic for the power budget of the satellite if the deviation is too high.

To better understand the sensitivity of the estimated sun direction to earth albedo and also to show that the attitude control software can reliably find the sun, a detailed sun sensor model was developed by [Falke 2008] which also models the influences of Earth's albedo. The already existing model was modified to better suit the conditions after the satellite's redesign, but the fundamental equations used to calculate the generated current stayed the same and are described in detail in [Falke 2008].

Figure 6.1.3 shows the arrangement and naming of sun sensors for *Flying Laptop*. Sun sensor C and H are unused in the configuration shown in Figure 6.1.3 in which the solar panel wings are deployed. Sensor E and G are facing in the same direction, but their field of view is not obstructed as it is the case for sun sensor C and H.

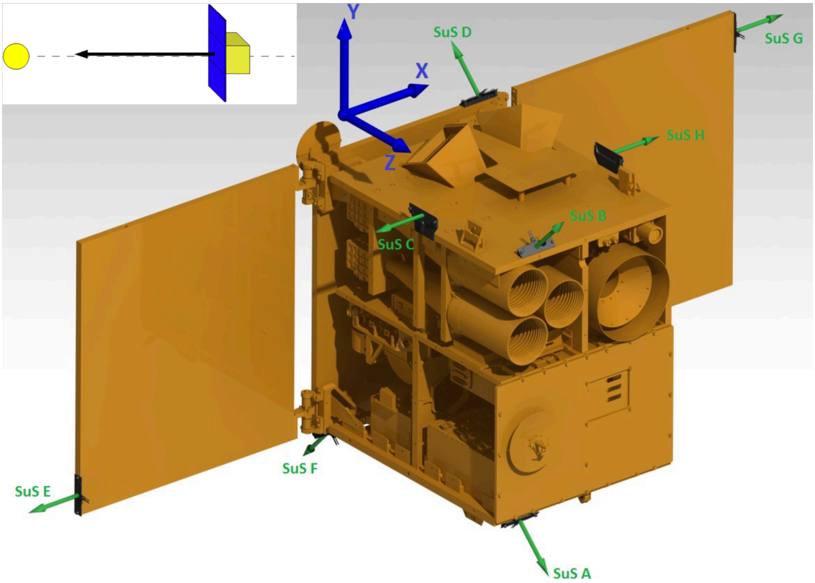


Figure 6.1.3: Orientation and identification of Flying Laptop sun sensors; *Idle* mode scenario, the satellite is oriented as shown in the upper left corner with SuS F and G generating current from sunlight

Figure 6.1.4 shows sun sensor currents recorded over one orbit generated by the simulation model and the resulting sun vector magnitude calculated by the attitude controller. For comparison, Figure 6.1.4 also shows the equivalent in-orbit data gathered during the commissioning phase. In this scenario, the satellite is actively orienting the solar panels to the Sun and is maintaining stable a attitude over the whole sunlight period which results in a constant value of -1 for the z-component of the sun vector, this value signifies a good orientation of the solar panels towards the sun.

In this scenario, sun sensor F and sensor D are facing in sun direction with a 45° angle. Sensor G and sensor E are facing perpendicular to the sun direction. Sensor A and sensor B are facing in opposite direction away from the sun and are not shown in Figure 6.1.4.

As described by [Falke 2008], an increased effort was put into making the output of the sun sensor model realistically sensitive to influences of temperature. However, as real-world data shows (see Figure 6.1.4), the influence of temperature on the sun sensor cells is far less drastic than anticipated.

While the cause for this overestimation of temperature influence on sun sensor current is subject to speculation, one possible reason could be that the sun sensor cells themselves heat up very fast when exposed to sunlight because of their very low thermal capacity and weak bonding to the satellite's main structure so that the thermal conditions very quickly become static. The static temperatures, in that case, would be the settling temperatures in thermal equilibrium.

As in-orbit data shows, at a maximum temperature of roughly 50°C in *Idle* mode, the sensors which are directly exposed to sunlight stay within moderate temperature ranges. Additionally, the temperatures displayed in Figure 6.1.4 are measured from thermal sensor elements which are likely to not correctly represent the actual cell temperature because they are located beside the sun sensors and are not directly coupled with the surface of the individual sun sensor cells. For this reason, it is more likely that the actual surface temperature of the sun sensor cell is more accurately represented by the temperature of the sun sensor model which is also shown in Figure 6.1.4.

The calculation of the sun sensor cell temperature by the sun sensor model is based on the Stefan-Boltzmann law with the assumption of thermal equilibrium in every simulation step. Thus, there is no asymptotic property of the temperature curves visible in Figure 6.1.4 when compared to the real in-orbit temperatures. Each sun sensor element is considered a grey body that absorbs incoming sun- and albedo-radiation (described in [Falke 2008]). This means that the sun sensor itself has no thermal capacity.

Also clearly visible both the real in-orbit data and the simulation model data is the effect of earth albedo on current and temperature of sun sensors G and E on the outer edges of the solar panels.

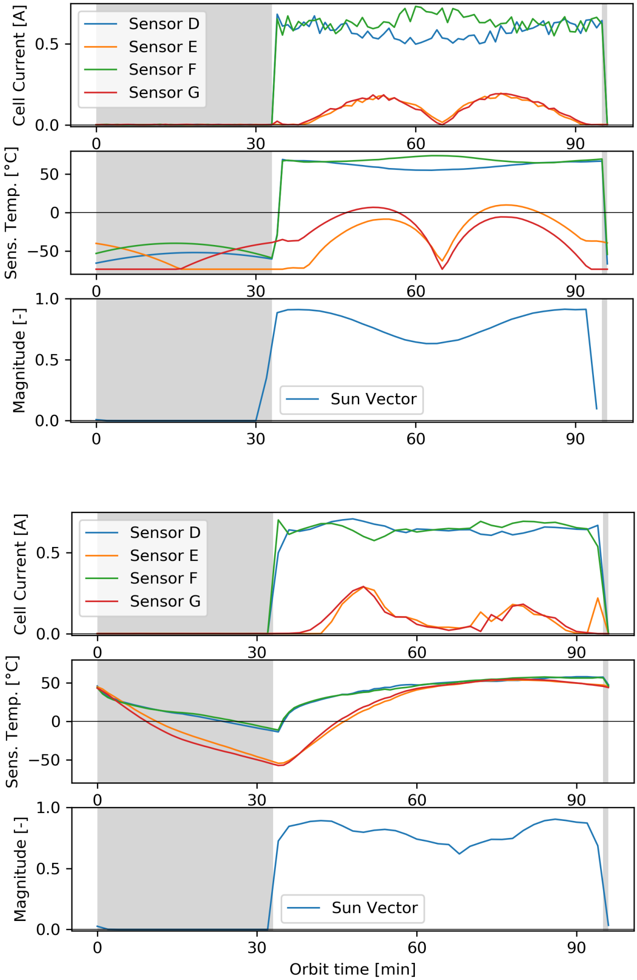


Figure 6.1.4: Sun sensor currents, temperatures and resulting sun vector from simulation model (top group of graphs) and real mission data (bottom group of graphs); grey area represents eclipse phase; two discernible rises in sensor current E and G visible in sim. data and real orbit data are a results of earth albedo

6.1.3 Evaluation of Reaction Wheel Model

The RW simulation model received special attention during functional system tests. The RWs are used as actuators for high precision target pointing attitude control. RWs are electric motors which spin an internal flywheel. The torque produced by the electric motor is causing the satellite to rotate as a counter-reaction in a drag-free environment. The internal dynamics of the RW have a significant impact on the precision of the attitude control system and the power consumed by each RW [Wertz et al. 2011].

Because the OBSW monitors the health of the RWs, additional characteristics have to be modeled correctly to present plausible data and not trigger any unwanted reactions.

For the RW simulation model the following characteristics were considered:

- Wheel speed and generated torque
- Current consumption in different modes of operation and at different set points
- Characteristic drag as a result of friction which also determines power consumption and slow-down of unpowered wheel
- Capacity of storable momentum

To produce the correct amount of torque, the RW model simulates the internal control loop of the real RW. The internal control loop always strives to keep the wheel speed at the commanded value. The consumed power is approximated by two separate components of the wheel current. One part is proportional to the actual wheel speed (required to counteract drag) and the other part is required if the wheel speed is changed to produce torque. To approximate the actual currents and drag coefficients, tests with the real RWs have been performed.

The recorded data has been used to extract the correct model parameters. Figure 6.1.5 shows the typical curve of individual motor currents and wheel speed recorded during this test with the real RW units. In the beginning, the RWs were commanded from no rotation to 350 rpm wheel speed which resulted in a short peak in motor current. After the wheel speed had settled, the internal control loop steadily kept the commanded speed. After approximately 170 seconds the wheel motor was deactivated and each wheel spun down. The spin-down curve characterizes each individual RW and is a result of the remaining drag caused by friction in the bearing of the RW. By comparing this curve to later results of repetitions of this test RW degradation can be analyzed later on e.g. during the operational mission phase.

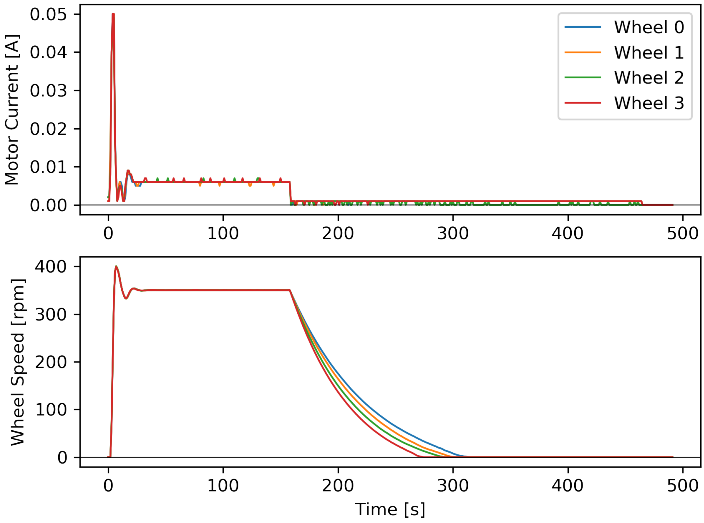


Figure 6.1.5: RW spin down curves of individual RWs recorded in ground tests by real RW units

The same test has been performed for the simulation model. Figure 6.1.6 shows the resulting data. While the overall curve looks similar compared to Figure 6.1.5 it is worth noting that the initial peak in motor current is much higher for the simulation model (almost twice as high in peak current).

Although this could be fine-tuned to match the real RW exactly, the maximum current of the simulation model is within specifications and at the same time presents a conservative assumption. The rate at which the wheel speed decreases after motor power has been cut off roughly matches the one observed from real RW data but takes more time to finally reach zero because friction effects around zero rpm are not modeled in detail. To create diversity among all simulation models, the individual friction parameters have been adjusted later on.

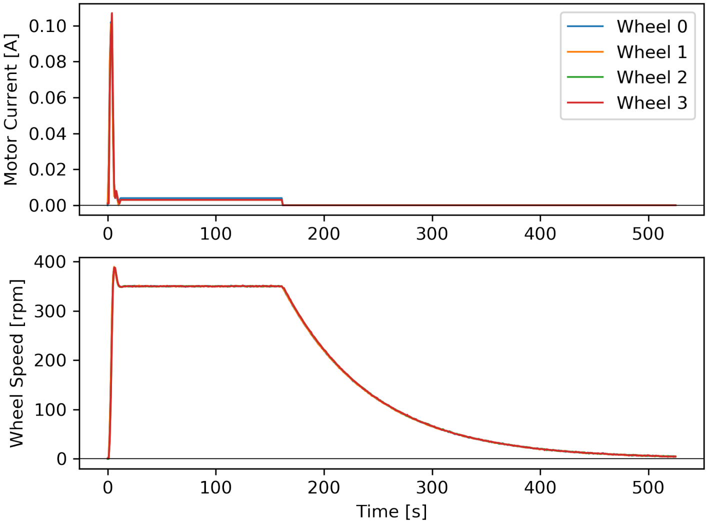


Figure 6.1.6: RW spin down curves generated in software tests by simulation models

6.2 Failure Detection, Isolation and Recovery

In [Witt 2016], an approach for testing failure detection isolation and recovery (FDIR) mechanisms of the OBSW is outlined. This chapter will elaborate and expand this approach and at the same time describe the more general basis for testing failure handling software. A detailed description of each fault that is handled by the OBSW as well as the failure handling framework can be found in [Witt 2016].

As [Witt 2016] states, testing failure handling in satellite systems is not done very easily. Assuming, that failure handling routines are already tested individually in isolated unit tests, the system still has to show that it can

1. detect any fault reliably
2. isolate the component which caused the error and
3. trigger the correct actions to recover and reach a predefined safe state.

Especially the isolation of the faulty component can be difficult. The picture that presents itself through a failure effect (the symptoms) can often be misleading. Failure scenarios and rules on how to respond have been analyzed in [Witt 2016]. Once detected, the OBSW uses these built-in rules to react to these failure scenarios. The last resort, in any case, is to go to *Safe* mode. Beyond *Safe* mode, there are no further means of the OBSW to handle problems. If the problem is located in the OBC itself, the PCDU can perform a redundancy switch of the OBC triggered by the internal PCDU watchdog.

Figure 6.2.1 shows each mode change which is performed by the OBSW as the result of a detected error. As a general rule of the design, the *Flying Laptop* FDIR inclines towards a fail-safe strategy. Through the concept of OBSW assembly components, however, an operational state is maintained as long as possible if enough redundant devices are healthy.

