Embedded Systems Development for SFL Satellites

by

Jakob Lifshits

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Aerospace Science and Engineering
University of Toronto

Copyright © 2010 by Jakob Lifshits
Abstract

Embedded Systems Development for SFL Satellites

Jakob Lifshits

Master of Applied Science

Graduate Department of Aerospace Science and Engineering

University of Toronto

2010

The work described in this thesis summarizes the author’s contributions to the design, development, and testing of embedded systems for SFL spacecraft. The unique environment of space and the constraints it imposes on embedded systems is described, and the testing methodologies employed to qualify spacecraft hardware for this environment are presented. The implementation of an automated functional test framework for SFL’s Generic Nanosatellite Bus satellites and the role it plays in the spacecraft development cycle is also discussed. Application software and device drivers in support of the BRITE mission were designed and developed. Finally, a controller was implemented for thermal control of the BRITE optical instrument. These contributions play an important role in the continual efforts to expand access to space and to prove the feasibility of the microspace approach in doing so.
Acknowledgements

I would like to begin by thanking my supervisor, Dr. Robert E. Zee, for providing me with
the amazing opportunity of being part of the SFL team and for his continual mentorship
throughout the past two years. SFL is truly a unique place, and I am honoured to
have participated in its endeavours. I must also thank Cordell Grant for entrusting me
with challenging assignments, and allowing me to work with flight hardware. The entire
SFL team has always been extremely helpful, but in particular I would like to thank
Mihail Barbu, who’s wealth of knowledge is unmatched by anyone I have met before,
and who was always full of lessons and advice, and Daniel Kekez who has always made
himself available to answer my never-ending questions. I am indebted to my parents and
grandparents for their unconditional support and encouragement in all of my ventures,
and for the values they bestowed upon me. I am also grateful to my sister and brother-
in-law for making the transition to a new city so pleasant. I cannot thank enough all
of my friends who contributed to making the past few years an amazing experience and
helped me maintain my morale, those who accompanied me on my outdoor adventures,
and those who provided a reliable belay while I was taking out my frustrations on the
rock.
# Contents

## 1 Introduction

1.1 Background ............................................. 1
  1.1.1 SFL and The Microspace Philosophy .............. 1
  1.1.2 SFL Spacecraft .................................... 2
  1.2 Embedded Systems on Spacecraft ..................... 7

## 2 Environmental Hardware Testing

2.1 Thermal Shock Test ................................. 11
2.2 Thermal Functional Test ............................. 12
2.3 Other Environmental Tests ......................... 13
2.4 Contributions and Lessons Learned ................. 14

## 3 Functional Hardware Testing

3.1 Requirements ........................................ 17
  3.1.1 Automation ....................................... 18
  3.1.2 Result Reporting ................................. 18
  3.1.3 Modularity ....................................... 19
  3.1.4 Ease of Setup .................................... 19
  3.1.5 Interface ........................................ 19
  3.2 Design Overview .................................... 19
  3.3 Detailed Design ..................................... 21
    3.3.1 Scheduler Module ............................... 21
    3.3.2 Communications Module ......................... 22
    3.3.3 Test Configuration .............................. 24
    3.3.4 XML Parsing Module ............................. 25
    3.3.5 Test Interface .................................. 27
3.3.6 Logging Module .............................. 28
3.3.7 User Interface .............................. 28
3.4 Deployment, Use, and Results .................. 30

4 Canadian Advanced Nanospace Operating Environment .................. 31
4.1 The Scheduler ................................... 31
4.1.1 Original Scheduling Scheme .................... 32
4.1.2 Revised Scheduling Scheme ...................... 33

5 BRITE Instrument Software ......................... 38
5.1 Overview of the BRITE Instrument .................. 39
5.1.1 Telescope ...................................... 39
5.1.2 Header Board and CCD Detector ................. 39
5.1.3 Instrument Computer ......................... 39
5.2 Instrument Computer Bootloader ................. 41
5.3 Instrument Computer Application Software ......... 42
5.3.1 Mission-Level Design and Operations .......... 42
5.3.2 Subsystem-Level Design and Implementation .... 44
5.4 CCD Header Board Software ..................... 61
5.4.1 Requirements .................................. 62
5.4.2 Architecture and Resource Allocation .......... 62
5.4.3 Controller ..................................... 64
5.4.4 Performance and Results ....................... 65

6 Star Tracker Driver ................................... 68
6.1 Star Tracker Interface ......................... 69
6.1.1 Hardware Interface ............................ 69
6.1.2 Software Interface ............................ 69
6.2 Driver Requirements ............................. 71
6.3 Driver Architecture and Detailed Design ............ 71
6.4 Driver Testing .................................. 73
6.4.1 Stage One .................................... 74
6.4.2 Stage Two .................................... 74
6.4.3 Stage Three .................................. 75
6.5 Progress and Future Work ....................... 75
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>LFFT requirements</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>CANOE thread states</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>BRITE instrument computer software requirements</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>BRITE instrument computer device drivers</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Non blocking receive call state machine</td>
<td>57</td>
</tr>
<tr>
<td>5.4</td>
<td>CCD header board requirements</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Star tracker driver requirements</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Figures

1.1 CanX-2 ................................................................. 3
1.2 Generic Nanosatellite Bus (GNB) ............................... 4
1.3 AISSat-1 .............................................................. 6
1.4 BRITE ................................................................. 7
2.1 Thermal shock test profile ........................................ 12
2.2 Thermal functional test profile ................................... 14
3.1 LFFT top level block diagram .................................... 22
3.2 LFFT Test Plan ....................................................... 23
3.3 LFFT global configuration file .................................... 26
3.4 LFFT test specific configuration file ......................... 26
3.5 LFFT XML parsing routine ....................................... 27
3.6 LFFT test report ...................................................... 29
4.1 Four cycles of CANOE scheduler in round-robin scheme .... 34
4.2 Four cycles of CANOE scheduler in priority round-robin scheme 37
5.1 BRITE telescope .................................................... 40
5.2 BRITE CCD header board ........................................ 40
5.3 BRITE instrument computer ...................................... 41
5.4 BRITE mission top level operations ............................ 43
5.5 BRITE mission on-board observation sequence diagram ... 46
5.6 BRITE IOBC manual state transition diagram ................. 49
5.7 BRITE IOBC automatic state transition diagram ............. 50
5.8 BRITE IOBC automatic delayed state transition diagram ... 51
5.9 UART/DMA buffering scheme .................................... 56
5.10 Non blocking receive call ........................................ 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.11</td>
<td>BRITE IOBC time synchronization state transition diagram</td>
<td>60</td>
</tr>
<tr>
<td>5.12</td>
<td>Header board control topology</td>
<td>64</td>
</tr>
<tr>
<td>5.13</td>
<td>Discrete PID controller</td>
<td>66</td>
</tr>
<tr>
<td>5.14</td>
<td>Header board controller performance</td>
<td>67</td>
</tr>
<tr>
<td>6.1</td>
<td>Star tracker sample output</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>Star tracker driver integrated into CANOE</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Star tracker driver under Windows</td>
<td>75</td>
</tr>
<tr>
<td>6.4</td>
<td>Star tracker driver under windows and OBC pass-through</td>
<td>76</td>
</tr>
<tr>
<td>6.5</td>
<td>Star tracker driver on OBC</td>
<td>76</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

1.1.1 SFL and The Microspace Philosophy

Traditionally, spacecraft development has been a lengthy and expensive endeavour reserved for government agencies and large corporations. The inaccessibility of space has been severely limiting the frequency and diversity of space missions, both commercial and scientific in nature. A new way of thinking about spacecraft development, dubbed the microspace philosophy, was developed as an alternative to the traditional approach with the goal of reducing the time frame and costs associated with spacecraft development and operations. The University of Toronto Institute for Aerospace Studies / Space Flight Laboratory (UTIAS/SFL) has embraced this principle, and has demonstrated its viability through the development, launch, and operation of a number of very capable and successful missions.

The reduced costs and development times touted by the microspace approach are achievable through a number of deviations from the traditional thought process. Primarily, a bottom-up approach is undertaken, whereby only missions that can be achieved through this method are selected, rather than the top-down approach in which the technology is tailored to the mission, often at great expense. Another major difference relates to the selection of electronic components. Traditionally, due to the harsh environment of space, components are specially designed and fabricated to be tolerant to radiation, vacuum, and extreme thermal conditions. Instead, the microspace approach favours the use of off-the-shelf components in clever ways which do not compromise the reliability of
the spacecraft. Further benefits are achieved by focusing on miniaturization of spacecraft technology. Nanosatellites (weighing under 10 kg) and microsatellites (weighing under 100 kg) can often achieve objectives similar to those of full-sized spacecraft, but at a fraction of the development and launch costs.

Besides being a research, development, and operations centre for satellites, SFL is also an academic institution. With guidance from experienced full-time staff, graduate students at SFL are training in this unique and exciting discipline and gain practical hands-on experience with every aspect of the spacecraft development and commissioning cycle.

1.1.2 SFL Spacecraft

Since its establishment in 1998, SFL has developed and launched a number of spacecraft. Along with building complete satellites, SFL also provides single space hardware components and complete subsystems to industry and government partners. Below is a brief description of satellites which SFL has developed in full or significantly contributed to.

MOST

MOST, which stands for the Microvariability and Oscillations of Stars, is Canada’s first space telescope. Launched in 2003, it is now in its seventh year of in-orbit operations providing highly useful scientific data and proving that reliable and useful spacecraft can be built using off-the-shelf components and within the size and mass constraints of microsatellites. MOST’s objective is to measure minute variations in the brightness of nearby stars, from which the star’s structure and chemical composition can be deduced. The ages of stars and the universe can be determined this way. From its 820 km orbit, MOST is not hindered by the atmosphere as are ground based telescopes [1][2].

CanX-1

The first in a series of the Canadian Advanced Nanospace Experiment (CanX) missions, CanX-1 was Canada’s first nanosatellite. It is a great example in spacecraft miniaturization, weighing under 1 kg and having dimensions of 10 cm on each side. Built by graduate students at UTIAS, CanX-1 launched alongside MOST in 2003. It carried a number of experimental payloads such as a horizon sensor, star tracker, GPS-based position determination system, and magnetic attitude stabilization system [3].
CanX-2 is the second nanosatellite developed by SFL. Weighing a mere 3.5 kg, CanX-2 carries an impressive assortment of experimental technologies and scientific payloads. As part of a technology demonstration campaign, CanX-2 carries a GPS receiver, a miniature cold-gas propulsion system, and a reaction wheel. These components are evaluated for their use on the upcoming CanX-4/CanX-5 formation flying mission. A number of scientific experiments, including an atmospheric spectrometer and a space materials experiment are being conducted on CanX-2 through partnerships with other Canadian universities. CanX-2 was successfully launched in 2008 [4]. A solid model and photograph of CanX-2 are shown in Figure 1.1.

NTS

The Nanosatellite Tracking Ships (NTS), also known as CanX-6, was designed to demonstrate Automatic Identification System (AIS) capabilities from space. Consisting of a satellite bus designed by SFL and a payload provided by COM DEV, NTS is capable of mapping maritime vessel traffic and relaying this information to the ground station.
Chapter 1. Introduction

This 6.5 kg satellite combines the electronics of CanX-2 with a mechanical structure of the Generic Nanosatellite Bus. NTS was designed and built within seven months and launched alongside CanX-2. At the time of writing, it has outlived its design mission life by a factor of four.

Generic Nanosatellite Bus (GNB)

One of the largest cost drivers in mission and spacecraft design is the amount of non-recurring engineering (NRE) that must be conducted. By utilizing a common satellite bus across many missions, SFL is able to significantly cut the amount of NRE while leveraging the space heritage obtained from previous missions. The Generic Nanosatellite Bus (GNB) was developed to reuse common subsystems such as on-board computers, attitude determination and control, power electronics, communications, thermal control, and structure, while maintaining adaptability for mission specific payloads and constraints. Drawing upon mature and tested components not only decreases design time and costs, but also guarantees a certain level of functionality and reliability. The GNB structure with all essential and some optional components is shown in Figure 1.2.
AISSat-1

AISSat-1 is the first satellite developed by SFL that is fully based on the Generic Nanosatellite Bus. As such, it has three-axis attitude determination and control capabilities. Developed for the Norwegian government, AISSat-1’s purpose, similarly to NTS, is to demonstrate the feasibility of detecting AIS signals from space [7]. A model of AISSat-1, along with a photograph of the built spacecraft with SFL staff is are shown in Figure 1.3.

BRITE

The Bright Target Explorer (BRITE) satellites, also known as CanX-3A through CanX-3F, are a stellar photometry mission designed to observe the variability in brightness of stars. While similar in concept to MOST, BRITE is designed to observe different targets, and to detect oscillations of much higher frequency. The precision of these measurements can be as much as 10 times higher than what is achievable from earth due to the absence of the atmosphere. By leveraging the low cost of each nanosatellite, a constellation of six spacecraft is planned to maximize science data generation [8]. Figure 1.4 shows an exploded view of the BRITE satellite.

CanX-4 and CanX-5

CanX-4 and CanX-5 are twin nanosatellites designed for a formation flight demonstration missions. This mission brings together a number of the aspects evaluated on CanX-2 into a GNB framework. It combines a three-axis attitude control system, a miniature cold-gas propulsion, and GPS receivers for relative position determination [9]. Novel control algorithms developed in house, and high accuracy relative positioning algorithms developed in collaboration with the University of Calgary are being employed to accomplish the mission objectives. Demonstrating formation flight on nanosatellites is a crucial step in opening up numerous possibilities for new mission concepts.

M3MSat

The Maritime Monitoring and Messaging Microsatellite (M3MSat) is a technology demonstration mission funded by the Canadian Space Agency and Defence Research and Development Canada. Developed by COM DEV International Ltd., it will use command
Figure 1.3: AISSat-1

(a) Solid model

(b) AISSat-1 in the DFL clean room

Figure 1.3: AISSat-1
and data handling as well as power subsystems provided by SFL. It will carry an AIS transponder that will expand on the technology demonstrated on NTS, as well as a secondary communications payload for civil and commercial use [10].

**NEMO-AM**

NEMO-AM, or Nanosatellite for Earth Monitoring and Observation - Aerosol Monitor, is a mission designed to observe light reflected from aerosol gases in the Earth atmosphere. Containing an optical instrument, it will observe light in the visual band and will be capable of operating in various modes of resolution up to 40 m ground-sample-distance. With a high speed data link, as much as 200 MB of raw data is expected to be downloaded and processed daily. The NEMO-AM structure is based on two GNB cubes, and reuses many subsystems from current spacecraft [11].