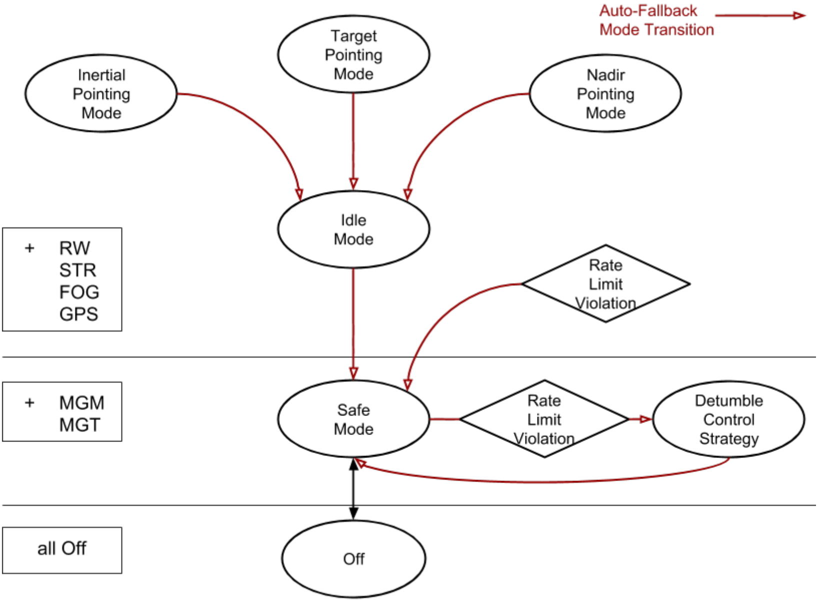


Figure 6.2.1: Automatic mode fallback of attitude control subsystem [Bätz 2018]

As visible from Figure 6.2.1, failures in higher modes like target pointing mode, etc. lead to a fallback into *Idle* mode first. If another fault occurs in *Idle* mode either because the same problem is also present or another new problem arises, the last resort is to fall back to *Safe* mode. Rate limit violations occur if the satellite's rotational rate exceeds the limit. A rate limit violation always triggers a fallback to *Safe* mode immediately. Once in *Safe* mode there is no further possible fallback. An unrecoverable fault in *Safe* mode would lead to mission loss if power supply cannot be maintained.

The chain of reactions caused by an error cannot be tested by isolated unit tests since it involves interactions between different OBSW components. Thus, testing FDIR functions is always a system-level discipline.

Failure handling of the OBSW follows a proven principle described in [Eickhoff 2012], to always handle errors on the lowest possible level. Often this is also described as handling the fault locally [Witt 2016]. The hierarchical *platform abstraction model* of the OBSW (see section 3.3.1 and Figure 3.3.3) allows escalating a failure incident to the next higher-level object (the parent) if it cannot be handled locally. The highest possible level of escalation is the system level.

To summarize, from a local perspective of an OBSW component (see Figure 3.3.3) there are two types of failure events:

- Failure events which can be handled locally, that means there are strategies for this specific failure event programmed into the local component object to handle such failures directly, e.g. a device handler resetting its device when there are data checksum errors or an assembly switching to a redundant unit.
- Failure events where no such strategy exists or strategies to mitigate the problem locally have been exhausted without success which means that the local object cannot keep its current mode of operation. In this case, such a failure event has to be passed on to the next higher-level.

It was reasonable to split failure detection from failure handling tests for the OBSW. In the OBSW design, failure detection takes place in lower level device handling and monitoring while failure handling is an effort spanning over all system levels. Because of this split in the OBSW architecture, it was practical to test failure detection apart from failure handling. This, however, requires the ability of the OBSW to react to error events injected from an external source without involving low-level error detection as described by [Freeman et al. 2010] (externalizing event sources for testing object-oriented software). This method is further described in section 6.2.1.

One further aspect to consider is that failing equipment cannot be satisfactorily realized with a real satellite system because it would require damaging or destroying equipment. To simulate e.g. a faulty sensor, the sensor unit would have to be at least disconnected from the satellite's harness which is oftentimes not possible[21].

For this reason, simulation-based tests were the only feasible method to test OBSW FDIR. In model-based simulation environments like the STB, equipment models have to support simulating error states. By introducing more complex error states into simulation models testing complex failure scenarios in which devices behave unreliably or deliver false or noisy data is also possible. Testing the lower level failure detection part of the OBSW is described in section 6.2.2.

6.2.1 Using Event Injection to Test Failure Handling

Decoupling of detecting and reacting to failures is part of the OBSW design [Witt 2016]. The ECSS PUS [ECSS Committee 2003] defines that the minimum capabilities of the event reporting service are to report onboard events from progress reports up to anomaly reports of high severity (subtype 1 to 4). Additionally, it defines the optional capability to enable/ disable event reporting (subtype 5 and 6).

For the OBSW, an additional custom subtype was introduced which allows injecting onboard events into the OBSW from ground over the TM/TC link [Eickhoff 2016]. The custom event injection sub-service (5, 128) can be used to inject failure events to the central event manager as if they were generated onboard (see section 3.3.4).

[21] Connectors are exposed to risk of damage with each plugging cycle. It is recommended to limit plug cycles for wired connections inside a satellite during assembly and testing because repeatedly plugging connectors can harm electrical contacts. Once installed these connectors are often hard to reach because of the high integration density and also connectors are often glued to prevent loosening.

This concept of event injection is a useful tool for testing the OBSW because any generated event is first sent to the event manager. The event manager forwards all events to the registered event listeners. Through the event manager, the OBSW components that react to certain events, e.g. if they perform failure handling actions, are connected to a central distribution. This way, any OBSW component can listen to onboard events and implement logic to react to them. Unlike traditional flight software architectures, the OBSW has no central FDIR application. The FDIR functionalities are distributed over small failure handling functions within components like e.g. assemblies or mode management.

Figure 6.2.2 illustrates the concept of event injection within the OBSW. Events can either be generated by OBSW components like the system would normally operate during the mission or events can be injected over the command link from ground. A TC source packet to inject a specific event holds the relevant information of the event i.e., event-id, object-id of the OBSW component that generated the event, severity level and two optional event-parameters (cf. Table 4.7.1).

The approach to test system reactions that way is valid because the only interaction between OBSW components for passing on failure information is by event generation. As soon as other interactions play a role in the system's reaction this test approach is not valid anymore. An example of such interaction is, e.g. if the FDIR reaction also depends on flags in the data-pool. For this reason, the failure handling mechanisms of the OBSW have been designed so that they solely rely on event exchange to pass on failure events between OBSW components.

Even more complex failure scenarios can be reproduced by using the event injection service, e.g. if more than one OBSW component is affected by a fault. This can be simulated by injecting the corresponding event pattern (a certain combination of events that is typical for a chain of errors). Such event patterns can be recorded during failure detection tests using low-level error injection (see also section 3.4.2).

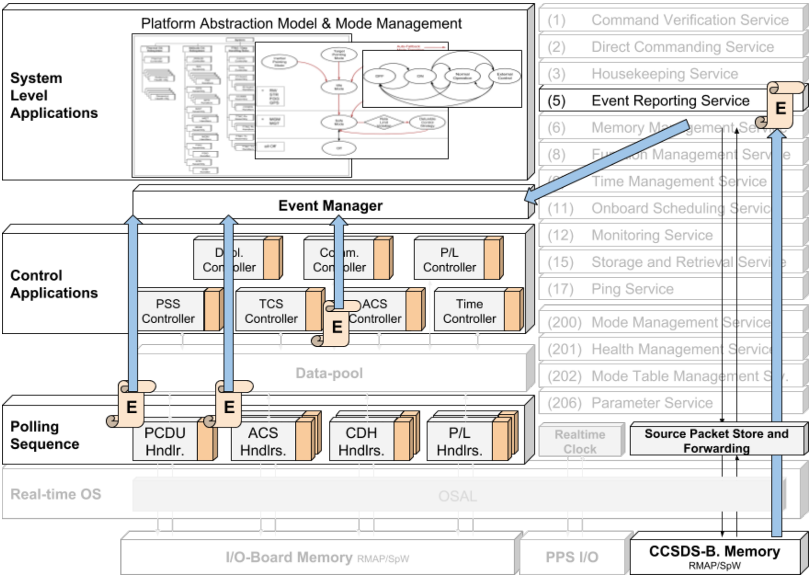

Platform Abstraction Model & Mode Management
System Level Applications
Event Manager
Control Applications
Dpl. Controller
Comm. Controller
P/L Controller
PSS Controller
TCS Controller
ACS Controller
Time Controller
E
Data-pool
Polling Sequence
PCDU Hndlr.
ACS Hndlrs.
CDH Hndlrs.
P/L Hndlrs.
Real-time OS
OSAL
I/O-Board Memory RMAP/SpW
PPS I/O
CCSDS-B. Memory RMAP/SpW
(1) Command Verification Service
(2) Direct Commanding Service
(3) Housekeeping Service
(5) Event Reporting Service
(6) Memory Management Service
(8) Function Management Service
(9) Time Management Service
(11) Onboard Scheduling Service
(12) Monitoring Service
(15) Storage and Retrieval Service
(17) Ping Service
(200) Mode Management Service
(201) Health Management Service
(202) Mode Table Management Srv.
(206) Parameter Service
Realtime Clock
Source Packet Store and Forwarding

Figure 6.2.2: Using the event injection service (5, 128) to trigger FDIR reactions; Events can either be generated by OBSW components or injected by sending event injection commands

6.2.2 Failure Detection

While section 6.2.1 explains how failure handling was tested for the OBSW, this section is about how these failures are detected and how this detection is tested. As shown in Figure 6.2.2, low-level OBSW components generate error events when errors are detected. Device handlers, e.g. notify FDIR if devices are unresponsive, deliver bad data or report problems by event generation. Controllers monitor if specific parameters exceed their limits or their gradient is too high, this concept was already described in section 5.3 in the context of monitors for temperature sensors.

To test if these errors are detected correctly within the device handlers, errors were injected into the equipment simulation models of the STB. The means of the STB to inject different error types are listed in Table 6.2.1. In most cases, the error injection user interface, described in section 3.4.2, can be used to control errors during simulation runtime.

Table 6.2.1: Different error types and how they are simulated by built-in STB functionalities and simulation models; TX = transmitting data line; RX = receiving data line

Error Type	Detected by OBSW Component	Injection Model	Simulation Method
Temporary device communication error	Device Handler	I/O Board	An uncommandable device can be simulated by disabling its TX (OBC to device) communication line. Command to I/O board model.
Malformed Device Reply	Device Handler	-	Malformed device replies can only be simulated by modifying the simulation model source code and change the formatting (length or structure) of the device reply. This cannot be activated during simulation runtime and requires recompiling the model.
Temporarily Missing Device Replies	Device Handler	I/O Board	Missing device replies can be simulated by disabling the device's RX (device to OBC) communication line. Command to I/O board model.
Device permanently damaged (no power consumed)	Device Handler	PCDU	A permanently damaged device can be simulated by disconnecting the power line in the PCDU model. In this case, there will be no device current measured by the PCDU.
Device permanently damaged (power consumed)	Device Handler	PCDU	A permanently damaged device which still consumes power (i.e. a device current will be measured by the PCDU) can be simulated by disabling both RX and TX communication lines. The device will be unresponsive and not communicating but it will be consuming power.
Sensor value stuck	Controller/Monitor (voting)	Error Injection	A stuck sensor value can be simulated by commands to the error injection model.
Biased sensor value	Controller/Monitor (voting)	Error Injection	By using the error injection model a constant offset can be applied to any sensor value during simulation runtime.
Noisy sensor value	Controller/Monitor (limit checking)	Error Injection	By using the error injection model, the noise level (mean deviation) of any sensor value can be adjusted during simulation runtime.

Because of the special OBSW architecture, it was possible to follow a scheme of separation of concerns for testing failure handling of the OBSW. In this approach, testing of failure detection and handling boils down to pure software tests on the STB. The flight model was not required to perform these tests which also relieved the occupation of the satellite itself.

6.3 Abbreviated Functional Tests

Abbreviated functional tests (AFTs) are test procedures that serve the purpose of quickly checking the overall health of a satellite during ground testing. These procedures automatically check if the satellite is in good health after, e.g. transportation or stressful mechanical tests like vibration or thermal vacuum tests. Table 6.3.1 is a simplified stepwise description of the procedure performed for the MGT AFT.

Table 6.3.1: Simplified sequence of test steps for the MGT AFT module

Description	**OBSW Component State**		
	I/O board Handler	MGT 0 Handler *(nominal coil group)*	MGT 1 Handler *(redundant coil group)*
Stage 1 - Initialization			
Set system in default configuration	nominal	OFF	OFF
Stage 2 - Check of nominal coil group and integrity of data connection to nominal I/O board			
Switch nominal coil handler on	nominal	ON	OFF
check all coil currents from TM *(between 0.046A and 0.058A)*	nominal	ON	OFF
Switch nominal coil handler off	nominal	OFF	OFF
Stage 3 - Check of redundant coil group and integrity of data connection to nominal I/O board			
Switch redundant coil handler on	nominal	OFF	ON
check all coil currents from TM *(between 0.046A and 0.058A)*	nominal	OFF	ON
Switch redundant coil handler off	nominal	OFF	OFF
Stage 4 - Simplified test with redundant I/O board			
Switch from nom. to red. I/O board	redundant	OFF	OFF
Switch nominal coil handler on	redundant	ON	OFF
Verify device communication	redundant	ON	OFF
Switch nominal coil handler off	redundant	OFF	OFF
Switch redundant coil handler on	redundant	OFF	ON
Verify device communication	redundant	OFF	ON
Switch redundant coil handler off	redundant	OFF	OFF
Stage 5 - Clean-Up			
Switch from red. to nom. I/O board	nominal	OFF	OFF
Reset default configuration	nominal	OFF	OFF

An AFT demonstrates that each piece of equipment is functioning and that each crosslink (data- or power-line) of the internal satellite harness is intact. As opposed to AFTs, [ECSS Committee 2012] defines full functional tests (FFTs) as a comprehensive series of tests that demonstrates the integrity of all functions of the test. AFTs, in contrast, are defined as a subset of FFTs to verify the integrity of the main functions of the system with a sufficiently high degree of confidence in a relatively short time.

For *Flying Laptop,* FFTs were performed manually in the form of manual functional system tests described in chapter 5 but AFTs, on the other hand, were realized in MOIS as automatic test scripts. To structure the very large amount of single MOIS procedures, the AFT was split into different modules. AFT modules were defined for core data handling, ACS, TCS and payload subsystems. The core AFT module was usually executed first. It consists of functional tests of the internal OBC boards, TT&C equipment, and the PCDU.

Because an AFT procedure checks all harness lines of a specific piece of equipment, the OBC has to switch between both redundant I/O boards during every AFT module since any equipment connected to the OBC has two communication lines, one for each I/O board like visible in Figure 3.4.2. During the internal check of the OBC, the redundant processor boards have to be switched which means that the OBSW will reboot several times during the core AFT module to test every possible combination of active OBC boards. The *None* mode which is defined as a system mode (see Appendix C) where all subsystems can be operated from ground by the tester without the interference of the OBSW's mode logic was required to freely change the mode of any component during AFTs.

The MGT equipment for which an AFT procedure is explained in a simplified way in Table 6.3.1 is a redundant unit. This means that the OBSW has two MGT handlers, one for each unit. Additionally, each MGT unit can actuate the two separate coil groups comprising one electromagnetic coil in each spatial direction. In order to check all MGT harness lines, the procedure has to activate each combination of redundant MGT unit and redundant I/O board. To check if the individual coils are healthy, the AFT procedure checks if there is a current on each coil, once the coil has been activated.

Each row in Table 6.3.1 represents an execution step of the MOIS procedure. E.g. switching the MGT on or off is achieved by sending the appropriate command over the TC link.

- *Stage 1* is the initialization stage. Here, it is checked if the system is in the default configuration for AFTs which means the nominal OBC I/O board is active and both MGT units are off.
- *Stages 2* and *3* check the nominal and redundant coil groups using the nominal I/O board. If the device is unresponsive, the command execution verification fails which would end the test and mark it as failed.
- *Stage 4* performs a reduced check of Stages 2 and 3 with the redundant I/O board since the coil groups have already been checked in *stages 2* and *3*.

Now each combination of redundant equipment and I/O board has been verified and the final stage of the AFT module which is to bring the system back to the initial state is performed.

A similar procedure is defined for all onboard equipment as listed in Appendix E. The single modules can be executed and in arbitrary order, because it is ensured that the system default state is reached after the completion of every single procedure.

6.4 PyOps

As already mentioned in section 2.2.4.2, the available solutions for scripted spacecraft operation automation like MOIS and other procedure execution agents are not particularly suited for integration into a Linux based continuous testing environment. A solution which plays well with the established build system and the overall Linux infrastructure at the University of Stuttgart was required to enable automatic release tests run by the build server like, e.g. continuous or scheduled daily/nightly builds.

Since Python is widely spread among the scientific community and due to the increased popularity of the programming language it was decided to create a Python module which can act as a test agent and which offers a simple yet powerful set of functionalities that follows the Python design philosophy which emphasizes code readability. The Python module which was created as part of a student thesis [Müller 2018] is called PyOps.

One requirement of PyOps was to interface with the existing ground infrastructure at the University of Stuttgart which relies on SCOS-2000. SCOS-2000 uses the Common Object Request Broker Architecture (CORBA) for external interfacing applications like, e.g.

MOIS. For this reason, the omniORBpy[22] library was used as an interface to the PyOps module to SCOS-2000's external interfaces. This enables to remotely control SCOS-2000 and use it to send commands and read out telemetry.

The Python scripts created with PyOps, however, are not specifically depending on SCOS-2000. This means that if in the future, a different MCS is introduced, the procedures can still be used with the new MCS only the underlying connection layer is exchanged.

On the operational level, PyOps offers the following functions to the script programmer which are part of the standard workflow of a human operator:

- Sending commands
- Verifying command execution
- Waiting/checking for specific onboard events in TM
- Waiting/checking for specific TM source packets
- Checking TM parameter values and validity

With these basic functions combined with the native abilities of Python itself, it is possible to programmatically script every interaction with the system under test that a human operator is also able to perform. An example of a possible test script that checks the orientation of the solar panels in *Idle* mode is outlined in Figure 6.4.1.

In principle, it is possible to implement this test procedure in MOIS or using any other spacecraft operations automation method. The benefits that the programming language Python brings, however, are obvious: The script is short and expressive, it can be version controlled using Git and it can run with the automatic build system in a Linux environment. As an additional benefit, since the Python programming language is common among the scientific community, the programmers in a university context do not have to adopt a new language to implement a test script.

Since Python offers a lot of powerful libraries, PyOps scripts can be enriched with lots of additional functionality by using additional Python modules. Possible use cases could include test result management, generation of test statistics or interactions with additional ground software like e.g. mission planning or mission databases through various APIs.

[22] omniORB is an open source CORBA implementation. For more information see: omniorb.sourceforge.net

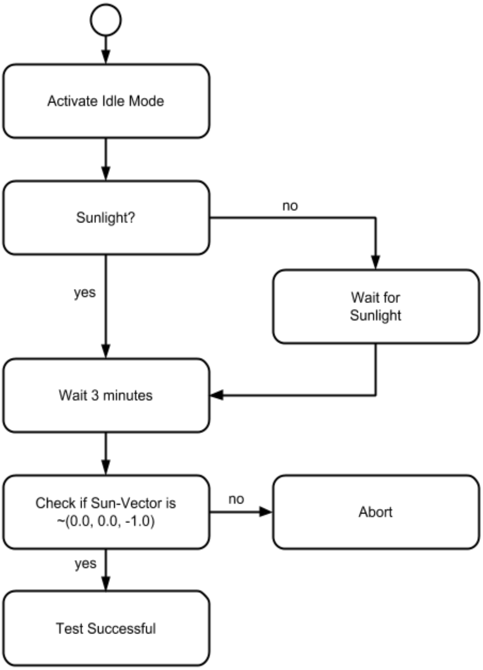


Figure 6.4.1: Example Test Script implemented in PyOps; compare Listing 6.4.1

The pseudo Python source code of Listing 6.4.1 is the implementation of the test script outlined in Figure 6.4.1 using PyOps. Note that to use PyOps, the user has to import the *Operator* object from the PyOps module.

Upon connection, the PyOps module creates a CORBA connection to SCOS-2000 and loads the mission-specific database (MIB). If the user requests, e.g. a command that is not contained in the database, PyOps will issue a corresponding error during the execution of the script. In this example, the only mission-specific information contained in the script are the command and TM parameter names and definitions.

Listing 6.4.1: Example PyOps test script which commands system *Idle* mode and then checks sun vector; pseudo-code example, following a rough "Pythonian" syntax

```
from pyops import Operator
with Operator() as op:
  op.connect()
  cmd = op.createCommand('YMC22001').setCommandParameter(mode='Idle')
  op.injectCommand(cmd)
  sunlight = op.registerTMParameter('AYTSUV00')
  while sunlight is False:
    time.sleep(180)
  sunvector = op.registerTMParameter('AYTSVC')
  if sunvector[0] > 0.1: fail()
  if sunvector[1] > 0.1: fail()
  if sunvector[2] < 0.9 or sunvector > 1.1: fail()
  success()
```

6.5 Automation of Integration Tests and Build Process

For testing each release automatically, a CI process was implemented. The process is based on the capabilities of the Jenkins automation server[23]. Jenkins builds (compiles) the OBSW binaries and then automatically starts simulation sessions on the STB to run pre-defined test scripts. A list of test scripts which were implemented for the OBSW is given in Table 6.5.1.

Because of the shortcomings of MOIS to remotely execute test procedures by Jenkins, PyOps (see section 6.4) is used to prepare these test procedure scripts. The execution of Python scripts can easily be integrated into the Jenkins workflow as an additional execution step after compilation of the OBSW executable.

Figure 6.5.1 exemplifies the basic steps that are executed automatically if a new release is tagged in the code repository. This is what happens under the hood in the *build server* step already mentioned in Figure 4.5.1. The Jenkins build server picks up this tagged commit runs all defined unit tests, cppcheck and compiles all required build artifacts. This step, of

[23] Jenkins is an open source, web-based build system. It automates parts of software integration, test and delivery tasks. For more information see: jenkins.io

course, can already fail. Given the compilation process was successful, Jenkins deploys the new OBSW binary to the OBC of the STB.

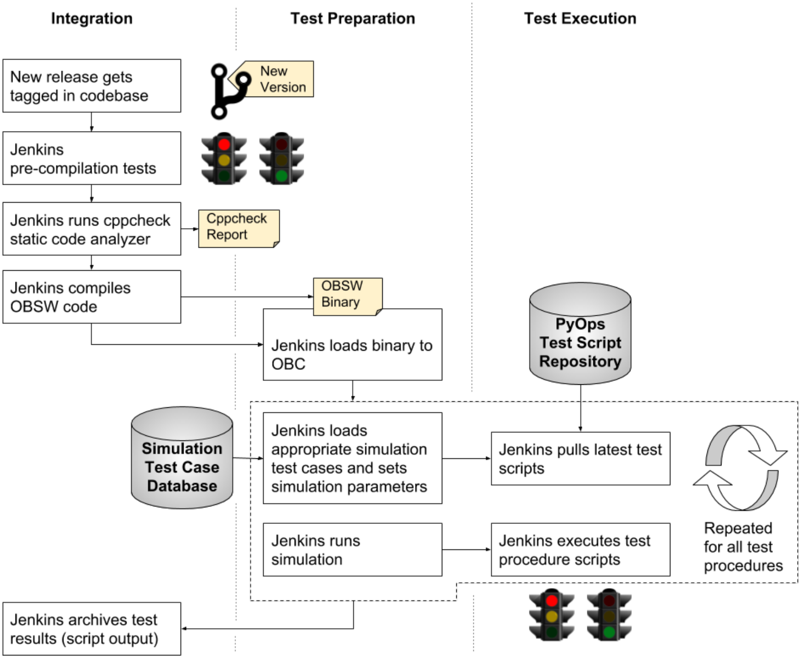

Integration
Test Preparation
Test Execution
New release gets tagged in codebase
New Version
Jenkins pre-compilation tests
Jenkins runs cppcheck static code analyzer
Cppcheck Report
Jenkins compiles OBSW code
OBSW Binary
Jenkins loads binary to OBC
PyOps Test Script Repository
Simulation Test Case Database
Jenkins loads appropriate simulation test cases and sets simulation parameters
Jenkins pulls latest test scripts
Repeated for all test procedures
Jenkins runs simulation
Jenkins executes test procedure scripts
Jenkins archives test results (script output)

Figure 6.5.1: Automated CI and test workflow which is executed by Jenkins build server for each OBSW release

The next step is for Jenkins to pull the latest test procedure scripts from the repository. Each test procedure can have different test case requirements for the simulation session which it interacts with. E.g., a test for the attitude controller might require a certain initial attitude or a test of redundancy management might require certain models to be faulty. Some of these parameters can be set through the test procedure scripts themselves through so-called test fixtures, but others require changing the simulation test cases and thus also require a separate simulation session.

Jenkins runs each available test script consecutively. Jenkins logs the output of each test script and stores the log with the corresponding build artifacts. Jenkins will also set the overall outcome of the build to either successful, if all test steps succeed, or failed if at least one test step fails.

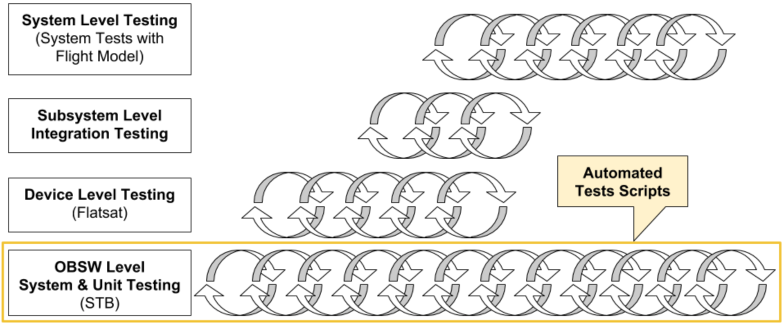

System Level Testing
(System Tests with Flight Model)
Subsystem Level Integration Testing
Device Level Testing
(Flatsat)
Automated Tests Scripts
OBSW Level System & Unit Testing
(STB)

Figure 6.5.2: Context of automated test scripts in the OBSW development process

As shown in Figure 6.5.2, the automated tests are all performed on the STB. System-level functional tests that involve the physical flight hardware, like e.g. flatsat tests and assembled flight model are not in the scope of this method. Test scripts have been realized for the OBSW throughout the development process including tests of mode transitions as described in 5.2 for system and low-level OBSW components, ACS functionality, TM/TC services.