### 1.2 Embedded Systems on Spacecraft

Embedded systems are an integral part of any modern spacecraft, and are ubiquitous to almost every satellite subsystem, including but not limited to command and data
Chapter 1. Introduction

handling, attitude determination and control, communications, power monitoring and distribution, active thermal control, and the payload. Embedded systems can range from high performance processors, to minimalistic microcontrollers, or even systems-on-a-chip (SOC). The use of low-cost embedded controllers and software can dramatically simplify electronic designs by eliminating the equivalent analog or digital circuitry. It is generally accepted that software is easier and cheaper to develop and debug when compared to hardware. On satellites, some software modules can be upgraded and replaced after launch, a luxury that does not extend to hardware components.

The embedded systems found on spacecraft differ significantly from those we encounter in our daily lives in consumer electronics such as cellular phones and appliances. These differences are mostly driven by perhaps the most significant design criterion for space borne systems, which is reliability. Exceptionally high reliability in the design, components, and fabrication process is necessary as failures after launch can often result in the loss of a mission. On larger spacecraft, whole subsystems are often backed up with spare redundant systems, and designs tend to be conservative with large de-rating values applied to components. These methods however must be traded against the increases in mass, volume, and power consumptions that result, and thus are not always appropriate for smaller satellites.

The harsh environment in which spacecraft operate imposes additional constraints on the design. Radiation, which is trapped by the magnetosphere before reaching Earth, can quickly damage electronics in space. The traditional solution to this problem is the use of radiation hardened electronics; however these are expensive and require long lead times. Instead, SFL screens off-the-shelf components for their performance under radiation. The selection of components that perform sufficiently is typically very limited, further challenging the embedded systems designer. Electronic components must also be able to withstand the variations in temperature that the spacecraft may see in space. These variations may not only push the components to their extreme operating temperature, but the rapid changes in temperature may also cause mechanical stresses due to contraction and expansion of materials.

One of the toughest challenges the embedded systems engineer faces is the limited availability of power on board the spacecraft. Small satellites such as the ones produced by SFL possess very limited power generation capacity due to the lack of space and weight to accommodate solar panel arrays. Embedded systems must be designed to consume a minimal amount of power, a requirement which imposes direct consequences on all other
subsystems. Power limitations often impact the computational capacity of on-board computers, the bandwidth of communication links, the control authority of attitude actuators, and many other aspects of the spacecraft. Trade studies in requirements are performed to achieve a balanced design. As an example, a power limited communications link might require that more of the payload data is processed on-board before being downloaded to the ground. This in turn drives the requirements of the on-board computer subsystems, which would now require more computational capacity, and more power. Trade studies such as these are very common in spacecraft design due to the unusually high coupling between subsystems.

It can be seen that embedded systems design for spacecraft is a unique discipline that requires many considerations and is limited by the numerous constraints that arise from the equally unique environment in which these systems operate. Factors such as power consumption, reliability, mass, size, computational capacity, storage capacity, and communications capacity are all very tightly coupled and must be traded against each other to achieve a balanced design. Throughout the past two years the author has worked on a number of projects that required embedded systems design, implementation, and validation. These projects touched on both flight systems that form a part of the spacecraft, as well as ground support systems that assist in the operations of the spacecraft before and after launch. The purpose of this thesis is to provide a detailed account of this work, and how it benefited SFL’s spacecraft. This document is structured into a number of chapters, with each chapter focusing on one particular project. The first section of the thesis deals with verification of hardware. Chapter two focuses on environmental testing of electronic components, while chapter three discusses the development of an automated framework for functional testing. The second section of this thesis focuses on the development of embedded software for satellites. Chapter four describes an enhancement of SFL’s proprietary operating system. Chapters five and six talk about software development for the BRITE stellar photometry mission. The final chapter summarizes and concludes this work. Collectively, these efforts directly contribute to the success of a number of missions through increasing reliability by performing testing on hardware, and through the addition of required functionality with software development.
Chapter 2

Environmental Hardware Testing

The unique environment of space brings about a number of challenges that the embedded systems engineer must address. The extreme thermal conditions, radiation, and vacuum may have adverse effects on spacecraft hardware, and if not properly managed could lead to catastrophic failures. Systems must be designed with this environment in mind and extensive testing must be performed to replicate the space environment as closely as possible on the ground. SFL performs a suite of environmental tests throughout the development cycle of spacecraft and space borne components.

Each test is performed on two levels: qualification, and acceptance. The qualification test is intended to qualify the design, that is to ensure that the design is capable of operating and meeting its requirements in the given operating environment. This test is typically very extensive, and the environmental parameters are pushed outside of the expected envelope to ensure that sufficient margin exists in the design. Qualification testing is performed only once for every design iteration. Acceptance testing is intended to verify the functionality of every unit that stems from a particular qualified design. This test tends to be higher level than qualification testing, and the environmental parameters are set to match those expected in orbit. Acceptance testing must be performed on every assembled unit as it is used to identify flaws in the assembly process or defects in components. This chapter describes the environmental testing efforts in which the author has participated and how this work benefited SFL’s spacecraft.
2.1 Thermal Shock Test

Thermal shock refers to the rapid change of temperature as experienced by the components and structure of a satellite. There are two main causes to this phenomenon in orbit. Primarily, it can be observed as the satellite orbits the earth and transitions from the sun pointing side of a planet into eclipse, and back. In the worst case scenario of a noon-midnight sun-synchronous orbit of low altitude, these transitions happen very rapidly. A secondary effect occurs as the satellite’s attitude is changing in orbit. As one side of the spacecraft points towards the sun and heats up, the opposite side may be pointed towards open space which causes it to cool down. As the satellite rotates the hot side begins to cool down while the cold side heats up. These temperature changes cause components on the spacecraft to contract and expand applying significant mechanical stress. Delicate components and poor solder joints are particularly susceptible. The packaging of integrated electronic devices can also fail if the shock is extreme or there is a defect in the fabrication process \[12\]. As such this test is primarily intended to evaluate the workmanship of assembly, and secondarily to identify components that may not be capable of withstanding the mechanical stresses seen on orbit.

The thermal shock test is conducted with the use of two thermal chambers, one of which is set to the maximum design survival temperature of the component, and the other to the minimum survival temperature. The component is placed in the cold chamber until its temperature significantly drops, and is then rapidly transferred into the hot chamber where it sees rapid increases in temperature. When sufficiently hot, it is returned into the cold chamber. This constitutes a single cycle of the thermal shock test. The full test consists of at least 25 cycles \[13\], and emphasis is placed on the rates of change of temperature rather than the absolute temperature reached. A minimum rate of change of 25 deg°C/min is required to achieve the desired mechanical stress. As the test is intended to identify mechanical faults only, the component-under-test is powered off. Faults are identified by performing a visual inspection and a functional test before and after the thermal shock test. The purpose of the visual inspection is to detect faulty solder joints and obvious damage to the exterior of electronic devices, while the functional test involves powering on the device and testing its functionality by running diagnostics software. The post thermal shock functional test is run over a varying temperature range, as described in the following section. It is meant to detect faults not caught through the visual inspection or those that occurred internally within
electronic device packaging. A sample thermal shock temperature profile is shown in Figure 2.1. The board temperature plots represent readouts from sensors attached to the component-under-test, and the ambient temperature refers to the temperature of the chamber.

### 2.2 Thermal Functional Test

While the thermal shock test represents an accelerated stress test, the thermal functional test is closer to a real-time orbital simulation. As part of qualification testing, it is intended to ensure that the design is tolerant to variations in temperature. In electronics especially, there are numerous temperature dependent variables. Resistivity, capacitances, break-down voltage of dielectrics and leakage current through them, as well as the current-voltage characteristics of p-n junction devices are a small subset of these [14]. A marginal design may perform perfectly under room temperature conditions, but would fail under extreme hot or cold scenarios. As part of an acceptance procedure, the thermal functional test is geared towards identifying poor workmanship. A cold solder joint may provide sufficient contact for current to flow at room temperature, but
the mechanical movements induced by changes in temperature could cause the joint to disconnect. Upon returning to room temperature the solder joint would return to its place, and this scenario would not be identified during a post thermal shock functional test at room temperature.

The thermal functional test is conducted with the use of a single thermal chamber. Four temperatures are defined: $T_{\text{hot, survival}}$ and $T_{\text{cold, survival}}$ are the absolute hottest and coldest temperatures that the device is designed to withstand while in the powered-off state. $T_{\text{hot}}$ and $T_{\text{cold}}$ are the hottest and coldest temperatures that the device is expected to operate in. An initial functional test is performed at room temperature to verify the device’s full functionality before the thermal test commences. With the device powered off, it is then heated to $T_{\text{hot, survival}}$, where it is soaked for some duration of time. After being cooled down to $T_{\text{hot}}$ the device is powered on and a full functional test is performed. This is known as the warm start functional test. With the power off the device is then cooled down and soaked at $T_{\text{cold, survival}}$. Its temperature is then elevated to $T_{\text{cold}}$ and another functional test is executed. This is the cold start functional test. Without turning the device off it is cycled a number of times from $T_{\text{cold}}$ through $T_{\text{hot}}$ with a short-form functional test executing periodically, and a long-form functional test executed at each plateau. At the end of these cycles a final long-form room temperature functional test is performed [15]. A thermal profile of this is presented in Figure 2.2.

### 2.3 Other Environmental Tests

There are a number of other environmental tests that the author did not directly participate in, but are briefly described here for completeness. Electronic components must undergo radiation screening as part of the selection procedure. This test is performed at the TRIUMF facility where the component is placed under a Proton radiation beam and its operation is observed for single event upsets (SEU) and single event latch-ups (SEL). Another environmental test is the thermal vacuum test, which is very similar in procedure to the thermal functional test, with the difference that it is performed under vacuum, and only as part of the qualification procedure. Finally, a vibration test is performed to verify that the spacecraft can withstand the vibration loads seen during transportation and launch.
Figure 2.2: Thermal functional test profile

2.4 Contributions and Lessons Learned

The author’s contributions to the testing efforts include the setup of the test environment, construction of the required test support equipment, preparation of the test article, execution of the test, and analysis of the test results. Test environment set-up consisted of configuring the thermal chambers to the desired profiles, with special care being taken to ensure that no components were pushed beyond their allowable thermal limits. Test support equipment was fabricated for each test, and had to be customized to the test article as well as the test environment. This typically included a mounting plate to which the test article was attached, as well as a wiring harness which provided power to the test article and allowed commands and telemetry to be sent and received. With the test article mounted to the plate, the wiring harness was connected and the item was sealed in an electrostatic-safe bag. Desiccant was added to absorb any moisture from the air surrounding the test article. The test was executed as per the procedures outlined in the above subsections, and results were collected, analyzed, and logged for future reference.

A number of lessons were learned while performing these tests, which could be applied to improve the testing process in the future. Firstly, it was noted that the desiccant was...
insufficient in providing a dry environment, and condensation was occasionally observed around the test article. If left unremedied, it could cause a short circuit on the test article electronics. It is recommended that future thermal functional tests are performed in thermal chambers with a dry-air purge system. Such a system provides humidity control of the air inside the chamber, and guarantees that condensation cannot form. Secondly, a significant amount of time was required to perform the thermal functional test, which consists of repeating the same set of test cases over the full thermal profile. It is recommended that automation is used wherever possible. Automation can be applied to the test software, as well as to the control of the chamber, a task currently performed manually. The following chapter describes the development of an automated test framework. While originally developed for system-level testing, it was later adapted to assist in unit-level thermal functional testing as well.
Chapter 3

Functional Hardware Testing

Functional testing is the process of verification of the behaviour of a component or a system of components when exposed to certain stimuli and environmental conditions. The functionality of a component or system can be verified with varying levels of depth. As an example, let us consider the on-board computer of a spacecraft. At the lowest level, one can verify the voltage levels at various circuit nodes, as well as the rise and fall times of signal edges on data buses. At a medium level individual peripherals such as the random access memory, flash, analog-to-digital converters, communications devices, etc. can be verified. At the highest level, the on-board computer hardware must be operated in conjunction with the final flight software as a single entity, in a manner which is representative of flight operations. Functional testing can also be performed on the unit level, such as testing the on-board computer individually, or at a system level, such as testing the on-board computer integrated into the spacecraft. In the latter case, the inter-component interaction is also verified.

The scope of a functional test greatly depends on the phase of the development cycle during which it is executed. During the design and prototyping phase of a component, the test tends to be very low level and very thorough. At this stage the designer intends to verify that the theoretical design is practical and implementable. This type of testing is typically performed early in the development cycle and is not repeated afterwards, similarly to the qualification testing discussed in the previous chapter. Acceptance testing on the other hand is performed for every unit that is built from an already qualified design. At this stage the tests is a measure of workmanship and component quality, intending to identify defects in parts and manufacturing rather than flaws in the design. Acceptance testing typically relies on verification of medium and high level functionality to identify
faults, and is often performed in conjunction with environmental tests such as thermal or vibration testing.

Functional testing is also performed to varying degrees on the spacecraft as a whole. The comprehensive performance test, also referred to as the long-form functional test (LFFT), is a comprehensive test of all of the spacecraft units connected in a flight configuration, as well as a test of the flight hardware with the flight software in a flight-representative manner. A subset of the LFFT can also be performed, and is known as the limited performance test, or the short-form functional test (SFFT). Spacecraft functional testing is performed beginning with the integration stage, and up to the commissioning stage of the spacecraft. When integrating a new component to the satellite, a short-form functional test is immediately executed to verify its connectivity, and effects on other components already in the spacecraft. Once fully assembled, long-form functional testing is performed in conjunction with the environmental testing that the spacecraft must undergo. After transportation and attachment of the spacecraft to the launch vehicle, long-form functional testing can be used as a final measure to verify that all connections are still intact. It can also be used as an initial checkout of spacecraft systems after launch.

As can be seen, functional level testing is performed frequently during the development and commissioning phases of a spacecraft. With multiple satellites being developed simultaneously, the effort required to perform these tests can be quite significant, consuming many man-hours. As such, it was desired to create an automated test application that would leverage the fact that GNB based missions share a large number of the same components and use the same protocols for communication. By automating the testing process, its consistency can be guaranteed to a higher degree, further improving the validity of the results. This section describes the requirement gathering, design, development, and deployment phases that the author has undertaken to produce an automated functional test framework. This automated test software focuses on the hardware portion of the long-form functional test, and is called Phase One of the LFFT.