Table 6.5.1: Automated tests performed with each OBSW release using the described infrastructure; full coverage means complete set of test cases; smoke tests do not fulfill the full coverage criteria

Test Name	Coverage
OBSW parameter check	full
Device handler mode transitions	full
Assembly health management	smoke test
System mode transitions	full
ACS function tests (*Safe* mode, *Idle* mode)	smoke test
PCDU direct commanding	smoke test
FDIR reactions to error events	full

6.6 Application of Exploratory Testing

As described in 2.3.6, exploratory testing is not an automated test method like the methods described in 6.3 and 6.5, but rather it was part of the continuous testing strategy for the OBSW. With the availability of the STB, it was possible to conduct exploratory bug hunts conveniently because the complete mission control system was available to be used by the testers. While exploratory testing did not have the highest priority among all OBSW test activities, the time invested in exploratory testing sessions has proven to be valuable.

One example of a successful application of the method for OBSW testing was a bug discovered by pure chance during an exploratory test session with an Apollo OBSW.

In a nutshell, the bug appeared only if a lot of large time-tagged commands were loaded into the onboard command schedule. The onboard command scheduling service maintains a list of commands in memory which will be executed at a specific point in time. This enables uploading schedules of activities that the satellite performs while no live link is available. Some operations like e.g. payload activities usually require a lot of time-tagged commands.

In this particular case, the source code which manages large commands had a problem once the maximum amount of storable commands was exceeded, but regularly sized commands worked fine. If the amount of storable large commands was exceeded, the OBSW crashed because of a memory overflow.

During exploratory testing sessions with the STB, this problem was identified by inexperienced students. Such sessions were conducted on a weekly basis and the students were given rough descriptions of areas of mission operation in which they should come up with interesting test cases through their own creativity and knowledge learned in lectures. Since the problem only occurred if enough commands of a specific size were scheduled, it was not detected during regular tests where mostly normal size commands were used.

This example shows that exploratory testing as an independent testing technique is worthwhile and should be carried out if the project schedule allows for it. Especially if the developed OBSW is not reused from different missions, this testing activity helps to increase the amount of real usage in contrast to developer tests.

7 Discussion and Outlook

In this work, an agile software development method for spacecraft flight software has been presented which on the one hand benefits from model-based system simulation and on the other hand is tightly connected to the spacecraft's system testing activities. The approach presented here benefits from a constant exchange between developers and testers and agile planning. The approach aims to both

- decrease software development time by overlapping software development and testing and at the same time to
- increase software quality through the various presented techniques in combination with the facilitation of a culture of direct exchange between people and direct feedback rather than relying on long and ponderous processes full of formalities.

In contrast to traditional spacecraft software development methods where functional software testing is one separate phase in the overall functional verification campaign, this approach incorporates software testing as a fundamental part in all phases.

To achieve this, the following work was contributed to the *Flying Laptop* project in which the approach was applied and field-tested as part of this thesis:

- Establishment of the iterative and incremental agile development approach for flight software development and functional system testing. The approach aimed to overcome problems encountered through the OBC technology change and spread in technology readiness (see section 3.1).
- Integration of the required agile planning into the already existing project management structure (see section 4.1).
- Assessment of simulation model readiness and calibration of simulation models through data collected from ground-based functional system tests (see section 3.4.1).
- Creation of an error injection framework for the model-based real-time simulation of *Flying Laptop* (see section 3.4.2).
- Creation of software tools for better evaluation and archiving of telemetry data gathered through tests (see section 4.7).

- Planning and execution of the flatsat test campaign to support equipment device handler development and demonstrate overall system compatibility in an incremental and iterative approach (see section 5.1).
- Planning and execution of functional system test campaign to further support OBSW development and to provide feedback for OBSW developers, continuing the same incremental and iterative mode of operation (see chapter 5).
- Preparation of automated test scripts for AFTs and per-release software tests and evaluation of different script execution frameworks (see chapter 6).
- Conduction of attitude control algorithm verification using an approach that combines tests performed with simulation technology and minimal open-loop functional system tests. Through the application of this approach, it was possible to reduce test complexity and to save costs for expensive ground support equipment (see section 5.4).
- Conduction of OBSW failure handling tests using the developed error injection framework (see section 6.2).
- Introduction of exploratory testing into the test approach for better discovery of formerly unidentified problems (see section 6.6).

7.1 Discussion

The discussion in this section tries to evaluate to which degree the goals of this thesis established in section 1.3 were met. The objectives pursued in this work can be categorized into three main areas of interest:

1. Is the described software development approach applicable to the *Flying Laptop* project and can general assumptions be derived about the applicability to similar projects?
2. In combination with this approach, how can functional verification for a small satellite be streamlined and simplified by simulation-based tests in a university context?
3. Furthermore, can software testing efforts be optimized by the use of modern test automation techniques, continuous testing, and continuous integration build chains?

Because the *Flying Laptop* mission was a success, it can already be argued that the approach is feasible. Whether it offers a better solution compared to other methods is highly dependent on the composition of teams and surrounding circumstances like organizational structure and the willingness of the team members to give up already established processes that they might be already familiar with.

While for the given reasons it is difficult to draw a general conclusion it can be determined that smaller projects like university-grade small satellites as well as larger projects like industry-grade satellites can both benefit from the presented approach. Smaller projects can increase software quality while saving costs to lower the risk of mission failure which is highly required especially in the context of CubeSat missions as mentioned in section 1.1.

Larger industry-grade projects, which on the other end of the spectrum already have high software quality, and thus success rates, can achieve shorter development terms if they are willing to let go of the principle to *only test finished software*. It is, of course, questionable if customers of larger industry-grade projects like major space agencies are willing to take the risk and apply such an agile approach.

One possible way of applying the approach to large-scale projects would be the re-build of an already existing platform. This re-build could include minor modifications to the original platform design and the success of such a project would be very likely because of the heritage from previous usages of the platform. Under such circumstances, a potential customer might be willing to agree to an agile system test driven process. The payload instrument for such a mission could still be classically developed following a V-model approach, but the platform which accommodates the payload instrument could be restructured in a more flexible, cost-efficient way.

7.1.1 Retrospective on the Software Development Approach

The primary goal of this thesis was to determine whether a flight software development approach which is guided by the system testing activities of a satellite is feasible. The underlying hypothesis is that such a software development approach can better adapt to design changes of the satellite system even late in the project, i.e., in phase C or even D. In a typical satellite project as conducted by space agencies, such late changes would dramatically delay the project and cause expensive reiteration cycles because of the formal methods that pervade these projects. It was shown in this thesis that by the application of this approach such delays can be minimized and reiteration cycles sped up because the agile reiteration cycles are much shorter and already part of the day-to-day work.

As [Boehm et al. 2009] suggests, neither agile nor plan-driven methods provide a silver bullet. "The future trends of software developments are towards balanced methods which apply aspects of both worlds." [Boehm et al. 2009] In this way of arguing, the presented approach offers a way to combine both worlds although the home grounds of the *Flying Laptop* project are clearly on the agile side as described in section 2.1.4 and illustrated by Figure 2.1.4.

A rather drastic but necessary consequence of the approach is that the majority of requirements engineering is left out which might not be possible in every classically structured project. Especially, if requirement-based quality assurance processes are already in place, this approach might not be readily applicable.

It is worth noting that the presented approach created a culture of embracing change over fearing or even rejecting it within the *Flying Laptop* team. The value of a specific change to the design, whenever brought up, was rationally discussed and often adopted throughout the project. One example of a drastic design change is the adoption of entirely different communication equipment. The initial design featured VHF/UHF amateur radio equipment which was exchanged with professional S-band TT&C transceivers. Another example of a very late design change is restructuring the composition of payloads in favor of an AIS ship signal receiver provided by DLR.

Even problems like the specification error of the PCDU, described in section 5.1.3.4, could be worked around in a time-saving and inexpensive way which speaks for the flexibility offered by the approach.

The approach also increased the user experience of the OBSW because small but helpful custom services were added to the original ECSS PUS service stack along the way (see Table 3.3.1). Custom ECSS PUS services like, e.g. service 206 (onboard parameter management service) were created because of the direct feedback from testers that had trouble working with ECSS PUS service 6 (memory management) alone while managing onboard parameters. These fresh ideas could quickly be implemented by the OBSW developers. Further additions to the original stack of TM/TC ground service were made possible due to the iterative nature and the direct communication between developers and testers. If the need for such additions became clear, custom ECSS PUS services were added to the original ground service stack no matter how far the development already progressed.

As described in section 5.6.3, a maybe less surprising realization is that while carrying out a very agile method of planning at the beginning of the functional system test campaign (see section 5.1 ff.), the style of test planning towards the end was more similar to the ones also performed in the classic approach as illustrated in Figure 5.6.2.

To summarize, the approach is best suited for small teams and projects with low or up to medium complex systems. It offers promising optimization potential as demonstrated for *Flying Laptop*. The larger the teams and the more complex the systems, the more difficulties will be encountered mainly because it is hard to keep large teams in sync for a task that requires a lot of communication and shared knowledge among team members. Most of the time, if teams get larger, the design methods become more plan-driven at which point the strengths of the approach are more and more diminished. A hybrid project or a "Scrum of Scrums" in which a less critical part of the system is built using the agile approach like, e.g., in a platform rebuild as mentioned above and the critical part is developed classically might be required.

7.1.2 Simplification of System Tests due to Simulation

Another goal of this thesis was to show that it is possible to simplify functional system tests by performing a large number of tests, which usually are performed in a closed-loop setup with the actual satellite, entirely in simulation.

The equipment models developed by [Falke 2008] et al. were reassessed and adjusted to fit the required degree of modeling detail. Models related to *Safe* mode attitude control and power supply received increased priority. These critical models were the PCDU model, the battery model, the solar panel model, the sun sensor model, the magnetometer model as well as the MGT model. The verification and calibration activities were primarily focused on these models. An example of such a verification activity is the mentioned end-to-end test of the sun sensor mockup described in section 5.4.3.

Significant simplifications in modeling detail are either justified because they do not affect OBSW behavior or the ability to run specific test cases. Simplifications were also accepted to a limited degree if they did not affect any critical model properties like e.g. the inaccurate spin-down curve of the reaction wheel model was tolerated (as described in section 6.1.3).

Additional dummy models without functionality besides power consumption and heat generation were created for payload equipment. A single model of the PLOC was created which provides dummy data as the central hub of payload communication to the OBSW. Because of early model calibration tests performed in the flatsat environment, the OBSW developers could directly test new functionalities in simulation and no longer required the real hardware components for many of their development activities. Furthermore, the flatsat enabled agile testing and development for the device handling part of the OBSW.

Examples like the problems due to an insufficient resemblance of the star tracker model described in section 5.4.5 show that simplifying functional system tests through shifting closed-loop tests to simulation environments, has limits. The approach struggles to reflect complex onboard equipment interactions like the one between star tracker and OBC.

Because no closed-loop tests with the real star tracker equipment were performed, the problem could not be identified in time from simulation runs alone. These potential shortcomings are a result of the trade-off that had to be made between full resemblance of the real unit and model complexity for the sake of speeding up model development. For the star tracker model, it was decided to sacrifice some of the complexity to handle the immense development effort that would otherwise be required.

Another aspect that could be considered to simplify system tests is to modularize onboard equipment. Better modularization would streamline flatsat tests since the many custom interfaces and data protocols caused a lot of overhead work for the developers. This amount of work could have been reduced if the interfaces would have been more similar for each component. Especially for payload equipment, the simplifications due to the usage of universal interfaces among payloads would have reduced the number of required iteration cycles during software development.

7.1.3 Automation of Software Tests and Build Chain

Through the automation of test procedures first using the commercial product MOIS and later using the PyOps as described in section 6.4, whole categories of tests became repeatable. This increases the overall software quality significantly because the automated tests cover great parts of the OBSW codebase for every release while at the same time reducing the workload on the team. One important aspect to chose an automation technique that integrates well with the already established build and test environment. It is of no use if the solution is incompatible with the build chain or causes problems that have to be resolved manually. E.g., all of the benefits gained from an automation solution are lost if the tester has to manually check what went wrong in every other test run. In short, an automated test environment has to be dependable to be useful.

Especially the creation of automated regression tests was rewarding. E.g., the deployment controller mentioned in 5.1.3.4 is an OBSW component which was used only on a few occasions, i.e., in LEOP at the start of the mission and during solar panel deployment tests. In the meantime, the functionality was not used which carries the risk that a developer might break the functionality unintentionally. If there had been no regression tests in place that were repeated automatically with every release, this might have remained unnoticed. In the best case, the problem is found later during repeated tests which would still cause more work than if the problem was found directly after it was introduced. In the worst case, such a problem might cost the mission.

7.1.4 Increasing OBSW Quality

Although unit tests, integration tests, and system-level tests are nowadays considered best practices in software development, the overall quality of their implementation varies strongly from project to project. The positive impact of software tests and CI tools on software quality cannot be overstated.

This work has shown that a mixture of different software test methods is suited for maintaining the software quality of the FSFW and OBSW. In this mixture, unit tests are the individual responsibility of the OBSW developer, system-level automated test scripts, regression test scripts, and exploratory testing are implemented and performed by independent software testers. This mixture of tests that are implemented by the software developers themselves and the independent testers, has proven to be desirable.

One fundamental problem in software testing is to determine when to stop. That means when has software been tested sufficiently. Most programs are so complex that it is impossible to find test cases for each and every possible special case or execution path. And if it is possible these test cases would be so numerous that it is impractical to run them all in a reasonable amount of time. In traditional engineering requirements play a significant role in determining test fulfillment (see section 2.1) but the fulfillment of requirements only testifies that all desired features are present and working as intended in the software; not that the software itself is error-free. Also, having many tests in place can create misleading comfort because as ever so often quantity is not quality. A lot of test cases do not necessarily mean good coverage. Sometimes there are legal obligations towards the amount of testing enforced by public authorities. These authorities often demand a certain testing method and minimum code coverage. But all in all, there is a lack of a meaningful, general way to measure test completeness.