### 3.1 Requirements

Before the design and implementation of the LFFT could commence it was necessary to formally define its requirements and functionality. With input from the SFL team, a requirements document was created [16], as summarized in Table 3.1. A detailed
Table 3.1: LFFT requirements [16]

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFFT.1</td>
<td>The GNB LFFT should be as automated as is reasonable to achieve.</td>
</tr>
<tr>
<td>LFFT.2</td>
<td>The GNB LFFT should be self-documenting as much as possible.</td>
</tr>
<tr>
<td>LFFT.3</td>
<td>The GNB LFFT software should make use of existing test software where feasible.</td>
</tr>
<tr>
<td>LFFT.4</td>
<td>The GNB LFFT software should be modular and easily modified for different satellite configurations and missions.</td>
</tr>
<tr>
<td>LFFT.5</td>
<td>The GNB LFFT software should be user friendly and require minimal setup.</td>
</tr>
<tr>
<td>LFFT.6</td>
<td>The GNB LFFT software should include an option to run a Short-Form Functional Test (SFFT).</td>
</tr>
<tr>
<td>LFFT.7</td>
<td>The LFFT and SFFT shall run over both the test port and radio links and should communicate via NSP.</td>
</tr>
<tr>
<td>LFFT.8</td>
<td>The LFFT software should have PEEK and POKE capabilities.</td>
</tr>
</tbody>
</table>

A description of each requirement and the motivation behind them is provided below.

### 3.1.1 Automation

One of the primary motivations for this project is to minimize the amount of operator intervention required to perform a test, allowing for more tests to be executed and generally increasing SFL’s productivity. One way to achieve this goal without compromising the integrity and depth of the test is through automation. By eliminating the need for user intervention wherever possible, the test can be performed significantly faster, while still maintaining its full scope, and allowing the operator to dedicate their attention to other tasks. There are however, tests which cannot be automated to the full extent. These tests may require operations such as shining a light onto a solar panel to measure its output, or tilting the spacecraft sideways to measure the change reported by the magnetometer. As such, this requirement states that the test software should be “as automated as is reasonable to achieve” [16].

### 3.1.2 Result Reporting

While manual execution of the functional test is a time consuming process, the logging of test results is equally so. A key requirement for the automated test software is to provide a test report that is as complete as possible, in a format that is easy to understand and refer to at a future date. This report should contain information about the test setup, and
be sufficient to reproduce the test. In addition to efficiency, automated report generation provides consistency. After hundreds of test logs are generated, it is significantly easier to parse through the data if every report is identical in formatting.

### 3.1.3 Modularity

As previously stated, a functional test can be run on an individual unit, on a partially integrated system, or a fully integrated system. The automated test software should be modular and configurable to accommodate any of these cases. Furthermore, at any given time there may be a number of satellites at different stages of development, each with a different complement of sensor, actuators, and payloads. The test software should be able to easily accommodate this scenario as well.

### 3.1.4 Ease of Setup

The previous requirement states that the test software should be adaptable to various configurations and scenarios. With this requirement we try to ensure that it is easy for an operator to take advantage of this adaptability by providing a simple and understandable configuration interface. A trade off must be made, as typically the more modular and adaptable a system is, the more time it takes to configure it to any particular scenario.

### 3.1.5 Interface

Spacecraft designed around the Generic Nanosatellite Bus can be communicated with either through a radio link, or via a serial port designed for debugging, called the test port. It was decided that the test software should be able to communicate with a satellite by either of these means. Communication over radio is required to simulate a flight like configuration, and to test each unit in the communications subsystem. Communications by test port are required for unit level or partial system level testing in which radios have not yet been integrated.

### 3.2 Design Overview

To ensure that the final product met the design goals and requirements to a satisfactory degree, an iterative design process was applied. Firstly, an architecture of the test frame-
work was created and circulated amongst the SFL team, addressing each requirement and how they will be met at the top level. After collecting feedback and creating a number of revisions, a final architecture document was produced \[17\].

One of the main challenges encountered with the design of the LFFT was determining how to achieve the required level of accessibility to the various components of the spacecraft. On the unit level, there typically exists a test wiring harness and ground support equipment (GSE) that allow a component to be connected directly to a personal computer and controlled from it. This approach is complicated to set up, directly violating the ease-of-setup requirement. Furthermore, the use of test GSE does not represent a flight-like configuration, and is not usable in a system level test where the component would be connected to other systems in the spacecraft. Since in a flight configuration all electronic units of the spacecraft are connected directly to one of three on-board computers (OBC), it was decided that the same setup would be used while testing. Communications with the test component can be accomplished through the OBC, and communications with the OBC can be accomplished through a radio if one is available, or through the test port.

With the on-board computers present, test code can be written to run directly on the OBC to exercise any component of the spacecraft. Manually loading this code onto the OBC however is time consuming and would take away from the automation capabilities of the LFFT. This task can be accomplished with assistance of the bootloader, a software component present on all OBCs. The bootloader is the first application that executes when an OBC is turned on. It provides minimal functionality allowing for more complex code to be loaded and executed on the spacecraft via a series of commands sent from the ground. With the use of specialized ground software, this functionality can be utilized to automatically load and execute test software. From this discussion, the minimum requirement for performing any test is a working on-board computer running a bootloader. This constraint was deemed acceptable as it represents a flight-like configuration, and all required components were available at the time of the design.

The final step in the top-level design of the LFFT was the selection of the various tools used for development. For the component of the test framework that operates on the OBCs, the C language was used for its effectiveness on embedded systems, and as it is the language used for SFL flight software. C++ was selected for the ground component of the test framework which will run on a personal computer, so that object oriented constructs could be used to better manage the project, and since computational
efficiency or code size is not a limiting factor on a PC. For ease of setup and configuration, the LFFT software relies on a number of configuration files that contain all necessary test parameters. These files are composed in the Extended Markup Language (XML) format, which was chosen for its inherent ability to organize data, and due to the availability of XML parsing libraries that can easily integrate with any C++ project. The MSXML 6.0 library from Microsoft Corporation was used for this project.

In summary, the test software consists of a framework, and of individual test modules. The framework is responsible for automatically processing user configuration files, loading the test procedure, invoking individual test modules, uploading test code onto the spacecraft, and logging the results. The test framework is executed from a personal computer. Test modules consist of a ground component, and an embedded component that gets uploaded to the spacecraft. These two components work in unison to perform the test and process the results. Alternatively to uploading the embedded test component, it can be prepositioned in the on-board computer’s non-volatile memory. Commands to the bootloader can be used to load this code into RAM, from which it can be executed. This approach dramatically reduces the communications overhead and test run time, which is especially beneficial when performing tests on orbit.

3.3 Detailed Design

With the completion of the preliminary design it was generally accepted that the selected approaches would meet the project requirements. A detailed design followed, concentrating on the specific operations of each module of the LFFT and how these modules interact amongst themselves and with the spacecraft component under test. Figure 3.1 shows a top level block diagram of the LFFT architecture, highlighting the main modules in the project. Each module is described in detail in this section.

3.3.1 Scheduler Module

The scheduler module lies at the heart of the LFFT. It uses the XML parsing module to read a test procedure from the configuration files and executes each test in the procedure in the proper order. This module provides automation capabilities to the test framework.

Through analysis of existing test plans and procedures, two test patterns were identified. A consecutive execution pattern, in which a number of tests are executed sequen-
tially, and a repetitive execution pattern in which a test is executed a number of times repetitively. A combination of these patterns can be used to create more complex test sequences, in which a group of repetitively executing tests are combined consecutively, a sequence of consecutive tests are executed repetitively, or any combination of the above. Analysis has also shown that conditional execution of tests is not required, and thus was not implemented. An option does exist however to halt the test software if a failure in any test has occurred to allow an operator to assess the situation and prevent potential damage to the spacecraft.

A sample test plan that combines sequential and repetitive execution patterns is shown in Figure 3.2. This plan shows that the OBCPING test is executed first, after which the OBCADC and OBCRAM test are executed 10 times. Finally, the OBCFlash test is executed once.

### 3.3.2 Communications Module

In SFL’s current communications hierarchy, ground software does not communicate with the spacecraft directly. All traffic is routed through multiplexers and switches. This is necessary for a number of reasons. Firstly, a different communications protocol is used depending on whether the spacecraft is in orbit and communicates over the radio, or
in the lab and communicates over a wired connection. If the spacecraft is in orbit, it can be controlled from a number of ground stations to maximize the number of contact windows per orbit. The switches direct traffic from the control centre, which is in a single location, to the ground station above which the spacecraft is currently passing. The multiplexers allow multiple ground applications to communicate with a spacecraft simultaneously, without creating collisions. As the control centre, multiplexers, switchers, and the ground station may be located anywhere in the world, they are all connected to each other through the Internet, over TCP/IP.

The communications module of the LFFT serves a number of purposes. First, it establishes a connection with the nearest multiplexer over a TCP/IP socket and performs the necessary authentication. It also encodes all outgoing commands into the form necessary for transmission to the multiplexer as defined by [19], and decodes incoming responses. A queue is used ensure that commands are sent out at a rate the spacecraft can process, and if a response is not received within a timeout period the command is automatically retransmitted. Since the spacecraft under test can be in the lab or in orbit, the communication module sends out the appropriate command dictating whether the spacecraft should respond over its radio transmitter, or over the test port wired connection. The communication module also allows the test framework to connect to multiple on-board computers simultaneously. This greatly simplifies the testing process during long environmental tests.

Figure 3.3 shows a sample configuration file which tests both the housekeeping computer (HKC) and the attitude determination and control computer (ADCC) within a single test procedure. This example assumes that each computer connects to its own multiplexer through the test port. While other configurations are possible, the one presented here is most common during development, and was chosen for its simplicity. The script

```
<TEST_PLAN>
  <TEST_NAME = "OBCPING" />
  <LOOP_COUNT = "10">
    <TEST_NAME = "OBCADC" />
    <TEST_NAME = "OBCRAM" />
  </LOOP>
  <TEST_NAME = "OBCFlash" />
</TEST_PLAN>
```
begins by defining two connection nodes, one for each multiplexer. These are defined within the CONNECTION tags, and contain the TCP/IP address and login information for the multiplexers. The test sequence is listed within the TEST_PLAN tags. To test the HKC, the test script must connect to its respective multiplexer with the CONNECT_TO tag, and select that computer’s destination address with the NSP DEST_ADDR tag. Following the successful establishment of the connection, the OBCPING and OBCRAM tests are executed. The test framework must then disconnect from the HKC, and connect to the ADCC, as specified by the second CONNECT_TO tag. The ADCC’s destination address is selected, and the OBCPING and OBCRAM tests are repeated. This sequence can be placed within LOOP tags and executed repetitively, similarly to the test procedure shown in Figure 3.2.

By providing the functionality described above as a single module within the test framework, it can be reused for each test without necessitating its re-implementation. Further, in conjunction with the multiplexers and switches, this module abstracts the link-specific communication protocols and the physical location of the spacecraft, allowing requirement LFFT.7 to be met.

3.3.3 Test Configuration

This section describes how the LFFT test framework addresses requirements LFFT.5 and LFFT.6, which deal with configurability of the test software. The framework must accommodate multiple missions containing different hardware, as well as the various stages of assembly of any particular spacecraft. This is accomplished through two levels of configuration: one global application configuration file and an optional configuration file for each of the test modules. These configuration files are formatted using XML and parsed in the application using the XML parsing module.

The global configuration file contains non test-specific parameters, which may include information such as the IP address and port number of the multiplexer as described in the Communications Module section. The test procedure is also included in this file, and consists of a list of tests that are to be executed as described in the Scheduler section. Each test entry may contain a path to the optional test-specific configuration file that should be used what that test is executed. This permits for the same test to be executed multiple times with different configuration files. A separate global configuration file can exist for each satellite, including only those tests which correspond to components that are
part of the mission. As the satellite goes through the assembly and integration process, new components would be added to the test procedure. A sample global configuration file is shown in Figure 3.3 and was described previously.

Test specific configuration files may include information such as test pass or fail criteria, or parameters that are specific to the hardware device that is being tested. In general, there are no limitations to what can be included in a test specific configuration due to the flexibility of the XML parsing module. The ability to run the same test with different configuration files proved to be very useful in a number of scenarios. For example, there are six sun sensors on the satellite that must be tested individually. However, depending on which sun sensor is being tested, a different address must be used. Rather than creating six different tests with only the address differing, six configuration files can be created. The test would then run six times, each time with a different configuration file, extracting the address from it [18].

A sample test-specific configuration file is shown in Figure 3.4. The src_source tag of this file specifies that the embedded test component for this test is to be loaded from on-board flash memory rather than uploaded from the ground. The data within this tag specifies the flash block and page numbers in which the test resides, its size in bytes, and the RAM address to which it is to be copied before executing.

3.3.4 XML Parsing Module

As mentioned previously, the LFFT uses configuration files written in XML for its ability to organize and structure data in a logical, hierarchical way, and due to the abundance of existing XML parsing libraries that are freely available for direct integration into the project. The intent of the XML module is to provide an abstract interface to XML based configuration files, while assuming as little as possible about the nature of the data, giving maximum flexibility to the developer of the test. A number of functions were created to read generic data from XML files. Figure 3.5 illustrates the syntax of a function provided by the XML parsing module that allows a single value to be read from an XML file by specifying its path. As implied by the function name, the value is read as an integer. Similar functions to read strings, hexadecimal numbers, and floating values were also created. In this example, the src_source tag of the XML file shown in Figure 3.4 is being read. This module assists in the rapid development of tests by abstracting the details of XML parsing and by providing a generic and convenient way of extracting
Chapter 3. Functional Hardware Testing

Figure 3.3: LFFT global configuration file

```xml
<GNB_LFFT>
  <CONNECTION NAME = "HKC">
    <LOGIN IP_ADDRESS = "255.255.255.254"/>
    <LOGIN PORT = "1000"/>
    <LOGIN USER = "LOGIN"/>
    <LOGIN PASSWORD = "PASSWORD"/>
  </CONNECTION>

  <CONNECTION NAME = "ADCC">
    <LOGIN IP_ADDRESS = "255.255.255.255"/>
    <LOGIN PORT = "1000"/>
    <LOGIN USER = "LOGIN"/>
    <LOGIN PASSWORD = "PASSWORD"/>
  </CONNECTION>

  <TEST_PLAN>
    <NSP DEST_ADDR = "0x20"/>
    <CONNECT_TO NAME = "HKC"/>
    <TEST NAME = "OBCPING"/>
    <TEST NAME = "OBCRAM" CONFIG_FILE = "lfft_obc_ram_test.xml"/>
  </TEST_PLAN>

  <NSP DEST_ADDR = "0x30"/>
  <CONNECT_TO NAME = "ADCC"/>
  <TEST NAME = "OBCPING"/>
  <TEST NAME = "OBCRAM" CONFIG_FILE = "lfft_obc_ram_test.xml"/>
</TEST_PLAN>
</GNB_LFFT>
```

Figure 3.4: LFFT test specific configuration file

```xml
<GNB_LFFT_TEST>
  <VALUE src_source = "FLASH">
    <VALUE flash_block = "1"/>
    <VALUE flash_page = "0"/>
    <VALUE size = "4096"/>
    <VALUE init_addr = "0x11000000"/>
  </VALUE>

  <VALUE result_addr = "0x11000000"/>
</GNB_LFFT_TEST>
```
xmlReader.GetNodeValueInt("//GNB_LFFT_TEST/VALUE/VALUE/@flash_block", &flashBlock);

Figure 3.5: LFFT XML parsing routine

information from configuration files.

### 3.3.5 Test Interface

The test interface acts as the glue code between the test framework and the unit test modules. It defines a set of rules that were followed in the development of test modules, which allowed them to be integrated into the test software. By having a common interface between the test framework and all test modules, tests can be automatically invoked by the scheduler, and can utilize all other framework functionalities such as sending and receiving packets through the communications module, and logging results through the logging module. Other, more specific tasks which are executed often were also implemented as part of the test interface. Examples of these include loading code to the spacecraft, executing the loaded code, and turning switches on the spacecraft on and off. These are referred to as helper functions.

In terms of object-oriented programming, each unit test is implemented as a class. The unit test class is derived from a base class that defines the test interface. The base class contains virtual function prototypes that the scheduler uses to call the test, and the test class implements with the actual test functionality. Helper functions are implemented as member functions of the interface (base) class, and are inherited by the derived test classes.

Figure 3.1 shows three unit test modules connected to the test framework. Each unit test module consists of a ground test component, an embedded test component, and a test-specific configuration file. A number of actions take place when the scheduler executes a test. Firstly, the test interface is invoked and the test-specific configuration file is loaded through the XML parsing module. The test interface then calls the ground component of the test module, passing to it all data from the configuration file. The ground test component calls helper functions from the test interface to upload and execute the embedded test component, and to log results when it is complete.
3.3.6 Logging Module

The logging module was designed to address requirement LFFT.2, which states that the test framework should be as self-documenting as possible. This module provides one centralized point through which all logging information is passed and recorded. It can be used by the application core (scheduler, communications module, XML module) as well as the unit test modules. Thus, the resulting logs contain application level information such as communication failures or improper configuration file contents, as well as the test results. This module guarantees that all pertinent information is logged, at all times, and in a consistent manner.

Every time the test application is started, a new log folder is automatically generated. The name of the folder is dated and time stamped. A test report file is generated within the folder which is also dated and time stamped. This test report contains the build date of the test application, the name of the tester, and global configuration parameters. As tests execute the scheduler module logs the test name and the test-specific configuration file being used. A tally of tests passed, failed, and skipped is reported at the end of the log. The logging module places a copy of the global configuration file and all test-specific configuration files into the log folder, so that the exact test setup can be examined and reproduced in the future. These actions take place automatically, reducing the efforts associated with the development of new test modules. A sample log file corresponding to the test plan in Figure 3.3 is shown in Figure 3.6.

In addition to the information that is logged automatically, a logging application programming interface (API) is provided to be used within test modules. It allows for test-specific results and information to be added to the log. Logging functions accept input in the ubiquitous printf format, which is very well known amongst software developers for its versatility and usefulness. This format provides an easy and quick way of inserting integers, strings, and other variable types into log messages [18].

3.3.7 User Interface

The automated test software was designed as a command line application. This decision was made to both reduce development time, as well as to make it easier to operate the application remotely. The console output seen on the screen is identical to that saved into a text file by the logging module. The XML configuration files can be edited by any text editor or by editors specialized for processing XML. To meet the ease of setup
2010−03−18 20−30−30: Starting GNB LFFT Software
Software Build Date: Mar 9 2010 11:32:04
Tester: Jakob Lifshits

2010−03−18 20−30−34: Setting NSP destination address: 0x20
2010−03−18 20−30−34: Connecting to HKC

2010−03−18 20−30−35: Test Ping, no configuration provided.

Satellite Name: CanX3a
Callsign: VA3SFL
Build: B04H
Compile Date: May 8 2009 02:26:39
Memory Mode: @ − Internal SRAM
Satellite Time: 102
2010−03−18 20−30−35: Test passed.

2010−03−18 20−30−39: Test OBC RAM, Configuration
lfft_obc_ram_test.xml
2010−03−18 20−30−39: Uploading SREC file...

2010−03−18 20−30−41: Executing SREC file...
2010−03−18 20−30−53: Test passed.

2010−03−18 20−31−37: Setting NSP destination address: 0x30
2010−03−18 20−31−38: Connecting to ADCC

2010−03−18 20−31−38: Test Ping, no configuration provided.
Satellite Name: CanX3a
Callsign: VA3SFL
Build: B04A
Compile Date: May 8 2009 02:27:00
Memory Mode: @ − Internal SRAM
Satellite Time: 159
2010−03−18 20−31−38: Test passed.

2010−03−18 20−31−42: Test OBC RAM, Configuration
lfft_obc_ram_test.xml
2010−03−18 20−31−42: Uploading SREC file...
2010−03−18 20−31−44: Executing SREC file...
2010−03−18 20−31−57: Test passed.

Test plan complete:
Tests Passed 4
Tests Failed 0

Figure 3.6: LFFT test report
requirement, an assistant application was created that utilizes a graphical user interface to assist in creating of test procedure. The test procedure is then saved as an XML file. The assistant application can then be used to start the test software with the newly created or modified configuration file, which automatically opens a console window and passes all the necessary parameters to it. An extensive user manual for both command line and graphical operation of the software was also created.

3.4 Deployment, Use, and Results

Since the completion of this project, the test framework has been utilized extensively in a number of situations. It was used as an incremental test during the preliminary integration of BRITE, and during the preliminary and final integrations of AISSat-1. As part of the integration procedure, it was executed after each new component was added to verify its connectivity and that it had no adverse effects on any components that were already present. It was also used as part of the qualification and acceptance testing for the fully assembled AISSat-1 spacecraft as it underwent a number of environmental tests, such as the vibration test and a thermal vacuum functional test, where it assisted in the identification and rectification of a number of issues. After the launch of AISSat-1, the test framework was executed on orbit to verify the health and status of subsystems. The test framework was further extended and adapted to perform unit level tests, taking advantage of its automation capabilities to significantly streamline the testing process, making the testing both more consistent and more efficient.

A number of key lessons were learned while developing and utilizing the test framework. While there are enormous benefits from having such a framework available, the efforts associated with its creation and maintenance are not insignificant. In SFL’s case, its creating is justifiable as it can be leveraged by all GNB-derived spacecraft. The author believes that more appropriate approaches may exist if such test software was to be created on a per spacecraft basis. Many of the design approaches throughout this project were made in order to achieve the desired levels of testability in places where the nominal system would not allow for it. It is recommended that future spacecraft and subsystems are designed for testability at the system and unit levels. Built-in test routines could be used on the ground, as well as on orbit to periodically assess the spacecraft’s health.
Chapter 4

Canadian Advanced Nanospace Operating Environment

As the hardware used on SFL satellites is very specialized and mostly developed in-house, third-party software is generally not available. The majority of the software used on-board the spacecraft and on the ground is also developed by SFL, and is tailored specifically to the flight hardware. One of the key software products developed by SFL is CANOE, or the Canadian Advanced Nanospace Operating Environment. While an early version of CANOE flew on CanX-2, the current version is tailored for GNB on-board computers. Builds of CANOE exist for the housekeeping computer, attitude determination and control computer, and in some instances, the payload computer. Simply stated, the role of an operating environment, or operating system, is to manage resources. A resource can be the main CPU, memory, peripheral devices, or even segments of code such as device drivers. In a real-time multi-threaded operating system like CANOE where multiple application threads execute simultaneously, the operating system ensures that no conflicts between threads occur and that no thread is starved of resources by the others.

4.1 The Scheduler

As mentioned above, a multi-threaded operating system allows a number of threads to run simultaneously. While it may seem to be concurrent, a single-core processor can only execute a single task at a time. To achieve the illusion of concurrency, the operating system alternates loading and running each thread for a short period of time. To the end
user it appears that the execution is parallel. The duration that each thread runs before switching is referred to as a time slice.

The portion of the operating system that handles this action is composed of the task switcher and the scheduler. When a thread is unloaded from the processor, its immediate state must be saved so that it resumes from where it left off when the thread is reloaded. This is performed by the task switcher, also referred to as the context switcher. The scheduler is responsible for selecting which thread is executed during each time slice. Different scheduling algorithms exist, with each possessing advantages and disadvantages over the others. This chapter describes the author’s work in improving the existing scheduler to allow CANOE to meet its performance requirements.

In CANOE, a thread can be in one of four possible states. The thread that is currently executing is said to be awake. A thread that is not currently executing but is ready to run on the next available cycle is said to be waiting. A thread that is not currently executing and cannot run in the next available cycles because certain preconditions are not met is said to be suspended. Finally, a thread that cannot run because it has not been added to the scheduling queue is inactive. As an analogy, an inactive thread is similar to any program on a PC that has not been launched by the user.

There are a number of factors that could cause an awake (running) thread to be unloaded by the context switcher. When the end of a time slice is reached, the operating system pre-empts the execution of a thread and unloads it. In this case the thread is ready to continue running upon the next available opportunity, it is placed in a waiting state. If however an awake thread attempts to use a resource that is occupied by another thread, it cannot proceed any further until the resource is relinquished and is thus placed in a suspended state. This can happen before a time slice fully expires. Once the resource becomes available the operating system places the suspended thread into a waiting thread, and it can proceed execution on the next available time slice. A summary of thread states is provided in Table 4.1.

### 4.1.1 Original Scheduling Scheme

The original implementation of the scheduler in CANOE was based on a round-robin scheduling scheme. This scheme is designed on the premise of giving each thread an equal time slice duration. Threads are arranged in order and are executed one at a time for a full time slice each or until a thread becomes suspended due to resource contention.
### Table 4.1: CANOE thread states

<table>
<thead>
<tr>
<th>State</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Awake</td>
<td>A</td>
<td>The thread is currently executing on the processor.</td>
</tr>
<tr>
<td>Waiting</td>
<td>W</td>
<td>A thread that is ready to run and is waiting for the next available time slice.</td>
</tr>
<tr>
<td>Suspended</td>
<td>S</td>
<td>A thread that cannot execute because it is waiting for a resource that is busy or occupied by another thread.</td>
</tr>
<tr>
<td>Inactive</td>
<td>I</td>
<td>A thread that has not been started and does not participate in the scheduling process.</td>
</tr>
</tbody>
</table>

Once the last thread completes its time slice, the cycle resumes from the beginning. Four round-robin scheduling steps of a hypothetical scenario are illustrated in Figure 4.1.

Each square in the diagram represents a process control block (PCB). One PCB structure exists for each thread and contains information about the thread such as its state, and the process identification number (PID). The PCB of the thread currently executing is highlighted with a cross-hatched pattern. In this scenario initially thread 0 is awake, there are two waiting threads, four suspended threads, and one inactive thread (Figure 4.1a). Thread 0 becomes suspended and must relinquish its time slot to the next waiting thread, thread 3 (Figure 4.1b). As thread 3 is executing, the resource thread 1 was waiting on becomes available, and its status is switched to waiting. Thread 3 also becomes suspended and thread 7 is awakened (Figure 4.1c). It runs for a complete time slice and is pre-empted by the task switcher. Thread 1 is the next awake thread in the queue and it becomes the awake thread (Figure 4.1d).

In this approach waiting threads are executed in a circular order, with no preferential treatment for any particular thread. It is a very simple approach, and lends itself to very efficient implementation. The efficiency of the scheduler is crucial as context switching occurs many times per second, and is considered as overhead to the actual functional threads.

### 4.1.2 Revised Scheduling Scheme

During the implementation of the attitude determination and control system (ADCS) software it became apparent that it would be a challenge to achieve the desired timing performance. This was a combined result of the use of computationally intensive algo-
<table>
<thead>
<tr>
<th>PID: 0</th>
<th>PID: 1</th>
<th>PID: 2</th>
<th>PID: 3</th>
<th>PID: 4</th>
<th>PID: 5</th>
<th>PID: 6</th>
<th>PID: 7</th>
</tr>
</thead>
</table>

(a)

<table>
<thead>
<tr>
<th>PID: 0</th>
<th>PID: 1</th>
<th>PID: 2</th>
<th>PID: 3</th>
<th>PID: 4</th>
<th>PID: 5</th>
<th>PID: 6</th>
<th>PID: 7</th>
</tr>
</thead>
</table>

(b)

<table>
<thead>
<tr>
<th>PID: 0</th>
<th>PID: 1</th>
<th>PID: 2</th>
<th>PID: 3</th>
<th>PID: 4</th>
<th>PID: 5</th>
<th>PID: 6</th>
<th>PID: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE: S</td>
<td>STATE: W</td>
<td>STATE: S</td>
<td>STATE: S</td>
<td>STATE: S</td>
<td>STATE: S</td>
<td>STATE: I</td>
<td>STATE: A</td>
</tr>
</tbody>
</table>

(c)

<table>
<thead>
<tr>
<th>PID: 0</th>
<th>PID: 1</th>
<th>PID: 2</th>
<th>PID: 3</th>
<th>PID: 4</th>
<th>PID: 5</th>
<th>PID: 6</th>
<th>PID: 7</th>
</tr>
</thead>
</table>

(d)

Figure 4.1: Four cycles of CANOE scheduler in round-robin scheme
rithms, limited processing power, and the overhead of the operating system. While the first two factors were unavoidable, improvements to the third were possible.