The application of exploratory testing (described in section 6.6) was a fruitful experience. Not only did the exploration of edge cases during usage of the OBSW uncover lots of bugs that otherwise would have gone under the radar.

It is important to note that exploratory testing cannot be automated. The reason for this is that it involves the tester's creativity. Instead, the tester should manually perform all steps and reiterate the test procedure along with his or her own learning curve. The reason for this is that automated test procedures are designed with a special test case in mind, are implemented statically and early in a project usually together with a new feature. This means that the software will be tested well for those obvious test cases, but other test cases which might be less obvious are never explored.

By applying more manual testing techniques like exploratory testing, the timing for different activities will vary more because of side effects caused by user input. The likelihood to uncover problems related to timing is thus increased.

7.1.5 Risk and OBSW Metrics

The strategy to manage risk by shifting test efforts toward vital functionalities of the satellite was successful. The only bugs encountered during the *Flying Laptop* mission were in higher-order functionalities. E.g., fine pointing attitude control did not reach the envisaged pointing accuracy at the beginning of the mission. It was later improved through OBSW updates.

Problems like the one described in section 5.4.5 and others related to proper sensor data fusion caused these initial difficulties. The STB served as a vital test environment to produce these OBSW updates. Other software bugs were related to the usability of payloads that did not endanger the mission itself but rather created an increased workload for the operators. All of these problems were successfully patched in orbit.

But there are other risks that come with this approach. Due to the lack of planning and the openness to late changes, the risk of navigating into a dead-end and overlooking hard to solve compatibility issues is higher than in comparable projects with abundant, up-front planning phases.

Based on the metrics gathered from bug tracking tools, an insightful picture of the general performance of the project can be drawn. Figure 7.1.1 shows the overall number of bugs filed with Bugzilla during the course of the OBSW development.

Figure 7.1.1 shows a constant rate of increase in discovered bugs of about 1.5 bugs per working day during the bread and butter period of the OBSW development which was between 2013 and 2017 where system testing was carried out. In 2017 there is a noticeable drop in discovered bugs which can be attributed to the different style of testing close to the launch date in July 2017. In this phase, the OBSW team focused more on finishing the first Apollo release in which fewer new functionalities were added to the OBSW but rather the usability was improved along with the preparation intensive operational tests (see section 5.6).

The upper graph of Figure 7.1.1 shows that on average during OBSW development about 30 bugs are constantly pending to be solved at a time (orange area). The bottom graph of Figure 7.1.1 also shows the same trend for only the Apollo release. Around November of 2017, a period of increased activity began which marks the OBSW development that was performed for the first in-orbit OBSW update.

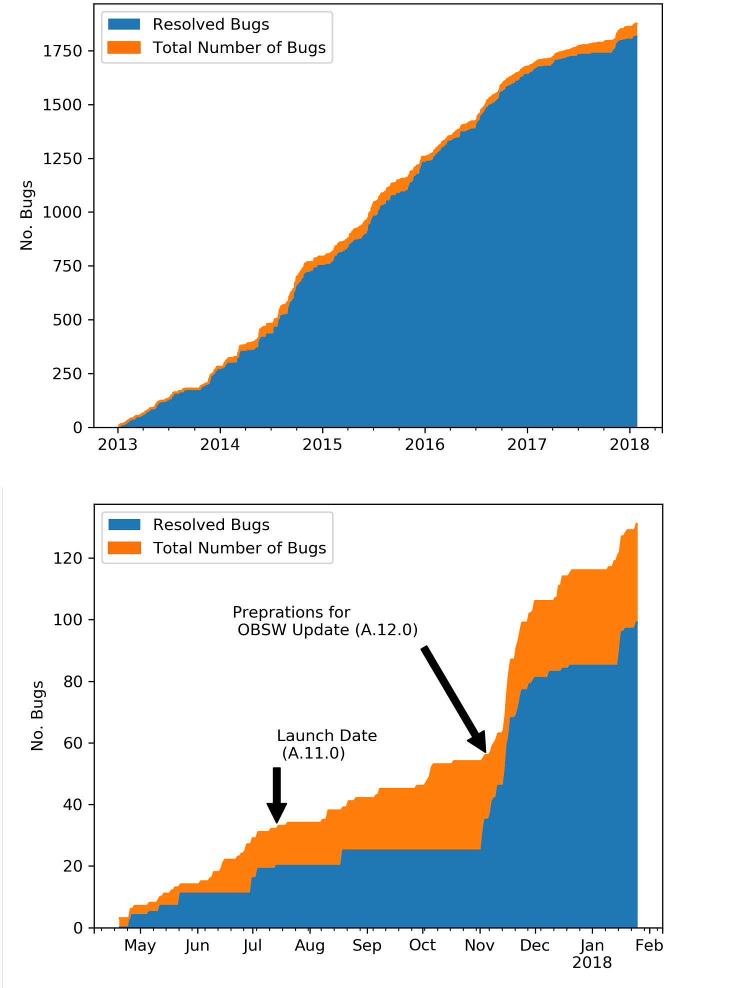


Figure 7.1.1: Bug trend over time for OBSW; (top graph) all releases combined; (bottom graph) Apollo releases in 2017; orange area marks the amount of open bugs at each point in time

Table 7.1.1 contains the key quality-related metrics for the OBSW. The metrics are based on the considerations given in section 2.3.7. As mentioned in section 2.3.7 it is not a recommended method to deduce function points from SLOC (using the backfire method). Table 7.1.1 still lists this figure for the lack of a better alternative as it can still be used to gain a rough understanding of the OBSW quality since no FP analysis was performed for the OBSW in the first place.

A rough conversion ratio for the C++ programming language lies between 50 and 80 [QSM Website 2018]. This way, an equivalent function point number for the OBSW can be expressed which is only meant to provide a rough estimate about the functional complexity of the OBSW to compare it to reference metrics and to express defect ratio per function point.

Since the agile development approach is not directly comparable to traditional software metrics, the figures given in Table 7.1.1 need to be considered carefully before comparing them to other software projects. The number of overall bugs includes all filed bugs since the start of the development (Mercury). For this reason, the delivered bug density is the metric which should be regarded as the most meaningful metric in this context because it does not include filed bugs before launch (OBSW delivery).

Table 7.1.1: Overview of software quality metrics of OBSW

Single lines of code (A.13.0-46)	70785
Equivalent FP range (based on SLOC/FP ratio 80/50)	885/1425
Overall number of bugs until 90 days after launch	1794
Overall closed bugs until launch	1730
Overall bug density (Bugs/KLOC)	25
Overall bug density (Bugs/FP)	1.25/2.0
Delivered bug density (Bugs/KLOC)	1.1
Delivered bug density (Bugs/FP)	0.05/0.08
Bug removal rate	0.96 (96%)

Compared to industry figures [Jones 2008], it can be summarized that the quality metrics of the OBSW are well comparable to the average industry software product with respect to bug density and bug removal rate. This speaks for the presented approach because not only can the software development process be streamlined by its application but it can also produce competitive software quality.

7.2 Outlook

The development of the *Flying Laptop OBSW* and especially the FSFW does not stop with the launch, nor with the end of the *Flying Laptop* mission. In fact, it is envisaged to continue the development for an indefinite time, as long as there is demand.

In this way, the codebase will constantly be refined and improved by the team of students at the University of Stuttgart and the FSFW can be used as a basis for many projects to come. With the help of the system test driven approach, it is possible to create a sustainable way of adopting the FSFW for other missions. These missions could then already profit from the foundation that has been established through the successful *Flying Laptop* project. Especially the reusable part of the OBSW, the FSFW presents an already matured and flight-proven embedded real-time software framework and a lot of the test activities that have been performed for *Flying Laptop* will not have to be repeated. The mission-specific part of the OBSW can be used as a reference implementation that other missions can use as a guideline on how to implement their mission-specific code.

By open-sourcing the FSFW, a further step towards broad availability for other missions was made. The contributions of other missions will further improve the code base with the goal of providing a proven-in-use flight software framework for future space missions.

As for the spacecraft simulation part of this thesis, a project was already started at the University of Stuttgart to overhaul the simulation models of *Flying Laptop* and to create a simulation environment that is better suited for future missions and for more virtualized environments using state of the art simulation technology. By rewriting the simulation models to also support the SMP standard a better baseline for the exchange of models for the rapid creation of simulators for new spacecraft is being established.

Another step forward that would improve the automation of software tests would be the creation of fully virtualized testbeds where spacecraft simulation, OBSW, and test agent run in an isolated containerized environment. This would allow to better integrate the automated tests in a CI environment. Such a virtualized testbed could be spawning multiple instances of simulation and OBSW to parallelize testing and speed up the release based testing process significantly in the spirit of CT. These containers, of course, would lose the benefits of the OBC hardware-in-the-loop testbed like real-time capability but the STB but could still be used for the majority of per release tests where the real-time capability of the system is not necessarily required such as system mode transition tests, reaction to failures and TC processing and TM generation.

Another way to further improve the gathering of test quality metrics would be to implement code coverage analysis into the test environment either for the STB or for a version of the FSFW running on Linux systems which would best leverage the possibilities of code coverage analysis tools. By having more code coverage metrics available for the automated test procedures, they could be step by step improved to increase their code coverage rate. By receiving direct feedback on the effect of an additional test step in the test procedure, a tester could better design test cases.

For this work, test cases have been generated manually. One further, potentially interesting follow-up research would be to investigate how event patterns (mentioned in section 6.2.2) can be used to generate test cases automatically. More generally, a smarter way of creating test cases semi-automatically or completely automatically would benefit the overall test coverage and also speed up the process. Methods to increase the diversity of test cases while still cover all relevant equivalence classes like search-based testing [McMinn 2004] or the application of fuzzing techniques [Godefroid et al. 2008] can probably help to increase the overall software quality.

BIBLIOGRAPHY

[Balzert 2011] Balzert, H. "Lehrbuch Der Softwaretechnik: Entwurf, Implementierung, Installation Und Betrieb". 3 ed. SpringerLink. Heidelberg: Spektrum Akademischer Verlag, 2011, ISBN: 978-3-8274-1706-0 (Druck-Ausgabe)

[Barschke et al. 2016] Barschke, M.; Brieß, K.; Renner, U. "Twenty-Five Years of Satellite Development at Technische Universität Berlin", Small Satellites Systems and Services Symposium (4S) (2016).

[Bass et al. 2015] Bass, L.; Weber, I.; Zhu, L. "DevOps: A Software Architect's Perspective". Addison-Wesley Professional, 2015

[Baumann et al. 2010] Baumann, C.; Beckert, B.; Blasum, H.; Bormer, T. "Ingredients of Operating System Correctness", Embedded World Conference, Nürnberg, Germany (2010).

[Bätz 2018] Bätz, B. "Design and Implementation of a Spacecraft Flight Software Framework", PhD Thesis, Stuttgart: University of Stuttgart, 2018, expected

[Bätz et al. 2014] Bätz, B.; Mohr, U.; Witt, R.; Bucher, N.; Eickhoff, J. "Applying Dynamic Development Methods to Satellite Software: The Software Framework for the FLP Micro-Satellite Platform", The 4S Symposium 2014 Proceedings (2014).

[Becher 2017] Becher, S. "Development of a Sun Sensor Model for the Verification of the Sun Direction Estimation of the Small Satellite Flying Laptop", PhD Thesis, Stuttgart: University of Stuttgart, November 15, 2017, Unknown

[Beck 2003] Beck, K. "Test-driven Development: By Example". Boston: Addison-Wesley, 2003, ISBN: 9780321146533

[Beck et al. 2005] Beck, K.; Andres, C. "Extreme Programming Explained: Embrace Change". Boston, MA: Addison-Wesley, 2005, ISBN: 9780321278654

[Beck et al. 2018] "Manifesto for Agile Software Development" http://agilemanifesto.org/iso/en/manifesto.html (accessed January 9, 2018).

[Boehm et al. 2009] Boehm, B. W.; Turner, R. "Balancing Agility and Discipline: A Guide for the Perplexed". Boston, MA: Addison-Wesley, 2009, ISBN: 0321186125

[Böhringer 2014] Böhringer, F. "Nutzlasten Des Universitären Kleinsatelliten Flying Laptop - Entwicklung, Aufbau Und Qualifikation", PhD Thesis, München: Verlag Dr Hut (VDH), February 12, 2014, ISBN: 9783843915540, Published http://www.dr.hut-verlag.de/978-3-8439-1554-0.html.

[Brieß 2017] Brieß, K. "Small Satellite Programmatics of the Technische Universität Berlin", Small Satellites for Earth Observation, 11th International Symposium of the International Academy of Astronautics (IAA) (2017).

[Brown 2002] Brown, C. D. "Elements of Spacecraft Design". Reston, VA: American Institute of Aeronautics and Astronautics, Inc, 2002, ISBN: 1563475243

[Carlson et al. 2014] Carlson, N.; Laplante, P. "The NASA Automated Requirements Measurement Tool: A Reconstruction", Innov. Syst. Softw. Eng. 10, no. 2 (2014): doi:10.1007/s11334-013-0225-8. http://dx.doi.org/10.1007/s11334-013-0225-8.

[Chaudhri et al. 2006] Chaudhri, G.; Cater, J.; Kizzort, B. "A Model for a Spacecraft Operations Language", SpaceOps 2006 Conference (2006): 5708.

[Demeuse et al. 1998] Demeuse, B.; Valera, S. "PLUTO, a Procedure Language for Users in Test and Operations", DASIA 98 - Data Systems in Aerospace 422 (1998): 307.