One of the sources of overhead from the operating system was determined to be related to the scheduling scheme. Because all threads were equal priority, the ADCS thread would be often pre-empted for lower priority tasks. If the ADCS thread was suspended while waiting for a resource, it would have to wait for a full scheduling cycle to complete before executing again, even if the resource became available within a shorter time period.

One of the possible solutions to the above problem is the use of a more sophisticated scheduling approach, namely a priority based round-robin scheme [20]. Unlike the regular round-robin approach, in this scheme threads are assigned priority levels. The scheduler would first pick threads from the highest priority level and schedule those in a regular round-robin fashion. Only once there are no more waiting threads at a higher priority level the scheduler would look for threads in the lower priority level and execute those in a round-robin fashion. If at any time a thread with a higher priority level becomes available, it is given the following time slice.

When developing this scheme it was essential to ensure that the implementation is as efficient as possible, otherwise any gains would be overshadowed by the overhead of the scheduler. After a few design iterations, the following scheme of sorting through process control blocks was developed. A static array was created with each array element corresponding to a priority level. This structure was chosen as the number of priority levels was known at compile time, and an array is perhaps the most efficient structure to access when elements do not need to be moved. Each array element contains a linked list. The linked list elements correspond to the process control blocks of threads of that priority level. The linked list data structure was chosen as it allows priority levels to be assigned to threads at run time, and moving elements within the list is very efficient.

Figure 4.2 illustrates four steps in a hypothetical scheduling sequence using this scheme. The scheduler begins by traversing the linked list attached to the highest priority level (level 0), and runs the first awake thread, thread 0 (Figure 4.2a). Once this thread completes executing, or if the time slice expires, the process control block is moved to the back of the queue for the corresponding priority level. This action enforces a round-robin arbitration within the priority level. Since at this stage there are no waiting threads in priority level 0, the queue corresponding to priority level 1 is searched, and the thread with PID 3 is awakened. While it executes, thread 1 becomes unblocked and switches
to the waiting state (Figure 4.2b). When thread 3 is done executing, it is moved to the back of the queue of priority level 1. Even though thread 4 is ready to execute, thread 1 is selected due to the higher priority level (Figure 4.2c). Thread 4 is selected on the next cycle (Figure 4.2d).

Unlike with a round robin scheduler, introducing priorities to threads can also introduce great risk. A scenario may occur in which high priority threads are constantly available to execute, blocking lower priority threads from running for long periods of time, or even indefinitely. To avoid such issues great care must be taken by the programmer when assigning priority levels. Another issue that may arise from the use of priority based scheduling is priority inversion. Priority inversion refers to the case when a lower priority thread executes before a higher priority one. One sequence of events which can lead to this effect involves three threads, of high, medium, and low priorities. It begins with a low priority thread executing and utilizing a certain resource. A high priority thread awakens and attempts to acquire the resource in use by the low priority thread. It becomes blocked on the resource. At this time a medium priority thread also awakens but is free to run. It interrupts the low priority thread, preventing it from relinquishing the resources under its control. As the low priority thread cannot release its resources, the high priority thread also remains blocked. Indirectly, the medium priority thread is allowed to execute before the high priority thread, causing priority inversion. Scheduler algorithms that avoid this issue do exist, however those add significant complexity and overhead to the system. Because of the relative simplicity of CANOE, it was decided to avoid such algorithms, relying on the careful assignment of priority levels by the programmer instead.

The scheme outlined herein has been implemented and integrated into CANOE. It is currently undergoing testing, and will form the default scheduler in CANOE. By allowing threads to be assigned different priority levels, real-time performance can be achieved for mission critical subsystems such as the attitude determination and control software.
Figure 4.2: Four cycles of CANOE scheduler in priority round-robin scheme
Chapter 5

BRITE Instrument Software

Stellar photometry is the science of accurately measuring the brightness of stars, and how it changes over time \[21\]. The BRITE mission will conduct stellar photometry focusing on some of the brightest stars in the sky, achieving precision at least 10 times higher than what can be seen from Earth \(8\). By measuring variability in brightness on timescales ranging from hours to months, scientists will be able to investigate the effects of solar winds on stellar life cycles, and determine the ages and histories of stars through asteroseismology \(22\).

The BRITE satellite is based on SFL’s Generic Nanosatellite Bus, structurally resembling a cube 20 cm per side and weighing under 10 kg. The power, communications, on-board computers, and attitude determination and control subsystems are all from the standard GNB complement. Due to the high pointing accuracy requirement of BRITE, a star tracker is included in the list of attitude sensors. The payload bay of the spacecraft contains a wide angle telescope.

To maximize the produced scientific data, it is desirable to observe the target stars through different colour filters. This however, must be done without introducing any moving mechanical components such as filter wheels. To overcome this limitation BRITE will consist of a constellation of satellites, arranged in three pairs. Each pair will contain one satellite with a blue filter, and one satellite with a red filter. The telescope will point towards a region of interest in the sky for approximately 15 minutes of each orbit.
5.1 Overview of the BRITE Instrument

The BRITE instrument is very unique in that it packs a high quality optical telescope into a miniaturized package, which also meets the weight and power requirements of a nanosatellite. Unlike most telescopes it provides a relatively wide angle view of the sky of approximately 25° which allows it to observe multiple bright stars simultaneously. The instrument can be divided into three main components:

- Telescope optics and housing
- Detector header board and heaters
- Instrument computer

5.1.1 Telescope

Figure 5.1 shows the mechanical assembly of the BRITE payload bay. The star tracker is mounted to the left side of the mounting bracket, and the telescope assembly is located on the right side. The telescope assembly contains three major sections. Towards the back is the housing and mounting for the header board and charge coupled device (CCD) detector. The middle cylindrical section is the optical cell. It contains a number of lenses that magnify and focus the incoming light and project it onto the CCD plane. The front section contains the baffle, which is designed to block stray light from entering the telescope.

5.1.2 Header Board and CCD Detector

The header board is located at the rear of the telescope. It acts as the mounting point for a CCD detector and signal amplification circuitry. It also contains four trim heaters that can be used to stabilize the CCD temperature, and a microcontroller that drives them. The header board is mounted to the optical cell through four spring-loaded screws that can be shortened or lengthened to achieve the desired focus. Figure 5.2 shows the header board and CCD detector.

5.1.3 Instrument Computer

The instrument computer is located in the +Z electronics tray of the GNB structure (Figure 1.2). While the main computer section is almost identical to other GNB based
Figure 5.1: BRITE telescope

Figure 5.2: BRITE CCD header board
on-board computers, it also contains custom digital and analog electronics that drive the CCD detector. These provide a number of voltage rails and clock signals that are necessary for imaging. The instrument computer board is shown in Figure 5.3. It interfaces with the ADCC and HKC, as well as with the microcontroller on the header board.

5.2 Instrument Computer Bootloader

The bootloader is perhaps the most critical software running on each of the on-board computers. Besides providing rudimentary control of the spacecraft, it allows application software to be modified and reloaded even after launch. The bootloader itself however cannot be changed in flight, therefore great effort is put into testing and debugging it. Once qualified, changes to the bootloader are permitted only under extreme circumstances. The BRITE payload computer bootloader was one such case. Even though the BRITE payload computer is based on the GNB OBC, the unique payload hardware required some of the pins of the processor to be reconfigured from general purpose communications peripherals to payload specific functionality. It was further discovered that the states to which these pins were initialized by the bootloader at start-up were not
compatible with the payload. If left unrectified, this could lead to the payload powering up into an undetermined state and potentially incurring a large continuous power draw. The author was tasked to perform the required changes.

Foreseeing that this issue may arise on future missions, it was decided to modularize the bootloader rather than remove the violating code completely. The components of each module that may not be present on every OBC were placed within pre-processor directives. Including the module names in the spacecraft configuration files of the bootloader would enable them at compile time. Due to the criticality of this software each and every permutation would have to undergo the complete qualification and burn-in procedure for the bootloader. At present the changes to the bootloader code have been completed, and qualification tests executed. The bootloader is currently undergoing a burn-in period through its use on hardware.

5.3 Instrument Computer Application Software

One of the main tasks that the author undertook was the design and development of application software for the BRITE instrument computer. This contribution is very significant as this software is the only means by which the payload can be controlled, and without it the BRITE mission objectives could not be achieved. Due to the project’s importance to the success of the mission, careful attention had to be paid to the design phase, taking into account not only the requirements at the spacecraft level, but also considering how this software would fit into the mission operations scheme. This included considering how the spacecraft would be controlled from the ground, what would be required to perform a single observation cycle, multiple observation cycles, and how the data might be used after it is downloaded from the satellite.

5.3.1 Mission-Level Design and Operations

Before software design could begin, a mission-level operations cycle had to be established and understood. We start by defining the terms observation campaign, observation, and exposure. In the context of this mission, a campaign of observations refers to all of the science data collected from a single star field, over multiple orbits, spanning large periods of time. A single observation refers to a group of exposures collected of the same starfield during the imaging window of one orbit. These exposures are combined
Figure 5.4: BRITE mission top level operations

together to form a single data point. An exposure refers to a single image capture of the star field. Exposures are combined to form observations, and observations of the same star field for the observation campaign. Based on information collected from \[23\], a simplified diagram was created to represent a single observation cycle, and is shown in Figure 5.4. At the start of the cycle the *science consortium*, which consists of the principal investigators and scientists for the mission, decide on which targets will be observed, along with the observation parameters such as exposure time, observation start and end times, the number of exposures to be taken, and what image processing operations should be performed on those exposures. This information is combined into the *observation plan* deliverable, and is entered into the *BRITE Target* ground based application. BRITE Target creates two outputs that are then uploaded to the spacecraft. To understand why two separate spacecraft uploadables are generated, a brief discussion of what happens on the spacecraft is in order.

As described previously, all GNB-based spacecraft carry three on-board computers: a housekeeping computer, an attitude determination and control computer, and a payload or instrument computer. Each computer is designated to perform a specific task, and all three computer must work together to achieve the mission objective. For BRITE, a number of events must happen in order to perform a successful observation. The
attitude of the satellite must be adjusted to point at the target star field. The power subsystem must be configured for the high power draw that results from operating the instrument. The instrument computer must be turned on, and the CCD trim heaters set accordingly. All of these actions must occur autonomously, and with accurate timing. These operations are handled by the Time Tag program on the housekeeping computer. It allows a command script to be executed at specific pre-programmed times.

The first output of BRITE Target is the setup file. It is an XML file that contains all of the information necessary to describe the observation plan in a human readable format. It is passed into BRITE Schedule, a ground application that converts the setup file into a set of scripts of time tagged commands in a format understandable by the spacecraft. These commands are uploaded to the housekeeping computer through the ground program Time Tag. The second output of BRITE Schedule is the upload file. This file contains instructions of how each exposure is to be processed, and is uploaded directly onto the instrument computer. It is saved in the instrument computer’s non-volatile flash memory through the Mass Transfer Protocol ground application.

At the scheduled observation time the housekeeping computer begins executing the uploaded script, commanding the attitude and instrument computers to perform the necessary tasks as shown in the sequence diagram in Figure 5.5. Observation results are stored in flash on the instrument computer, and are downloaded as the spacecraft passes over the ground station. The results are returned the science consortium for analysis, completing the observation cycle.

5.3.2 Subsystem-Level Design and Implementation

With a broader understanding of the mission concept and operations, the systems involved, and how they interact, the unit level design could commence. The requirements for the instrument computer software are described in Table 5.1. It is worth noting that these requirements were generated at an early stage of the program, and thus may be broad.

An examination of these requirements shows that they can be divided into two groups: requirements that deal with general spacecraft tasks such as controlling hardware and handling communications, and requirements that deal with processing of the gathered science data. This leads to the logical conclusion that the software itself should also be split into two categories: the operating environment, and the science data generation code,
Table 5.1: BRITE instrument computer software requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BE7.1.1</strong></td>
<td>Coordination with satellite. The science software shall coordinate between the photometer instrument and the main operating system on the satellite. The science software shall be the only software that interacts with the photometer instrument.</td>
</tr>
<tr>
<td><strong>BE7.1.2</strong></td>
<td>Control of photometer. The science software shall be able to fully control and operate all parts of the photometer instrument electronics.</td>
</tr>
<tr>
<td><strong>BE7.1.3</strong></td>
<td>Accepting commands. The science software shall be able to accept commands from the ground (relayed through the operating system), or from other software threads running on the satellite to take an image. These commands include, but are not limited to taking images, manipulating data, and data storage and retrieval.</td>
</tr>
<tr>
<td><strong>BE7.2.1</strong></td>
<td>Time knowledge. The science software shall be able to provide accurate absolute time information with each image. The time shall be accurate to 0.1 s.</td>
</tr>
<tr>
<td><strong>BE7.2.2</strong></td>
<td>Exposure time. The science software shall provide accurate exposure time with each image. The time shall be accurate to 0.01%.</td>
</tr>
<tr>
<td><strong>BE7.2.3</strong></td>
<td>Field Flattening. It shall be possible to easily perform field flattening on the ground. It should be possible to perform field flattening in orbit.</td>
</tr>
<tr>
<td><strong>BE7.3.1</strong></td>
<td>Sub-rastering. The science software shall be able to analyze an image taken by the photometer to establish appropriate regions of interest.</td>
</tr>
<tr>
<td><strong>BE7.3.2</strong></td>
<td>Binning. The science software shall be able to bin a full image such that 1K x 1K image is stored.</td>
</tr>
<tr>
<td><strong>BE7.3.3</strong></td>
<td>Averaging. The science software shall accommodate averaging or mediating of individual short exposures of windowed images for improvement of S/N and for cosmic ray removal.</td>
</tr>
<tr>
<td><strong>BE7.3.4</strong></td>
<td>Compression. The science software shall be capable of performing lossless compression of the stored image to reduce download requirements.</td>
</tr>
<tr>
<td><strong>BE7.3.5</strong></td>
<td>Co-adding of images. The science software shall be able to coadd images of each region of interest taken during a nominal 15 min observation window.</td>
</tr>
</tbody>
</table>
or SDGC. Such a divide also makes sense from an implementation perspective. SFL developers are very familiar with the GNB architecture and the spacecraft hardware, which makes them ideal candidates for developing the operating environment. The scientists on the other hand are much better suited for the development of science data processing code. To successfully develop complex software in separate modules, a detailed interface control document was created.

**Topology**

At the outset of the detailed design, a software topology had to be selected. Software topology refers to the general layout of the program and the execution path of the code. Two general topologies were considered: a multi-threaded environment in which two or more functions appear to execute simultaneously, and a loop topology in which functions execute one after the other in a loop. A multi-threaded environment such as the one provided by CANOE can be advantageous as a number of lengthy functions take turns using the processor, providing the illusion that they occur simultaneously and no single task is starved of resources. The disadvantage of such an environment is that the sequence in which events happen may be difficult to control and analyze. Special care must be
taken to ensure that any data used by multiple threads is accessed in the proper order and that no race conditions occur. Because tasks can be interrupted, timing fidelity is also harder to guarantee. In a loop topology, code is only interrupted by interrupt service routines, which are few and execute very quickly. Thus it is much more predictable, and simpler concurrency protection mechanisms are required. In this topology, concurrency protection may be necessary to protect any program variables that are shared by the program loop and interrupt service routines. The disadvantage of a loop topology is that the loop iteration time may be long and some tasks and device driver may not be serviced quickly enough. For this project the loop topology was selected for its relative simplicity and determinism. Measures were taken to ensure that hardware drivers are handled sufficiently often, while minimizing the use of interrupts to ensure predictable operations.

State Machine

In a loop based topology the main loop is executed indefinitely, or until an exit condition is encountered. Because the code runs over and over, there is a need keep track of the software state. The software state refers to which task was executed in the previous loop iteration, and which task must be executed in the current iteration. The simplest way to keep track of this information is through the use of a state machine. To maximize code reuse and reduce development time this project utilizes a generic, configurable state machine previously developed within SFL.

The state machine controls the sequence of events that occur when an observation is performed. It corresponds to the time between the OBSERVATION_START and OBSERVATION_STOP commands on the IOBC time line in Figure 5.5. After power up and initialization, the application software awaits for commands in the IDLE state. To begin an observation an OBS_START command must be sent to the IOBC, containing the name of the upload file to be used. Amongst other information, the upload file specifies the observation mode which can be MANUAL, AUTO, and AUTO_DELAYED. These are shown in Figure 5.6, Figure 5.7, and Figure 5.8 respectively, and are described below.

In MANUAL mode each exposure composing the observation must be commanded manually. After entering this mode the state machine awaits in the WAIT_EXP_START state for an EXPOSURE_START command. When this command is received, an exposure is initiated and a transition into the EXP_IN_PROGRESS state occurs, in which
the software awaits for the exposure to complete. Once complete, the state machine transitions into \texttt{EXP\_POST} state to process and store the resulting image. At this point a check is made whether the number of exposures for this observation has been met. If so, the state machine transitions into the \texttt{OBS\_FINALIZE} state to store any observation results, and then returns to the \texttt{IDLE} state. If the number of exposures for the observation has not been met, the state machine returns to the \texttt{WAIT\_EXP\_START} state. As can be seen, in this mode only one observation occurs.

\textbf{AUTO} mode is much like \textbf{MANUAL} mode, but provides the facility for higher observation throughput. In this mode, once an observation is started with the \texttt{OBSERVATION\_START} command, no additional commands are required to initiate exposures. The software automatically starts an exposure and transitions into the \texttt{EXP\_IN\_PROGRESS} state. After an exposure is complete and processed a check is performed to determine whether the number of exposures in the observation has been met. If not, another exposure is automatically initiated. Otherwise the observation is finalized and a new observation is started. This cycle continues indefinitely until an \texttt{OBSERVATION\_STOP} command is sent. \textbf{AUTO} mode is intended to maximize observation output by reducing the overhead of the extra commands needed in manual mode.

The last mode, \textbf{AUTO\_DELAYED}, is almost identical to the \textbf{AUTO} mode, with the addition of the \texttt{CLOCK\_ALARM} state. This state allows the absolute time of the next exposure to be specified. The state machine waits before the specified time has elapsed before initiating the exposure. This mode can be used when precise intervals between exposures are desired, which is important for differential imaging.

\textbf{Device Drivers}

In order for the operating environment to gain full control over the instrument hardware, a number of device drivers were developed. Some of these drivers were based on existing code from the bootloader and other applications, while others had to be written from scratch. The list of drivers and a brief description is presented in Table 5.2.

\textbf{Communications}

One of the main features provided by the operating environment is an inter-OBC communications framework. Commands can be sent and received from multiple sources. All commands that are sent to the instrument computer from the ground, as well are those
Figure 5.6: BRITE IOBC manual state transition diagram
Figure 5.7: BRITE IOBC automatic state transition diagram

num_exposures
specified in upload file

[exposures_taken < num_exposures]

[exposures_taken = num_exposures]

num_exposures
taken = num_exposures

taken < num_exposures
Figure 5.8: BRITE IOBC automatic delayed state transition diagram
Table 5.2: BRITE instrument computer device drivers

<table>
<thead>
<tr>
<th>Device</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART</td>
<td>The Universal Asynchronous Receiver Transmitter (UART) peripheral facilitates asynchronous serial communications between devices. In the BRITE architecture it is used to establish links between each of the on-board computers, as well as between the on-board computers and some sensors and actuators.</td>
</tr>
<tr>
<td>DMA</td>
<td>The Direct Memory Access (DMA) peripheral can automatically transfer data between two memory-mapped devices without tying up the processor for the duration of the transaction. In the BRITE instrument computer architecture it is used to transfer data received by the UART into external RAM.</td>
</tr>
<tr>
<td>HET</td>
<td>The High End Timer (HET) is a sophisticated timing device on the main processor. On BRITE it is used for high precision time keeping.</td>
</tr>
<tr>
<td>I²C</td>
<td>The Inter-Integrated Circuit (I²C) is a synchronous multi-master bus. On the BRITE instrument computer it is used as the communications link to the CCD header board.</td>
</tr>
<tr>
<td>SPI</td>
<td>The Serial Peripheral Interface (SPI) is a synchronous multi-slave bus. It is used to communicate with a number of peripherals on the instrument computer board.</td>
</tr>
<tr>
<td>TWS</td>
<td>The Three Wire Serial protocol is a synchronous serial interface very similar to SPI. Unlike SPI, there is no dedicated controller for TWS on the processor. Instead, it was implemented by directly controlling input/output pins.</td>
</tr>
<tr>
<td>GPIO</td>
<td>The General Purpose Input Output (GPIO) driver provides an interface for controlling pins on the processor. This driver is used by the TWS driver, as well as to control imager signals.</td>
</tr>
<tr>
<td>KSC1000</td>
<td>The KSC1000 is a high precision timing chip that generates signals to drive the CCD. It communicates with the processor over the TWS link, and is used to provide precise exposure times.</td>
</tr>
<tr>
<td>AD298</td>
<td>The AD298 is a high speed analog-to-digital converter that converts analog data from the CCD.</td>
</tr>
<tr>
<td>MAX1231</td>
<td>The MAX1231 device is an external analog-to-digital converter used to measure telemetry points on the board. It communicates with the processor through SPI.</td>
</tr>
<tr>
<td>Processor ADC</td>
<td>This is the processor analog-to-digital converter, used to measure telemetry points on the board.</td>
</tr>
<tr>
<td>SDRAM Controller</td>
<td>The SDRAM controller is implemented on an FPGA and is used to store data from the CCD into SDRAM. It is a memory mapped peripheral.</td>
</tr>
</tbody>
</table>
sent through the Time Tag program are routed through the housekeeping computer. The instrument computer also communicates with the attitude control computer to gather attitude related telemetry. These connections are established through the two UART ports on the processor. The design of the communications system was especially challenging as it had to be fast enough to meet minimum data transfer requirements while minimizing the use interrupts.

**Receive Functionality** Three high level approaches were considered for the communications system receive functionality. The first and simplest approach considered was the polling loop. When a byte is received by the UART peripheral it is stored in a register and a flag is raised. On every iteration of the main program loop the receive flag could be checked, and if set, the character copied into a buffer where frame synchronization would be performed. This approach was insufficient as it would not be able to keep up with the incoming traffic. Since the UART hardware register is only one character long, this character must be copied out before the next character arrives, or it would be dropped. As each main loop iteration may be significantly longer than the character arrival rate, communications would be highly unreliable.

The second approach deals with the above issue by allowing an interrupt to be generated when the receive flag is set. An interrupt allows the main loop execution to momentarily pause while the UART handling code copies the received character into a buffer, before resuming the execution where it left off. This approach would guarantee that all characters are processed in a timely manner, but it would make the main loop execution time very unpredictable. As the communications rate is increased, so would the frequency of interrupt generation. Eventually a point would be reached at which the time handling interrupts is a significant portion of the total execution time, making the system inefficient.

The third approach overcomes these issues through the use of Direct Memory Access. The DMA peripheral can be configured to work in tandem with the UART. Every time the receive flag is set by the UART, the DMA copies the received character from the UART register directly into a specified memory buffer without requiring processor intervention. If the buffer is large enough, it can contain a number of received packets. Thus, if the main loop is busy processing data, received packets can be queued up and handled at a later time without losing any data. The DMA peripheral can be configured to generate an interrupt once the destination buffer is full. While the documentation specified that
the DMA can also report how many characters were transferred, a hardware bug was identified which prevents this feature from working. This has a very serious consequence in that there is no easy way to determine if any data has been received until the buffer has been completely filled. If a packet arrives that is smaller than the buffer, it may remain unhandled for long periods of time, which is unacceptable.

The selected approach was a combination of the three methods described above. A large memory buffer was dedicated for each of the two UART channels. Each buffer was further subdivided into smaller sections, henceforth referred to as frames. Initially, the DMA is configured to transfer incoming data from the UART into the first frame of the memory buffer. This initial configuration is shown in Figure 5.9a. Three pointers are used to keep track of the buffer state. The frame pointer points to the frame that the DMA is currently writing into. The read pointer represents the next location to be read by the software. The DMA write pointer points to the next location to be written to by the DMA. It is internal to the DMA hardware, and is not known due to the hardware bug described above. Incoming data is written into the DMA buffer until the current frame is completely full. An interrupt is triggered, and the frame pointer is advanced by the interrupt service routine (Figure 5.9b). In every main loop execution cycle the software polls these DMA buffers searching for complete packets. Any complete frames between the read pointer and frame pointer are known to have new data. However, as the DMA pointer cannot be read from software, it is not known if there is any new data in the currently active frame until it is full. The read pointer is advanced to the beginning of the new frame and any complete packets are processed (Figure 5.9c). The current frame is then scanned for synchronization characters to determine if it also contains complete packets. Figure 5.9c represents a scenario in which some data has been transferred into the current frame, however this data does not comprise a complete packet. After some time, the remainder of a packet arrives and is stored in the frame as shown in Figure 5.9d. As the read pointer scans within the current frame, it locates packet synchronization characters and the packet is processed. The read pointer is advanced to the end of the packet within the frame as shown in Figure 5.9e. When the frame pointer reaches the end of the buffer, it wraps around to the beginning but stops short of reaching the read pointer to ensure that old commands are never overwritten before being handled (Figure 5.9f). This scheme generates relatively infrequent interrupts, using them only when a frame overflows rather than for every incoming character. It guarantees that packets are processed in a timely manner, and that data is never lost if a sufficiently large buffer is
provided.

**Send Functionality**  Up to this point we have only discussed receiving commands by the instrument computer. Some special considerations had to be given to handling commands that are sent from the instrument computer to other computers on the spacecraft. Firstly, a method had to be devised to determine whether an incoming packet is a command to the instrument computer, or a response to a command sent from the instrument computer. Secondly, since a non pre-emptive topology was selected, the receive functionality had to be non-blocking. A blocking receive function is one that would not proceed until a response has been received, preventing the rest of the loop from executing.

To overcome the first issue, knowledge of the internal structure of the communication packet is utilized. Every time a command is sent out the destination address and the command code are saved. If a received packet’s source address is equal to the saved destination address, and the command codes are also equal, it is assumed to be a response. The state of the send/receive state machine is updated as described below. This is handled by the application code’s receive functionality.

The second issue was overcome by implementing the send/receive functionality as a state machine. This was done by creating a single ‘send_receive’ function that performs a different task depending on the state of the state machine during the function invocation. Initially, the state is set to ‘IDLE’ indicating that no commands have been sent and no responses are expected. When the function is called in this state, the request command is sent out, and the internal state is transitioned to ‘IN_PROGRESS’ indicating that a response is awaited. The function returns immediately with a return code of ‘IN_PROGRESS’, and the main code can continue to execute. The receive functionality of the main code checks incoming packets for a response as described previously. If a response to this request is identified, the state of the state machine is updated to ‘RECEIVED’. If a response has not been received in a predefined period of time, the state is set to ‘TIMEOUT’. As the ‘send_receive’ function is invoked on each following loop iterations, the state is again checked. If a response has not been received and a timeout event has not occurred, the function returns ‘IN_PROGRESS’ and the state also remains ‘IN_PROGRESS’. Otherwise, a return code of ‘TIMEOUT’ or ‘RECEIVED’ is returned, and the state is transitioned to IDLE. Upon the next invocation of the function, a new request will be sent. Table 5.3 summarizes the state transitions and return codes of the ‘send_receive’ function. The first column indicates the state of the state
Figure 5.9: UART/DMA buffering scheme
Table 5.3: Non blocking receive call state machine

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>Return Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>IN_PROGRESS</td>
<td>IN_PROGRESS</td>
</tr>
<tr>
<td>IN_PROGRESS</td>
<td>IN_PROGRESS</td>
<td>IN_PROGRESS</td>
</tr>
<tr>
<td>RECEIVED</td>
<td>IDLE</td>
<td>RECEIVED</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>IDLE</td>
<td>TIMEOUT</td>
</tr>
</tbody>
</table>

machine at the time of the function invocation. The second column indicates the state to which it transitions when the function exists. The last column indicates the exit code returned by the function to its caller. As can be seen, transitions from ‘IDLE’ to ‘IN_PROGRESS’, as well as from ‘RECEIVED’ and ‘TIMEOUT’ back to ‘IDLE’ occur by invocation of the ‘send_receive’ function. Transitions from ‘IN_PROGRESS’ to ‘RECEIVED’ and ‘TIMEOUT’ are made by the receive functionality of the code. The pseudo-code for the non-blocking function call within the main loop is shown in Figure 5.10. This code emphasizes that the function call is non-blocking as it returns immediately and allows the rest of the program, which includes the receive functionality, to continue executing. It also shows how the receive functionality updates the state machine state when a response is received. In this configuration, a new request will automatically be sent if a timeout event occurred.