[Desjardins et al. 1978] Desjardins, R.; Hall, G.; Mcguire, J.; Merwarth, P.; et al. "GSFC Systems Test and Operation Language (STOL) Functional Requirements and Language Description" (1978).

[ECSS Committee 2009] "ECCS Standard on Project Planning and Implementation" Report, European Space Agency, http://ecss.nl/standard/ecss-m-st-10c-rev-1-project-planning-and-implementation/, March 6, 2009, European Space Agency

[ECSS Committee 2003] "ECSS Standard on Ground Systems and Operations - Telemetry and Telecommand Packet Utilization" Report, European Space Agency, January 30, 2003, European Space Agency

[ECSS Committee 2011] "ECSS Standard on Simulation Modelling Platform (SMP)" Report, European Space Agency, January 25, 2011, European Space Agency

[ECSS Committee 2009] "ECSS Standard on Space Software Engineering" Report, European Space Agency, March 6, 2009, European Space Agency

[ECSS Committee 2017] "ECSS Standard on Space Software Product Assurance" Report, European Space Agency, February 15, 2017, European Space Agency

[ECSS Committee 2012] "ECSS Standard on Testing" Report, European Space Agency, June 1, 2012, European Space Agency

[ECSS Committee 2018] "ECSS Standard on Verification" Report, European Space Agency, February 1, 2018, European Space Agency

[Eickhoff 2013] Eickhoff, J. "A Combined Data and Power Management Infrastructure for Small Satellites". Berlin Heidelberg: Springer, 2013, ISBN: 9783642355561

[Eickhoff 2016] Eickhoff, J. "The FLP Microsatellite Platform Flight Operations Manual". Cham: Springer International Publishing, 2016, ISBN: 9783319235028

[Eickhoff 2012] Eickhoff, J. "Onboard Computers, Onboard Software and Satellite Operations: An Introduction". Berlin ; New York: Springer, 2012, ISBN: 9783642251696

[Eickhoff 2018] Eickhoff, J. "Satellite System Verification". Lecture Manuscript, Institute of Space Systems, University of Stuttgart, Rev. WS 2018, ISSN 1866-737, 2018

[Eickhoff 2009] Eickhoff, J. "Simulating Spacecraft Systems". Heidelberg ; New York: Springer, 2009, ISBN: 9783642012754

[Eickhoff et al. 2006] Eickhoff, J.; Falke, A.; Röser, H. P. "Model-based Design and Verification - State of the Art From Galileo Constellation Down to Small University Satellites", Acta Astronautica 61, no. 1-6 (2006).

[Falke 2008] Falke, A. "Modell-basierte Systemsimulation Eines Kleinsatelliten Mit Einem FPGA-basierten On-board Computer", PhD Thesis, 2008, Unknown

[Fowler et al. 1999] Fowler, M.; Beck, K. "Refactoring: Improving the Design of Existing Code". Reading, MA: Addison-Wesley, 1999, ISBN: 0201485672

[Freeman et al. 2010] Freeman, S.; Pryce, N. "Growing Object-oriented Software, Guided by Tests". Upper Saddle River, NJ: Addison Wesley, 2010, ISBN: 0321503627

[Fritz 2013] Fritz, M. W. "Hardware- Und Software-Kompatibilitätstests Für Den Bordrechner Eines Kleinsatelliten", PhD Thesis, April 29, 2013, Unknown

[QSM Website 2018] QSM. http://www.qsm.com/resources/function-point-languages-table (accessed September 24, 2018).

[Garcia 2008] Garcia, G. "Use of Python As a Satellite Operations and Testing Automation Language", GSAW2008 Conference, Redondo Beach, California (2008).

[Godefroid et al. 2008] Godefroid, P.; Levin, M. Y.; Molnar, D. A. "Automated Whitebox Fuzz Testing", NDSS 8 (2008): 151-166.

[Grünfelder 2017] Grünfelder, S. "Software-Test Für Embedded Systems: Ein Praxishandbuch Für Entwickler, Tester Und Technische Projektleiter". Dpunkt. Verlag GmbH, 2017, ISBN: 9783864904486

[Hagel 2018] Hagel, P. "Modular Software Architecture for FPGA Based Payload Module Computers", PhD Thesis, 2018, Unpublished

[Helvajian et al. 2008] Helvajian, H.; Janson, S. W. "Small Satellites: Past, Present, and Future". El Segundo, Calif.: Aerospace Press ; Reston, Va. : American Institute of Aeronautics and Astronautics, 2008, ISBN: 9781884989223

[Hendricks et al. 2005] Hendricks, R.; Eickhoff, J. "The Significant Role of Simulation in Satellite Development and Verification", Aerospace Science and Technology 9, no. 3 (2005): 273 - 283.

[Hernek et al. 2008] "ESA Guide for Independent Software Verification & Validation" Report, European Space Agency, December 29, 2008, European Space Agency

[IEEE 1998] IEEE, S. E. S. "IEEE Recommended Practice for Software Requirements Specifications: IEEE Std 830-1998". New York: IEEE, 1998, ISBN: 0738103322

[Irvine et al. 2002] Irvine, M.; Bégin, M.; Eickhoff, J.; de Kruyf, J. "System Simulation and Verification Facility (SSVF)", Data Systems in Aerospace Vol. 509 (2002).

[Johnson et al. 2013] Johnson, B.; Song, S.; Murphy-Hill, E.; Bowdidge, R. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?", Proceedings of the 2013 International Conference on Software Engineering (2013): 672-681.

[Jones 2008] Jones, C. "Measuring Defect Potentials and Defect Removal Efficiency", CrossTalk The Journal of Defense Software Engineering 21, no. 6 (2008): 11-13.

[Jungmayr 2004] Jungmayr, S. "Improving Testability of Object Oriented Systems". Berlin: Dissertation.de, 2004, ISBN: 3898257819

[Kaner 2008] Kaner, C. "A Tutorial in Exploratory Testing", Tutorial presented at QUEST2008.(Available online at: http://www. kaner. com/pdfs/QAIExploring. pdf, accessed: 26 Jan 2014) (2008).

[Klemich et al. 2016] Klemich, K.; Bucher, N.; Böttcher, M.; Braun, J.; et al. "A Ground Segment for Small Satellite Operations in a University Context Combining Professional and Custom Software Tools", 14th International Conference on Space Operations (SpaceOps 2016) Proceedings (2016): 3505.

[Kramer et al. 2008] Kramer, H. J.; Cracknell, A. P. "An Overview of Small Satellites in Remote Sensing", International Journal of Remote Sensing 29, no. 15 (2008): doi:10.1080/01431160801914952.

[Kries et al. 2002] Kries, W. V.; Schmidt-Tedd, B.; Schrogl, K. -U. "Grundzüge Des Raumfahrtrechts: Rahmenbestimmungen Und Anwendungsgebiete". München: Beck, 2002, ISBN: 9783406497421

[Kubrick 1968] Kubrick, S. "2001: A Space Odyssey", Motion Picture, Metro-Goldwyn-Mayer, 1968

[Laplante 2013] Laplante, P. A. "Requirements Engineering for Software and Systems". 2nd ed. Boston, MA, USA: Auerbach Publications, 2013, ISBN: 9781466560819

[Lengowski et al. 2015] Lengowski, K.; Steinmetz, N.; Bucher, M.; Klemich, J.; et al. "Environmental Flight Acceptance Tests of the Small Earth Observation Satellite Flying Laptop", Proceedings of 29th Annual AIAA/USU Conference on Small Satellites (2015). https://digitalcommons.usu.edu/smallsat/2015/.

[Ley et al. 2011] Ley, W.; Wittmann, K.; Hallmann, W. "Handbuch Der Raumfahrttechnik". München: Hanser, 2011, ISBN: 9783446424067

[Liggesmeyer 2009] Liggesmeyer, P. "Software-Qualität: Testen, Analysieren Und Verifizieren Von Software". Heidelberg: Spektrum, Akad. Verl., 2009, ISBN: 9783827422033

[Linz 2017] Linz, T. "Testen in Scrum-Projekten: Leitfaden Für Softwarequalität in Der Agilen Welt". 2nd ed. Heidelberg: dpunkt.verlag, February 1, 2017, ISBN: 3864904145

[Lions 1996] "ARIANE 5 Flight 501 Failure" Report, Ariane 5 Flight 501 Inquiry Board, http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, July 19, 1996, Ariane 5 Flight 501 Inquiry Board

[Louridas 2006] Louridas, P. "Static Code Analysis", IEEE Software 23, no. 4 (2006): 58-61.

[McMinn 2004] McMinn, P. "Search-based Software Test Data Generation: A Survey", Software testing, Verification and reliability 14, no. 2 (2004): 105-156.

[Müller 2018] Müller, A. "Conceptual Design of An Operations Language to Automate Satellite Operations". University of Stuttgart: Institute of Space Systems, Master's Thesis, August, 2018

[Myers et al. 2011] Myers, G. J.; Sandler, C.; Badgett, T. "The Art of Software Testing". John Wiley & Sons, 2011

[NASA 1995] "Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems" Report, NASA, https://ntrs.nasa.gov/search.jsp?R=19980227975, July 1, 1995, NASA

[NASA 2006] "Software Assurance Standard" Report, NASA, https://standards.nasa.gov/standard/nasa/nasa-std-87398, May 5, 2006, NASA

[Pan 1999] Pan, J. "Software Testing", Dependable Embedded Systems 5 (1999): 2006.

[Parkes et al. 2005] Parkes, S.; McClements, C. "Space Wire Remote Memory Access Protocol", DASIA 2005-Data Systems in Aerospace 602 (2005).

[Parnas et al. 1986] Parnas, D. L.; Clements, P. C. "A Rational Design Process: How and Why to Fake It", IEEE Trans. Softw. Eng. 12, no. 2 (1986): 251-257.

[Pearson et al. 2012] Pearson, S.; Trifin, F.; Reid, S.; Heinen, W. "An Integrated Development and Validation Environment for Operations Automation", ESA Requirements and Standards Division, ESTEC, The Netherlands, Spaceops (2012).

[Poensgen et al. 2012] Poensgen, B.; Plewan, H. -J. "Produktive Softwareentwicklung: Bewertung Und Verbesserung Von Produktivität Und Qualität in Der Praxis". dpunkt. verlag, 2012

[Rau 2016] Rau, K. -H. "Agile Objektorientierte Software-entwicklung: Schritt Fur Schritt Vom Geschaftsprozess Zum ... Java-programm". [Place of publication not identified]: Morgan Kaufmann, 2016, ISBN: 9783658007751

[Roeser et al. 2005] Roeser, H.; Huber, F.; Grillmayer, G.; Lengowski, M.; et al. "A Small Satellite of the University Stuttgart - a Demonstrator for New Techniques", Proceedings of the 31st International Symposium on Remote Sensing of Environment (ISRSE) at NIERSC (Nansen International Environmental and Remote Sensing Center), Saint Petersburg, Russia (2005).

[Rosenberg 1997] Rosenberg, J. "Some Misconceptions About Lines of Code", Proceedings Fourth International Software Metrics Symposium (1997): doi:10.1109/METRIC.1997.637174.

[Rubin et al. 2008] Rubin, J.; Chisnell, D. "Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests". Indianapolis, IN: Wiley Pub, 2008, ISBN: 9780470185483

[Sander et al. 1995] Sander, P.; Stucky, W.; Herschel, R. "Automaten, Sprachen, Berechenbarkeit". Stuttgart: Teubner, 1995, ISBN: 351912937X

[Schoenermark et al. 2005] Schoenermark, M.; Grillmayer, G; Lengowski, Michael; Walz, S; et al. "Earth Observation with the Flying Laptop – a Small Satellite of the University of Stuttgart", Proceedings of the 31st International Symposium on Remote Sensing of Environment (ISRSE) at NIERSC (Nansen International Environmental and Remote Sensing Center), Saint Petersburg, Russia (2005): 20-24.

[Schwaber et al. 2002] Schwaber, K.; Beedle, M. "Agile Software Development with Scrum". Upper Saddle River, NJ: Prentice Hall, 2002, ISBN: 0130676349

[Shepherd 1965] Shepherd, C. M. "Design of Primary and Secondary Cells—Part 2. An Equation Describing Battery Discharge", Journal of Electrochemical Society (1965): 112.

[Sikora et al. 2012] Sikora, E. T.; Pohl, B.; Klaus "Industry Needs and Research Directions in Requirements Engineering for Embedded Systems", Requirements Engineering 17, no. 1 (2012): doi:10.1007/s00766-011-0144-x.

[St. John et al. 1987] St. John; Moorman; Brown, B. W. "Real-time Simulation for Space Station", IEEE, Proceedings (1987): 383-398. https://ntrs.nasa.gov/search.jsp?R=19870050024.

[Steinmetz 2016] Steinmetz, F. "Realisierung Des Thermalkontrollsystems Für Einen Kleinsatelliten", PhD Thesis, München: Verlag Dr. Hut, 2016, Unknown

[Stephenson 1999] "Mars Climate Orbiter Mishap Investigation Board Phase I Report" Report, November 10, 1999,

[Stickler et al. 1976] Stickler, A. C.; Alfriend, K. T. "Elementary Magnetic Attitude Control System", Journal of Spacecraft and Rockets 13, no. 5 (1976): doi:10.2514/3.57089.

[Swartwout 2017] Swartwout, M. "CubeSats and Mission Success: 2017 Update", Electron. Technol. Workshop, NASA Electron. Parts Packag. Program (NEPP) Electron. Technol. Workshop (ETW) (2017).

[Swartwout 2016] Swartwout, M. "Secondary Spacecraft in 2016: Why Some Succeed (And Too Many Do Not)", 2016 IEEE Aerospace Conference 5-12 March 2016 (2016) doi:978-1-4673-7676-1.

[Terraillon et al. 2010] Terraillon, J. -L.; Jung, A.; Arberet, P.; Montenegro, S.; et al. "Space On-Board Software Reference Architecture", DASIA 2010 - Data Systems In Aerospace 682 (2010): 58.

[Viscor 2006] Viscor, T. "SSETI - Past, Present and Future", Proceedings of the 20th Annual AIAA/USU Conference on Small Satellites (2006).

[Wertz et al. 2011] Wertz, J. R.; Everett, D. F.; Puschell, J. J. "Space Mission Engineering: The New SMAD". Hawthorne, CA: Microcosm Press, 2011, ISBN: 9781881883159

[Witt 2016] Witt, R. "Failure Detection, Isolation and Recovery Capabilities for the University Small Satellite Flying Laptop", PhD Thesis, München: Verlag Dr Hut (VDH), 2016, ISBN: 9783843926737, Published

[Wolf et al. 2005] Wolf, H.; Roock, S.; Lippert, M. "EXtreme Programming: Eine Einführung Mit Empfehlungen Und Erfahrungen Aus Der Praxis". Heidelberg, Germany: Dpunkt.verlag, 2005, ISBN: 3898643395

[Woodling et al. 1973] "Apollo Experience Report: Simulation of Manned Space Flight for Crew Training" Report, NASA Johnson Space Center, https://ntrs.nasa.gov/search.jsp?R=19730011149, March 1, 1973, NASA Johnson Space Center

APPENDIX A. Detailed Functional Test Program

Table i: Full Summary of System Level Tests Performed for Flying Laptop (part 1/2)

Test type	Name	Performed at (test facility)	Date
ACS	Actuator/Sensor Polarity Tests (all units)	IRS	Q1/2 2016
	Safe Mode Open-Loop Plausibility Test	IRS	Q4 2016
	RW Assignment Check	IRS	23/01/2014
	RW Characterization through Spin-Down	IRS	continuous
	MGT Coil Assignment Check	IRS	20/02/2017
	Sun Sensor Open-Loop Mockup Test	IRS	Q2 2017
	Sun Sensor Allocation Check	IRS	01/07/2015
	Residual Magnetic Field Characterization	Airbus DS	15/07/2014
	Pulse-per-second (PPS) Characterization	IRS	06/06/2013
	GPS Signal Reception Test	IRS	03/07/2015
	GPS Orbit Scenario Simulation	IRS	11/08/2016
	STR Attitude Solution from Stimulated Star Map	IRS	25/01/4016
	STR Image Downlink	IRS	21/08/2016
	STR Baffle Light Proofness Test	IRS	26/01/2016
CDH	OBC Reconfiguration Test	IRS	25/07/2013
	OBSW Update Test	IRS	05/04/2015
EMC	Electromagnetic Compatibility Test	Airbus DS	15/07/2014
ENV	Thermal Vacuum Test w/ structural thermal model	CSL	16/01/2013
	Sine Vibration Test w/ structural thermal model	CSL	Feb. 2013
	Random Vibration Test w/ structural thermal model	CSL	Feb. 2013
	Sinusoidal Wavelet Vibration Test w/ structural thermal model	CSL	Feb. 2013
	Sine Vibration Test w/ flight model	ZARM	04/11/2014
	Random Vibration Test w/ flight model	ZARM	06/11/2014
	Thermal-Balance Test w/ flight model	DLR	25/11/2014
	Static Load Test w/ flight model	Tesat Spacecom	26/07/2016
	Calibration of Structural Model for Shock Load Simulations	IRS	04/12/2016
	Determination of Center of Gravity and Mass	IRS	28/03/2014

Table ii: Full Summary of System Level Tests Performed for Flying Laptop (part 2/2)

Test type	Name	Performed at (test facility)	Date
FNC	Abbreviated Functional Tests	IRS	continuous
	Thermal-Vacuum Test	DLR	18/11/2014
	System Mode Transitions Tests	STB	continuous
	FDIR Error Detection Test	STB	continuous
	FDIR Error Handling Test	STB	continuous
	Flatsat Device Compatibility Tests	IRS	Q2/3 2013
	Payload Flatsat Device Compatibility Tests	IRS	Q1/2 2014
OPS	Launch and Early Orbit Phase Operations Test	IRS	04/10/2016
	One Week of Nominal Operation	IRS	05/12/2016
	Payload Operation Test	IRS	22/03/2017
P/L	MICS Alignment	IRS	Q1 2017
	MICS Calibration	IRS	Q1 2017
	PAMCAM Alignment	IRS	Q1 2017
	DDS Payload Data Downlink	IRS	Q3 2016
	AIS Data Reception Test	IRS	Q1 2016
	OSIRIS Data Transmission Test (bit pattern)	IRS	18/082015
	OSIRIS Alignment Test	IRS	08/09/2016
PSS	Power Supply Subsystem Test	IRS	27/06/2013
	Complete PSS Shutdown and Recovery (a.k.a. Muenchhausen) Test	IRS	27/11/2013
	Solar Panel Deployment Test	IRS	04/02/2016
	Characterization of Onboard Device Power Consumption	IRS	01/08/2014
RFC	RF Compatibility Test w/ IRS G/S	IRS	22/02/2016
	RF Compatibility Test w/ Weilheim G/S	WHM	21/06/2016
	Antenna Characterization	DLR	28/06/2016

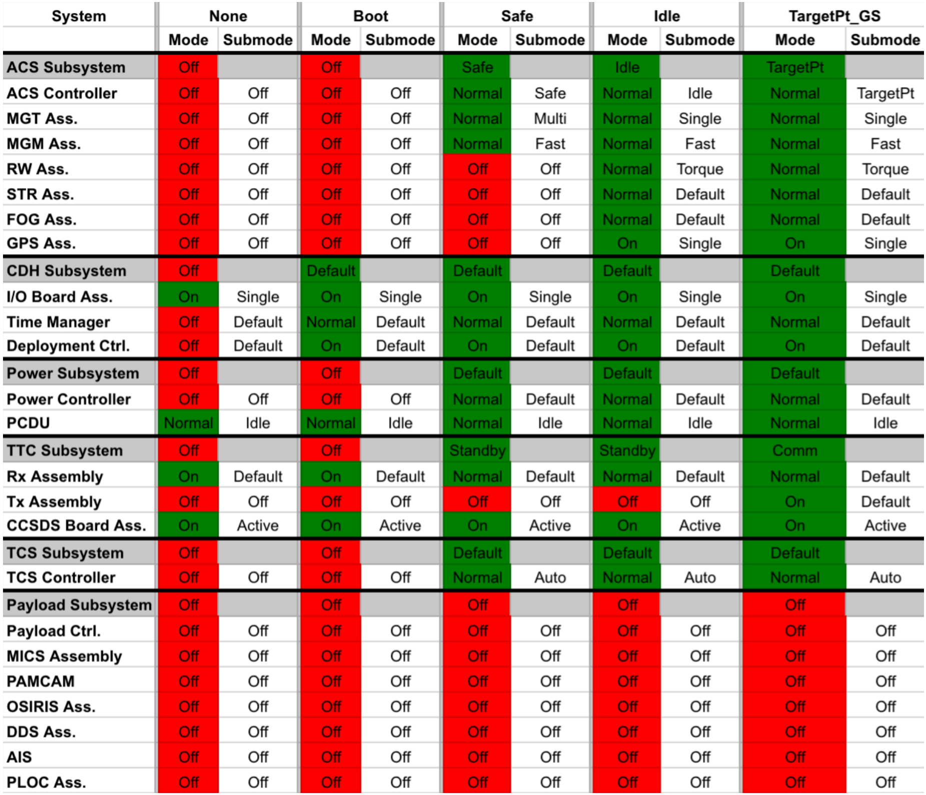
APPENDIX B. *Flying Laptop* Simulation Models

Table iii: List of Simulation Models for the Flying Laptop Simulator

Subsystem	Name	Description	Number of instances	Type
ACS	RW_SW	Reaction Wheel Model	4	Equipment
	MGT_SW	Magnetic Torquer Model	2	Equipment
	MGM_SW	Magnetometer Model	2	Equipment
	SUS_SW	Sun Sensor Model (One single instance for all sun sensors)	1	Equipment
	GPS_SW	GPS Receiver Model	3	Equipment
	STR_SW	Star Tracker Model	1	Equipment
	FOG_SW	Fibre-optical Gyroscope Model	3	Equipment
PSS	PCDU_SW	Power Control and Distribution Unit Model	1	Equipment
	SolarPnl_SW	Solar Panel Model	3	Equipment
	Battery_SW	Battery String Model	3	Equipment
COM	TTCRX_SW	TT&C Receiver Model	2	Equipment
	TTCTX_SW	TT&C Transmitter Model	2	Equipment
	DDS_SW	Data Downlink Transmitter Model	2	Equipment
TCS	HEATER_SW	Heater Unit Model	8	Equipment
	HMU_SW	PT1000 Temperature Sensor Model	> 50	Equipment
P/L	AIS_SW	AIS Receiver Model	1	Equipment
	DOM_SW	Deployment Mechanism Model	1	Equipment
	MICS_SW	Multispectral Camera Model	3	Equipment
	PAMCAM_SW	Panoramic Camera Model	1	Equipment
	PLOC_SW	Payload Computer Model	2	Equipment
	OSIRIS_SW	Optical Data Downlink Transmitter Model	1	Equipment
SIM	EnvDyn_SW	Environment and Dynamics Model	1	Physics
	Thermal_SW	Thermal Model	1	Physics

APPENDIX C. System Modes

Table iv: Flying Laptop System Mode Definitions (part 1/5)



System	None		Boot		Safe		Idle		TargetPt_GS	
	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode
ACS Subsystem	Off		Off		Safe		Idle		TargetPt	
ACS Controller	Off	Off	Off	Off	Normal	Safe	Normal	Idle	Normal	TargetPt
MGT Ass.	Off	Off	Off	Off	Normal	Multi	Normal	Single	Normal	Single
MGM Ass.	Off	Off	Off	Off	Normal	Fast	Normal	Fast	Normal	Fast
RW Ass.	Off	Off	Off	Off	Off	Off	Normal	Torque	Normal	Torque
STR Ass.	Off	Off	Off	Off	Off	Off	Normal	Default	Normal	Default
FOG Ass.	Off	Off	Off	Off	Off	Off	Normal	Default	Normal	Default
GPS Ass.	Off	Off	Off	Off	Off	Off	On	Single	On	Single
CDH Subsystem	Off		Default		Default		Default		Default	
I/O Board Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
Time Manager	Off	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Deployment Ctrl.	Off	Default	On	Default	On	Default	On	Default	On	Default
Power Subsystem	Off		Off		Default		Default		Default	
Power Controller	Off	Off	Off	Off	Normal	Default	Normal	Default	Normal	Default
PCDU	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle
TTC Subsystem	Off		Off		Standby		Standby		Comm	
Rx Assembly	On	Default	On	Default	Normal	Default	Normal	Default	Normal	Default
Tx Assembly	Off	Off	Off	Off	Off	Off	Off	Off	On	Default
CCSDS Board Ass.	On	Active	On	Active	On	Active	On	Active	On	Active
TCS Subsystem	Off		Off		Default		Default		Default	
TCS Controller	Off	Off	Off	Off	Normal	Auto	Normal	Auto	Normal	Auto
Payload Subsystem	Off		Off		Off		Off		Off	
Payload Ctrl.	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
MICS Assembly	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
PAMCAM	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
OSIRIS Ass.	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
DDS Ass.	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
AIS	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
PLOC Ass.	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off

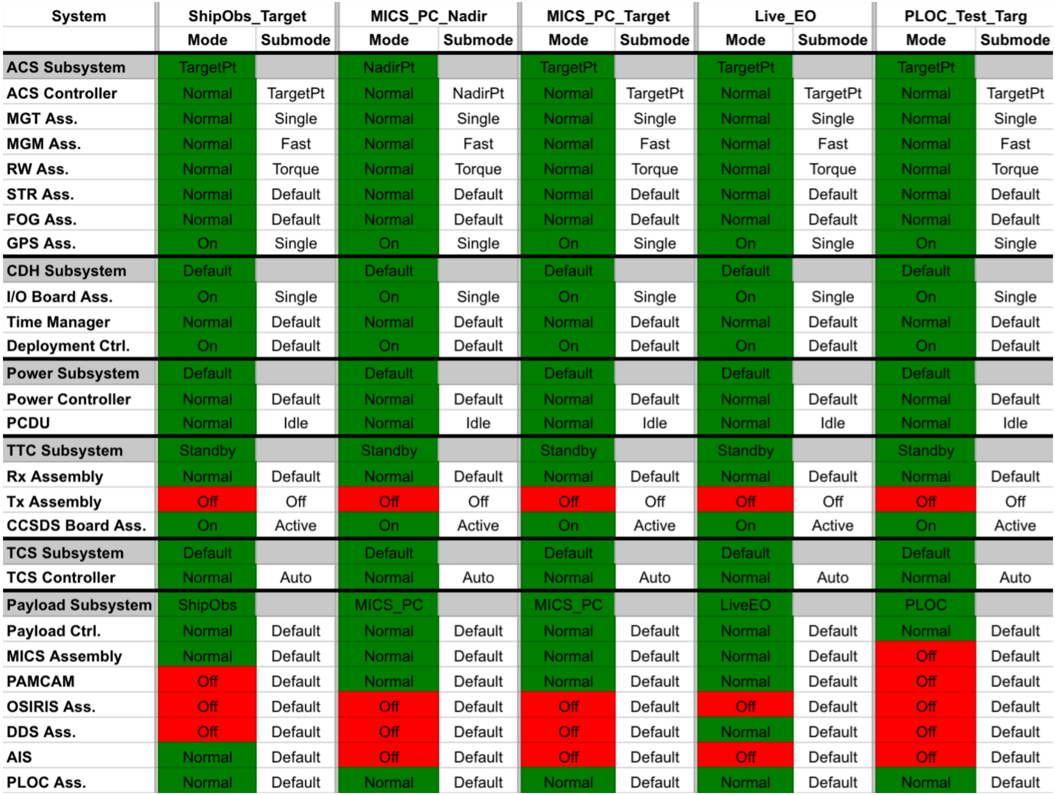
Table v: Flying Laptop System Mode Definitions (part 2/5)