Telemetry and Command

Telemetry and command, (or T&C for short), is an aspect common to all spacecraft on-board computer systems. Commands are the primary means by which the spacecraft can be communicated with, commanded to change modes, and directed to perform tasks. Likewise, commands can be sent to query the various states, voltage, currents, and temperatures on the spacecraft, collectively referred to as telemetry. Because the spacecraft is inaccessible after launch, commands need to be general enough to allow the operator to handle any number of unpredictable scenarios. Telemetry points need to be plentiful enough so that issues can be narrowed down and debugged in as much detail as possible. The BRITE instrument computer implements general purpose commands that are consistent in format and functionality to the general commands on all other GNB derived computers and spacecraft, as well as commands that are application specific to the BRITE mission. A number of telemetry points are monitored as both an indication that the hardware is functioning properly, and to verify that the spacecraft environment

Figure 5.10: Non blocking receive call

is suitable for performing scientific imaging. The implemented commands are described in Appendix A [25].

Time Keeping and Synchronization

One of the key requirements of the BRITE instrument computer is to provide absolute time information to within 100 ms of ground time. This requirement imposes two separate challenges: synchronizing the on-board clock to ground time, and maintaining time accuracy once synchronized.

Since all commands to the instrument computer must pass through the housekeeping computer, there may be significant latency as the packet is analyzed and routed by the housekeeping computer operating system and device drivers. These delays depend on the state of the operating system and can be highly indeterministic, which makes it very difficult to put an upper bound on the latency. The problem can be simplified by dealing with synchronizing the housekeeping computer to ground time, and synchronizing the instrument computer with the housekeeping computer as two separate tasks. The former is outside of the scope of this work, and remains to be addressed. The latter is performed
in two steps. First, the HKC sends a command to the IOBC which puts it into a time synchronization mode. In this mode the IOBC will not perform any regular tasks but will await a time synchronization command that contains the current HKC time. With this approach the main loop time of the IOBC is minimized, ensuring that the delay between receiving the command and saving it in the local clock is as brief as possible. The effectiveness of this approach remains to be tested once the necessary HKC code is implemented. This is shown in Figure 5.11.

Once synchronized, the clock must then maintain the specified time accuracy until it can be resynchronized again. Analysis has shown that the on-board oscillator stability is sufficient to maintain the desired accuracy for approximately 12 hours. This does not pose any problems as there are typically up to three passes during this interval. The on-board clock in CANOE is implemented on a millisecond timer. This hardware timer generates an interrupt every millisecond that is accumulated to the total time. To minimize the use of interrupts and general processing overhead associated with the clock, the High End Timer (HET) peripheral was used for the IOBC software. This peripheral is part of the on-board processor, and is in itself a microcode execution unit. It can be programmed through a highly specialized assembly language to perform all necessary time keeping functionality, without generating any interrupts. Processor intervention is required only to set a new time, or to read the current time. This is the first use of this peripheral on SFL missions, and the author believes that its use on future missions can alleviate some of the loading of current software on the processor.

SDGC Interface

As mentioned previously, the BRITE IOBC application code was split into two portions: the operating environment, and the science data generation code. An interface was developed to allow these parts to be developed independently and to ensure that the final product met all functionality requirements. The interface consists of five functions that are called by the operating environment at various state transitions to perform science data processing. These are described below.

- **SDGCSetupObservation:**
  The ‘setup observation’ function is called by the operating environment whenever a transition between the IDLE and OBS_START states occur. The SDGC performs any necessary pre-exposure setup, and based on the upload file determines the
Figure 5.11: BRITE IOBC time synchronization state transition diagram

number of exposures that must be taken and how they should be processed. If this function returns successfully a transition into the next appropriate state occurs depending on the observation mode.

- **SDGCFinalizeObservation:**
  The ‘finalize observation’ function is called when the OBS_FINALIZE state is entered. This can occur either when an observation is complete, or if an OBS_STOP command is sent. The SDGC must differentiate these two cases and handle them appropriately. At this point the final observation results are processed and stored into flash memory.

- **SDGCPReExposure:**
  The ‘pre exposure’ function is called upon transition into the EXP_IN_PROGRESS state, immediately before an exposure is taken. This gives the SDGC an opportunity to verify all telemetry points are within acceptable range before the exposure begins, and to store the exact time of the exposure start.

- **SDGCPPostExposure:**
  The ‘post exposure’ function is called when transitioning from the EXP_IN_PROGRESS state into the EXP_POST state. At this stage the exposure is guaranteed to be complete and copied from the CCD into SDRAM. This function can then extract the
regions of interest and perform any post processing, such as binning or co-adding to previous exposures of the observation.

- **SDGCACSCLoop:**
  The ‘ACS loop’ function is called on every iteration of the main loop while in the EXP_IN_PROGRESS state. It allows the SDGC to poll telemetry points from the IOBC as well as the ADCC to verify that they are within acceptable limits and that the spacecraft pointing is on target.

### 5.4 CCD Header Board Software

The CCD header board is located at the rear of the telescope and performs a number of important functions. Primarily, it serves as a mounting point for the CCD and provides its electrical interface, which consists of a number of different voltage supplies, clock signals, and the output amplification chain. The header board also houses the thermal control system for the CCD, which is necessary for two reasons. Firstly, the noise in an image increases with temperature. Secondly, the output level of each pixel also increases with temperature for a given exposure time. The latter is the most significant to this mission. Since each observation consists of a stack of co-added exposures, it is essential that the temperature of the CCD for each exposure remains the same. Failing to do so would produce meaningless scientific results. While it is beneficial to operate the CCD at the lowest possible temperature to reduce noise, it is more important to operate the CCD at a constant temperature throughout the observation cycle.

The thermal control system consists of heaters, temperature sensors, and a microcontroller. The microcontroller receives commands from the instrument on-board computer, reads the temperature sensors, and controls the heaters. Four heaters are located at the four corners of the CCD, and each temperature sensor is placed equidistantly between two heaters along an edge of the detector. This section describes the work performed by the author on software development for the CCD thermal control system. Because an active cooling system is not feasible on a spacecraft of such limited size and power, the operational plan calls for the CCD to operate just above the maximum temperature that would be naturally seen on orbit. This would allow maintaining the CCD temperature at a constant value with the use of heaters only. Such a scheme requires the thermal design of the spacecraft to ensure that the CCD temperature is biased cold when no heaters are
Table 5.4: CCD header board requirements

<table>
<thead>
<tr>
<th></th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>During scientific operations the thermal control system shall ensure that the temperature of the CCD detector is stable to within 5 °C during a single observation (up to 15 minutes) [26].</td>
</tr>
<tr>
<td>2</td>
<td>If possible, the thermal control system should ensure that the temperature of the CCD detector is within 1 °C during a single observation [26].</td>
</tr>
<tr>
<td>3</td>
<td>The header board shall communicate with the instrument on-board computer via an I²C bus.</td>
</tr>
<tr>
<td>4</td>
<td>All communications to and from the header board shall use KISS and NSP encoding as described in [19].</td>
</tr>
</tbody>
</table>

active.

5.4.1 Requirements

A number of software and hardware requirements have impacted the development of the header board. These are outlined in Table 5.4. The first two deal with thermal control. They drive the selection of the heaters and the amount of power they will require, as well as the controller performance characteristics such as steady state error and overshoot. The final two requirements dictate the hardware and software protocols that shall be used for communications between the header board microcontroller and the instrument computer.

5.4.2 Architecture and Resource Allocation

Following the subsystem requirements it can be seen that the code on the microcontroller has to perform a number of tasks in a real-time manner. It must simultaneously handle communications with the IOBC, measure four temperature sensors, execute a controller cycle based on the desired and measured temperatures, and apply the controller output to the heaters. Any idle processor time is occupied by memory scrubbing and health monitoring routines. To successfully achieve the desired performance on a computationally limited microcontroller a detailed design had to be conducted. The design process included allocation of microcontroller peripherals to ensure that the available resources were sufficient to meet all requirements.

The heaters on the header board are implemented with surface-mount resistors. When a voltage is applied to a resistor, a current forms that is inversely proportional to its resistance. All of the power dissipated by the resistor is released as heat. It can be
derived from Ohm’s law as: \( W = \frac{V^2}{R} = I^2R \). This relation suggests two ways of controlling the power dissipated through an ideal fixed resistor: either by controlling the voltage drop across the resistor, or by controlling the current that flows through it. Both of these methods would require analog driver circuitry that would be challenging to fit on a board as small as the header board, and would incur a loss in efficiency as any additional electronics would. A pulse-width-modulation (PWM) approach was selected as an alternative. By applying a square wave of fixed period and variable duty cycle to the resistor, the average power could be varied. As the exact thermal properties of the header board were not yet well known, the ideal PWM period could not be determined. Preliminary tests suggested that the thermal time constant was quite high, yielding relatively long periods on the order of seconds. The system should be flexible enough to allow the PWM period to be adjusted as part of the controller tuning procedure that would be performed under thermal vacuum tests.

Allocation of microcontroller peripherals was performed iteratively, evaluating the feasibility of each configuration at every step. First, all obvious assignments were made. For instance, the SMBus module was assigned to the I²C communication pins as that is the only module on the microprocessor compatible with the I²C protocol. The analog-to-digital (ADC) module was assigned to the temperature sensors, which output an analog signal and must be converted before they can be used in the controller. Timer assignment proved to be more challenging, as there are four general purpose and one special purpose timer on the microcontroller. With each timer operating from a different timer base and at different resolutions, it was necessary to ensure that the final assignment met all timing requirements. Two of the general purpose timers had to be assigned to the SMBus module to provide timing for communication signals. The others were used to time the temperature sensor conversions, execute the controller at a fixed interval, and to generate the PWM outputs.

Two options were considered for generating PWM signals for the heaters. The first approach uses general purpose timers to manually generate the waveforms. The PWM period and duty cycle could be changed relatively easily in this scheme. Every time the timer overflows, it triggers an interrupt and a counter is incremented. When the counter reaches the duty cycle count, the signal is switched from high to low. When the count reaches the period count, the signal is switched from low to high, and the counter is reset. This scheme is simple, but to allow for high resolution changes in PWM duty cycle a relatively fast timer would be required. This would generate frequent interrupts, heavily
loading the microcontroller. The second option involves using a specialized counter-timer peripheral in the microcontroller that is capable of generating PWM signals without software intervention. It was determined that to achieve the desired range of PWM periods, this peripheral has to be clocked by one of the general purpose timers. While this solution uses two peripherals instead of just one, it significantly relieves the amount of required software intervention.

5.4.3 Controller

The function of the controller is to adjust heater power in a manner that would cause the header board temperature, as reported by the on-board temperature sensors, to match the commanded desired temperature. The ubiquitous proportional-integral-derivative controller was selected for this purpose. It is simple to implement in the discrete form, does not require much computational power, and has been proven in many industrial applications. Most importantly, it can be tuned intuitively by analyzing the settling time, overshoot, and steady state error of a step response. Initial tuning can be performed on the ground, and final tuning will take place in orbit by adjusting the three controller constants during the commissioning phase of the spacecraft. The complete controller topology is shown in Figure 5.12.

The four temperature sensors on the header board are measured through the ADC, and their temperatures are averaged. The average temperature is then subtracted from the desired temperature to form an error signal, which is fed into the PID controller.
The PID controller contains four tuneable parameters: $K_p$, $K_d$, $K_i$, and the integrator saturator which is necessary to avoid excessive integrator windup. The output signal from the controller represents the amount of power that should be added to the system to meet the desired temperature. It must be converted to a PWM duty cycle. First, the controller output signal is saturated to the maximum power the system can produce. The PWM duty cycle is then determined as the fraction of commanded power to the maximum power the heaters can produce, and is denoted by $K_{PWM}$. The maximum power that can be generated by the heaters is a tuneable parameter that can be determined analytically and verified experimentally.

In continuous time the transfer function of a PID controller is given by Equation 5.1.

$$C = K_p + \frac{K_i}{s} + sK_d$$  \hspace{1cm} (5.1)

A sample implementation of a discrete PID controller is presented in Figure 5.13 [27].

### 5.4.4 Performance and Results

The discrete PID controller described in Figure 5.13 has been implemented, and initial testing was conducted. This testing is considered only initial as it was performed at room temperature conditions at standard pressure. As the controller will operate in vacuum on orbit, a number of differences are expected to be seen. Primarily the maximum increase in temperature ($\Delta t$) generated by the heaters will be much greater due to the absence of heat losses to convection. This will also however reduce the rate at which the header board cools down when the heaters are off, which is the only way to decrease the header board temperature in this system. The thermal variation of the environment is expected to be higher on orbit, and the conduction paths from the header board to the spacecraft may also differ from the bench setup.

Figure 5.14 shows the controller performance. Initially the controller is turned off and the board temperature oscillates due to fluctuations in the ambient temperature. These oscillations have a magnitude of approximately 1 °C. When the controller is turned on at time 4350 s, the output is briefly saturated to 100% duty cycle, but drops as the board temperature approaches the setpoint. The output saturation is an artefact of the controller gains selected for this test. It is possible to avoid output saturation with lower gains, however the response of the controller would also be slower. Small oscillations of the board temperature and controller output are seen due to the fluctuations in the
const float iSaturatedMin;
const float iSaturatedMax;
const float outputSaturatedMin;
const float outputSaturatedMax;

float PID(float current, float desired) {
    static float iState = 0;
    static float dState = 0;
    float err;
    float output;

    err = desired - current;

    iState = iState + err;
    if (iState > iSaturatedMax) {
        iState = iSaturatedMax;
    } else if (iState < iSaturatedMin) {
        iState = iSaturatedMin;
    }

    output = Kp * err + ki * iState - kd * (current - dState);
    if (output > outputSaturatedMax) {
        output = outputSaturatedMax;
    } else if (output < outputSaturatedMin) {
        output = outputSaturatedMin;
    }

    dState = current;

    return output;
}

Figure 5.13: Discrete PID controller
ambient temperature. The steady state error is bounded to \( \pm 0.1 \, ^\circ C \), well within the requirements. All gains for this test were tuned experimentally. A thermal vacuum test with representative ambient temperature profiles and heat conductance paths remains to be done to verify the effectiveness of this control scheme, as well as to perform preliminary controller tuning. Final tuning will be done during the commissioning phase of the spacecraft in orbit.
Chapter 6

Star Tracker Driver

One of the key enabling technologies that allows SFL to develop advanced spacecraft is a state of the art attitude determination and control system (ADCS). In its default configuration, this system can provide attitude determination to within 1.5° RMS, and pointing accuracy of 2° RMS or lower on all three axes [28]. The nature of the BRITE mission however imposes much stricter requirements. When performing differential photometry, a sequence of images taken over time are compared to determine a trend in the brightness of stars. To do so successfully, it is essential that the telescope is able to reacquire the target to sub-pixel accuracy. This implies a required attitude determination accuracy of 1′ [29], which is not achievable using the standard complement of attitude sensors and actuators. A star tracker is used on the BRITE mission to supplement the standard attitude determination and control system.