System	GENIUS		MICS_PC_Inertial		MICS_EO_Nadir		PAMCAM_EO_Nad		PAMCAM_EO_Targ	
	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode
ACS Subsystem	GENIUS		InertialPt		NadirPt		NadirPt		TargetPt	
ACS Controller			Normal	InertialPt	Normal	NadirPt	Normal	NadirPt	Normal	TargetPt
MGT Ass.	Normal	Single	Normal	Single	Normal	Single	Normal	Single	Normal	Single
MGM Ass.	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast
RW Ass.	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque
STR Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
FOG Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
GPS Ass.	On	GENIUS	On	Single	On	Single	On	Single	On	Single
CDH Subsystem	Default		Default		Default		Default		Default	
I/O Board Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
Time Manager	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Deployment Ctrl.	On	Default	On	Default	On	Default	On	Default	On	Default
Power Subsystem	Default		Default		Default		Default		Default	
Power Controller	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
PCDU	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle
TTC Subsystem	Standby		Standby		Standby		Standby		Standby	
Rx Assembly	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Tx Assembly	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
CCSDS Board Ass.	On	Active	On	Active	On	Active	On	Active	On	Active
TCS Subsystem	Default		Default		Default		Default		Default	
TCS Controller	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto
Payload Subsystem	Off		MICS_PC		MICS		PAMCAM		PAMCAM	
Payload Ctrl.	Off	Off	Normal	Default	Normal	Default	Normal	Default	Normal	Default
MICS Assembly	Off	Off	Normal	Default	Normal	Default	Off	Default	Off	Default
PAMCAM	Off	Off	Normal	Default	Off	Default	Normal	Default	Normal	Default
OSIRIS Ass.	Off	Off	Off	Default	Off	Default	Off	Default	Off	Default
DDS Ass.	Off	Off	Off	Default	Off	Default	Off	Default	Off	Default
AIS	Off	Off	Off	Default	Off	Default	Off	Default	Off	Default
PLOC Ass.	Off	Off	Normal	Default	Normal	Default	Normal	Default	Normal	Default

Table vi: Flying Laptop System Mode Definitions (part 3/5)

System	OSIRIS_HPLD		OSIRIS_EDFA		DDS		AIS		ShipObs_Nadir	
	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode
ACS Subsystem	TargetPt		TargetPt		TargetPt		NadirPt		NadirPt	
ACS Controller	Normal	TargetPt	Normal	TargetPt	Normal	TargetPt	Normal	NadirPt	Normal	NadirPt
MGT Ass.	Normal	Single	Normal	Single	Normal	Single	Normal	Single	Normal	Single
MGM Ass.	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast
RW Ass.	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque
STR Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
FOG Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
GPS Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
CDH Subsystem	Default		Default		Default		Default		Default	
I/O Board Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
Time Manager	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Deployment Ctrl.	On	Default	On	Default	On	Default	On	Default	On	Default
Power Subsystem	Default		Default		Default		Default		Default	
Power Controller	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
PCDU	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle
TTC Subsystem	Standby		Standby		Standby		Standby		Standby	
Rx Assembly	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Tx Assembly	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
CCSDS Board Ass.	On	Active	On	Active	On	Active	On	Active	On	Active
TCS Subsystem	Default		Default		Default		Default		Default	
TCS Controller	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto
Payload Subsystem	OSIRIS_HPLD		OSIRIS_EDFA		DDSdown		AIS		ShipObs	
Payload Ctrl.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
MICS Assembly	Off	Default	Off	Default	Off	Default	Off	Default	Normal	Default
PAMCAM	Off	Default	Off	Default	Off	Default	Off	Default	Off	Default
OSIRIS Ass.	Normal	HPLD	Normal	EDFA	Off	Default	Off	Default	Off	Default
DDS Ass.	Off	Default	Off	Default	Normal	Default	Off	Default	Off	Default
AIS	Off	Default	Off	Default	Off	Default	Normal	Default	Normal	Default
PLOC Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default

Table vii: Flying Laptop System Mode Definitions (part 4/5)



System	ShipObs_Target		MICS_PC_Nadir		MICS_PC_Target		Live_EO		PLOC_Test_Targ	
	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode
ACS Subsystem	TargetPt		NadirPt		TargetPt		TargetPt		TargetPt	
ACS Controller	Normal	TargetPt	Normal	NadirPt	Normal	TargetPt	Normal	TargetPt	Normal	TargetPt
MGT Ass.	Normal	Single	Normal	Single	Normal	Single	Normal	Single	Normal	Single
MGM Ass.	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast
RW Ass.	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque
STR Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
FOG Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
GPS Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
CDH Subsystem	Default		Default		Default		Default		Default	
I/O Board Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
Time Manager	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Deployment Ctrl.	On	Default	On	Default	On	Default	On	Default	On	Default
Power Subsystem	Default		Default		Default		Default		Default	
Power Controller	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
PCDU	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle
TTC Subsystem	Standby		Standby		Standby		Standby		Standby	
Rx Assembly	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Tx Assembly	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
CCSDS Board Ass.	On	Active	On	Active	On	Active	On	Active	On	Active
TCS Subsystem	Default		Default		Default		Default		Default	
TCS Controller	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto
Payload Subsystem	ShipObs		MICS_PC		MICS_PC		LiveEO		PLOC	
Payload Ctrl.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
MICS Assembly	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Off	Default
PAMCAM	Off	Default	Normal	Default	Normal	Default	Normal	Default	Off	Default
OSIRIS Ass.	Off	Default	Off	Default	Off	Default	Off	Default	Off	Default
DDS Ass.	Off	Default	Off	Default	Off	Default	Normal	Default	Off	Default
AIS	Normal	Default	Off	Default	Off	Default	Off	Default	Off	Default
PLOC Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default

Table viii: Flying Laptop System Mode Definitions (part 5/5)

System	OSIRIS_LiveEO_HPLD		OSIRIS_LiveEO_EDFA		Rotation		InertialPt		NadirPt	
	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode	Mode	Submode
ACS Subsystem	TargetPt		TargetPt		Rotation		InertialPt		NadirPt	
ACS Controller	Normal	TargetPt	Normal	TargetPt	Normal	Rotation	Normal	InertialPt	Normal	NadirPt
MGT Ass.	Normal	Single	Normal	Single	Normal	Single	Normal	Single	Normal	Single
MGM Ass.	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast	Normal	Fast
RW Ass.	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque	Normal	Torque
STR Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
FOG Ass.	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
GPS Ass.	On	Single	On	Single	On	Default	On	Single	On	Single
CDH Subsystem	Default		Default		Default		Default		Default	
I/O Board Ass.	On	Single	On	Single	On	Single	On	Single	On	Single
Time Manager	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Deployment Ctrl.	On	Default	On	Default	On	Default	On	Default	On	Default
Power Subsystem	Default		Default		Default		Default		Default	
Power Controller	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
PCDU	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle	Normal	Idle
TTC Subsystem	Standby		Standby		Standby		Standby		Standby	
Rx Assembly	Normal	Default	Normal	Default	Normal	Default	Normal	Default	Normal	Default
Tx Assembly	Off	Off	Off	Off	Off	Off	Off	Off	Off	Off
CCSDS Board Ass.	On	Active	On	Active	On	Active	On	Active	On	Active
TCS Subsystem	Default		Default		Default		Default		Default	
TCS Controller	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto	Normal	Auto
Payload Subsystem	OSIR_EO1		OSIR_EO2		Off		Off		Off	
Payload Ctrl.	Normal	Default	Normal	Default	Off	Off	Off	Off	Off	Off
MICS Assembly	Normal	Default	Normal	Default	Off	Off	Off	Off	Off	Off
PAMCAM	Normal	Default	Normal	Default	Off	Off	Off	Off	Off	Off
OSIRIS Ass.	Normal	HPLD	Normal	EDFA	Off	Off	Off	Off	Off	Off
DDS Ass.	Off	Default	Off	Default	Off	Off	Off	Off	Off	Off
AIS	Off	Default	Off	Default	Off	Off	Off	Off	Off	Off
PLOC Ass.	Normal	Default	Normal	Default	Off	Off	Off	Off	Off	Off

APPENDIX D. Full Operations Tests

Table ix: Sequence of Events during the week-in-the-life operation test (days 1 and 2)

Day	Pass	G/S	Activity
1/5	1	IRS	• uplink time-tagged TCs for the upcoming 24h
	2	IRS	• switch to redundant Tx
	-	-	• heat MICS to operational temperature • Perform all MICS calibration (dark sky) while in eclipse • save data in MMU
	-	-	• heat MICS to operational temperature
	-	-	• perform target pointing Earth observation, • all MICS and PAMCAM, save data in MMU
2/5	3	NYA	• First GS Pass NYA
	-	-	• heat MICS to operational temperature
	-	-	• perform ship observation data take in Nadir pointing • all MICS, save data in MMU
	-	-	• Reset DDS R
	4	NYA	• downlink payload data using DDS R
	-	-	• perform GENIUS experiment in Nadir pointing
	-	-	• NEO Observation CM 1
	5	IRS	• reset battery capacity values • uplink time-tagged TCs for the upcoming 24h
	6	IRS	• perform OSIRIS EDFA data downlink from MMU with concurrent TM reception
	-	-	• heat PAMCAM to operational temperature
	-	-	• perform nadir pointing earth observation with PAMCAM, save data in MMU
	-	-	• Perform NEO observation CM2 with offset
	-	-	• Reset DDS R
	7	IRS	• uplink time-tagged TCs for the upcoming 24h • downlink payload data using DDS R
	8	IRS	• reset DOM Timer via HPC using VC1
	-	-	• set payload to external control
	9	NYA	• AIS Long Data Take

Table x: Sequence of Events during the week-in-the-life operation test (days 3, 4 and 5)

Day	Pass	G/S	Activity
3/5	10	NYA	• AIS Long Data Take
	-	-	• Reset Payload
	11	IRS	• uplink time-tagged TCs for the upcoming 24h • downlink payload data using DDS R
	12	IRS	• reboot OBSW
	13	IRS	• check satellite system after reboot (should be in safe mode, cooled down), • simulate contact outage due to safe mode • uplink time-tagged TCs for the upcoming passes
	14	IRS	• reset battery capacity values • load GPS Almanac and TLE • warm start of GPS • enable GPS F40 TM • monitor time update • set on-board time on FM
4/5	15	IRS	• check OBSW image for corruption • switch commanding to VC4 • uplink commands for upcoming passes
	16	IRS	• manipulate STR for attitude solution • command system to idle mode and monitor • upload time-tagged TCs for STR image
		-	• Take image with STR and save it in the TM Store
	17	IRS	• check spacecraft state • downlink STR image • uplink commands for next 24h
	18	IRS	• use HPC to reset DOM timer on VC3 • erase TC schedule and resend TCs for upcoming 24h
	-	-	• Erase MMU
	-	-	• Perform MICS calibration (moon) while in eclipse, save data in MMU
5/5	-	-	• Perform EO in Nadir pointing, all MICS + PAMCAM
	-	-	• Perform EO in pattern in target pointing, single MICS
	19	IRS	• reset DDS N speed (2.8), upload TCs for upcoming 24h
	20	IRS	• downlink payload data using DDS R
	21	IRS	• perform live earth observation with DDS R
	22	IRS	• perform OSIRIS HPLD pattern data downlink

APPENDIX E. AFT Modules and Test Scripts

Table xi: List of Abbreviated Function Test Modules for Flying Laptop

Module	Test Equipment Target of Module
Core AFT module	• OBC core nominal, OBC core redundant • CCSDS board 0, CCSDS board 1 • I/O board nominal, I/O board redundant • TX nominal ,TX redundant • RX 0, RX 1 • PCDU CPU-A, PCDU CPU-B
Reaction wheel AFT module	• RW0, RW 1, RW 2, RW3
GPS assembly AFT module	• GPS 0, GPS 1, GPS 3
Star tracker DPU AFT module	• STR DPU
Star tracker assembly AFT module	• STR CHU A, STR CHU B
FOG assembly AFT module	• FOG unit 0, FOG unit 1, FOG unit 2, FOG unit 4
MGM assembly AFT module	• MGM 0, MGM 1
MGT assembly AFT module	• MGT 0, MGT 1 • nominal MGT coil x, MGT coil y, MGT coil z • redundant MGT coil x, MGT coil y, MGT coil z
DDS/OSIRIS/PLOC AFT module	• DDS 0, DDS 1 • OSIRIS HPLA • OSIRIS EDFA • PLOC N, PLOC R
MICS AFT module	• MICS NIR, MICS Green, MICS Red
MICS Baffle AFT module	• MICS calibration LED current check
PAMCAM AFT module	• PAMCAM
AIS AFT module	• AIS receiver
Power/Thermal AFT module	• Temp. sensor plausibility • Fuse current sensor plausibility • Solar panel, battery, power bus sensor plausibility