The star tracker consists of a high resolution CCD sensor, an optical lens, and a digital signal processor. It operates by capturing images of the sky and comparing them to an on-board star map. By matching star constellations in the captured image to those in the star map, the spacecraft attitude in inertial space can be determined with the required level of accuracy. The field of view provided by star tracker lens is relatively wide at 30° [30] matching that of the BRITE optical instrument.

A number of software components were required to accommodate the addition of a star tracker. Firstly, a device driver was needed to communicate with and control the unit. The driver had to be integrated into the CANOE operating environment that runs on the attitude determination and control computer. Finally, the attitude determination filter had to be extended to make use of the additional input. Rigorous testing at the unit and system levels was necessary as with any other flight software component. This
chapter describes the author’s work on developing the star tracker driver and the unique testing strategies that had to be applied due to limitations in hardware availability.

6.1 Star Tracker Interface

The star tracker unit defines a number of interfaces that are used to communicate with it, and that must be considered for device driver development. The hardware interface defines a signalling scheme used to transfer data over the physical connection medium. The software interfaces define how the data is interpreted.

6.1.1 Hardware Interface

A serial transfer protocol is utilized to communicate with the star tracker. The signal levels conform to RS422 electrical specifications, which define a balanced differential multi-drop bus. A logic-level shifter on the OBC converts the RS422 differential signalling used by the star tracker to the single-ended TTL signalling of the OBC.

6.1.2 Software Interface

Two fundamental modes of operation can be defined for the star tracker, with each requiring a different software interface. The two modes are command mode, in which the star tracker receives and processes commands, and a file transfer mode, in which the star tracker sends and receives large quantities of binary data.

Command Interface

The command mode of the star tracker is used to send small quantities of data that contain commands to the star tracker. Commands can be used to configure star tracker settings such as exposure times and gains, or to direct the star tracker to capture an exposure, perform attitude determination routines, or initiate an image download. When in this mode, the star tracker input and output consists of ASCII characters. Commands are typically single letters followed by comma delimited numerical arguments. Responses are made up of human readable strings that are followed by a “SUCCESS” or “ERROR” keywords that signify whether or not a command was successfully completed. Such an interface is very convenient when the unit is operated manually by a human, however it
Figure 6.1: Star tracker sample output

is very awkward and resource intensive to process automatically by machine. A sample output from the star tracker is shown in Figure 6.1.

**File Transfer Interface**

The second mode supported by the star tracker unit is file transfer mode. This mode is designed to transfer large quantities of binary data between the star tracker and host computer. It is used to download images from the star tracker or upload firmware and star map updates to the star tracker. To enter this mode, a file transfer must be first initiated from the command mode. Once the transfer is complete, the star tracker automatically returns to command mode. The transfer protocols used in this mode are YMODEM and YMODEM-G, two established variations of a protocol originally used to transfer data over networks.
Table 6.1: Star tracker driver requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANOE Compatibility</td>
<td>The star tracker driver shall be developed in such a way as to make it compatible with the CANOE operating environment, and the associated memory space and timing constraints.</td>
</tr>
<tr>
<td>LFFT Compatibility</td>
<td>The star tracker driver shall be developed in such a way as to make it compatible with the LFFT framework, to allow easy development of star tracker tests.</td>
</tr>
<tr>
<td>Command Mode</td>
<td>The star tracker driver shall support the command mode interface of the star tracker unit to allow for nominal operations.</td>
</tr>
<tr>
<td>File Transfer Mode</td>
<td>The star tracker driver shall support the file transfer mode interface of the star tracker unit to allow for star map and firmware updates as well as image downloads.</td>
</tr>
</tbody>
</table>

6.2 Driver Requirements

Based on the software modes and interfaces of the star tracker, as well as the scenarios in which it might operate, a list of requirements was compiled to assist in the driver development. These are described in Table 6.1.

6.3 Driver Architecture and Detailed Design

A top level architecture of the star tracker driver, addressing the requirements listed in Table 6.1 is presented in this section. A diagram of the driver as integrated into the CANOE operating environment is shown in Figure 6.2.

At the heart of the driver are three main blocks: the control block, YModem Receiver block, and YModem Transmitter block. These blocks contain the functional part of the driver. They communicate with the star tracker through the serial driver, which is accessed through the serial interface. In nominal operations commands are sent to the driver from the attitude determination and control thread through the driver API, and files are sent and received from the GNB flash file system through the in stream and out stream interfaces.

The control block receives commands through the driver API and dispatches them to the star tracker. If the received command is a file download or upload, the control module initializes and starts the YModem receiver or transmitter respectively. Responses from
the star tracker are also processed by this module. The length of star tracker responses and the latency at which they arrive can vary greatly. Simply waiting the maximum duration of time and assuming a full response has been received would be very inefficient, especially on a time critical system such as the attitude determination and control computer. Instead, a state machine that synchronizes on the termination string “SUCCESS” and “ERROR” was implemented. The state machine analyzes every incoming character and transitions based on its value and the previous state. For example, if an ‘E’ character arrives when in “STATE_IDLE” state, this could signify the beginning of the string “ERROR”, and the state is set to “STATE_E”. However, if the character ‘E’ arrives when in state “STATE_SUCCE”, this could signify the continuation of the string “SUCCESS” and the state is set to “STATE_SUCCE”. This process continues until a full termination string is received, or a timeout event occurs. The benefit of this approach is that it does not assume the length of a response ahead of time, which makes it very efficient.

The YModem Receiver and Transmitter blocks are required to download and upload files to the star tracker, respectively. They implement the YModem serial file transfer
protocol. Data sent to the star tracker is read from the In Stream interface and sent over
the serial driver. Data downloaded from the star tracker is received from the serial driver
and written out to the Out Stream interface. To avoid unnecessary reimplementation,
the YModem code was based on an open source library provided by [31]. As the origi-
nal library was written in C++, it had to be modified to exclude any object oriented
constructs.

One of the largest constraints on the attitude determination and control computer
is memory space. To reduce the amount of memory required by the star tracker driver,
a single shared memory buffer was implemented. This was possible as only one of the
main blocks can operate at a time. The control block uses this buffer to store responses
from the star tracker. The YModem blocks use the buffer as an intermediary location
for processing incoming and outgoing blocks of binary data, such as to perform CRC
calculations.

As the driver has to operate in both CANOE and the LFFT, it had to be designed in
a modular and flexible manner to account for their differences. In particular, the serial
drivers used by CANOE and the LFFT are different, and the GNB flash file system is not
present in the LFFT. Rather than creating multiple versions of the star tracker driver,
such that each was configured for the corresponding serial driver, interfaces were used.
The serial interface for example, consists of function pointers to functions that are ex-
pected to be part of any generic serial driver. When the star tracker driver is instantiated
in CANOE, these pointers are initialized with the functions of the CANOE serial driver.
When used in the LFFT, the LFFT serial driver functions are used. The star tracker
driver can then call these function pointers regardless of the environment it operates in.
A similar approach was taken for the in and out interfaces. This method proved instru-
mental in testing the driver under the Windows operating system, as described in the
following sections.

6.4 Driver Testing

At the time of the driver development, the star tracker was undergoing preliminary inte-
gration into the spacecraft. In the assembled configuration, the star tracker was connected
to the on-board computer, making a direct connection to it impossible. The lack of direct
access to the unit significantly complicated the development and testing process. This
section describes the innovative methods that were undertaken to achieve incremental
development and testing while overcoming the unavailability of direct physical access to the star tracker unit.

6.4.1 Stage One

The first stage of development and testing was conducted without any interfacing with the actual star tracker hardware. Instead, all testing was done under the Microsoft Windows operating system as shown in Figure 6.3. A wrapper framework was created to compile the driver under Windows, and to provide the means to interact with it. In place of the attitude determination and control thread, a user could enter commands to the driver through a Windows terminal program. The GNB flash file system was replaced by the Windows file system for reading and writing files. The CANOE serial driver was replaced by a Windows serial driver. The star tracker itself was simulated by another terminal application running on the same PC. A serial cable was used to connect the two serial ports of the driver and star tracker simulator. Under this configuration, the developer could enter commands into the driver terminal and verify the data sent to the star tracker in the star tracker terminal. The expected star tracker responses could then be manually entered into the star tracker terminal, and the drivers output verified through the driver terminal. Since the terminal application used to simulate the star tracker also had built in support for Y-Modem, files could be transferred back and forth testing the Y-Modem functionality of the driver. Working under Windows, the developer had access to the very powerful debugger application provided in Microsoft Visual Studio, which greatly simplified the debugging process.

6.4.2 Stage Two

Once it was verified that the star tracker driver worked flawlessly under the Windows environment, its performance with actual flight hardware had to be tested. As direct access to the star tracker was still not available, an “OBC pass-through” application was written for the on-board computer. This application received data on the on-board computer test port and forwards it to the serial port connected to the star tracker unit, and similarly passed data from the star tracker port back to the test port. With the star tracker driver still running on Windows as before, a serial cable was used to connect the PC to the test port on the on-board computer. A direct link between the driver and star tracker hardware was simulated with the pass-through application on the OBC. This
configuration is shown in Figure 6.4

6.4.3 Stage Three

Up to this stage the driver core was tested under Windows only. The next stage was to verify that the driver functioned as expected on the spacecraft on-board computer. In flight, the driver would be integrated into CANOE, however at the time of this work CANOE was not yet ready for testing and special measures had to be taken to control the driver on the spacecraft. Two embedded serial drivers were created: one to bridge the star tracker driver to the hardware unit, and the second to bridge the star tracker driver to the test port. A Windows terminal could then be used to control the driver and send and receive files over the test port, but a direct link between the driver and the star tracker hardware now existed. This configuration is shown in Figure 6.5.

6.5 Progress and Future Work

At this time the star tracker driver has been thoroughly tested using the methods described above. It remains to be integrated with the attitude determination and control system in CANOE, at which point further testing will be required to insure its proper
Figure 6.4: Star tracker driver under windows and OBC pass-through

Figure 6.5: Star tracker driver on OBC
operation in the CANOE environment, and to ascertain that its functionality meets the attitude system’s memory and timing constraints. This section has demonstrated how modular development and the use of interfaces allowed for the driver to be tested in a variety of configurations with only minimal support code and no modifications of the driver itself. This experience has shown that the quality of code can significantly benefit from being developed with the availability of sophisticated debugging tools and a highly controllable test environment. It is recommended that future embedded code development efforts are made as platform independent as possible, to allow similar testing and debugging methodologies to be utilized.
Chapter 7

Conclusions

The advent of the microspace approach has brought about a new era of capable spacecraft, developed at a fraction of the cost and time of those following the traditional approach. These achievements were in part possible due to advances in miniaturization of electronic components and increases in computational capacity and power efficiency. Advances in electronics have also enabled the use of off-the-shelf components for space application. With this methodology successfully demonstrated, there has been a growing demand for cheaper and rapidly developed spacecraft. This thesis has described a number projects relating to embedded systems development for SFL satellites and how they were approached within the constraints of the microspace philosophy.

The unique environmental conditions experienced in space and the effects they may have on electronic devices on-board a satellite were discussed. The thermal shock and thermal functional test procedures were described in detail in the context of unit level and system level testing, as well as from qualification and acceptance perspectives. Other environmental tests were also briefly mentioned for completeness. An extensive test framework was developed to perform system level functional testing on spacecraft hardware. The detailed design and implementation of the framework, and how it could be easily configured for a variety of test configurations were described.

A new scheduling scheme was proposed for SFL’s custom operating system, CANOE. By introducing priority levels to the scheduling algorithm, it was shown that a real-time system could be achieved for mission critical components. A discussion on the selection of data structures and search algorithms and how they affected the efficiency of the final solution was provided.

The most significant contribution described in this thesis is the design and implement-
tation of software for the BRITE mission. The bootloader, the most critical section of spacecraft software had to be tailored for the BRITE payload hardware. The application level software was designed by analyzing the top level mission operations, and three levels of autonomy were added for increased flexibility. Code for the CCD header board was written to provide hardware drivers as well as a control algorithm to maintain a constant CCD temperature.

Finally, a star tracker driver was designed and developed to dramatically increase the attitude determination and control capabilities on SFL satellites. It was demonstrated that object-oriented like design practices and ingenuity allowed development and testing to proceed even when physical access to the hardware was restricted, and how these practices were used to assist in debugging efforts.

Spacecraft design and development is an exciting discipline that is riddled with unique challenges. The inaccessibility of space due to the high costs and long development times is becoming an issue of the past due to continual efforts by SFL and other laboratories like it to advance the microspace approach and provide low-cost reliable and capable spacecraft. The efforts described in this thesis form an important contribution to this movement.
Appendix A

BRITE Instrument Software
Commands

This section describes all of the commands that were implemented for the BRITE instrument computer application software. These commands include the standard set of commands available on all GNB flight software, such as PING, INIT, PEEK, POKE, READ, WRITE, etc., as well as commands which are specific to the scientific operation of the BRITE payload.

• **IOBC_PING**: The PING command is used to verify the communication link with the spacecraft. The PING response contains information such as the spacecraft name, software version, and spacecraft time. It can be used to verify which spacecraft the ground station is communicating with, and the spacecraft time can be used to determine if a reset has occurred.

• **IOBC_INIT**: An INIT command directs the spacecraft to begin executing code at a particular memory location. As the spacecraft defaults into bootloader mode on power up, this command must be sent to start the operating system.

• **IOBC_PEEK**: A PEEK command is used to view on-board memory, when the length of the request does not exceed the maximum length of a single communication packet. In operation, this command is typically used to view the contents of variables for debugging or to determine the state of the system. It can also be used to read the contents of memory-mapped hardware registers and peripherals.
• **IOBC_Poke**: A POKE command is used to write to on-board memory, when
the length of the request does not exceed the maximum length of a single com-
munication packet. In operation, this command is typically used to change the
value of software variables, or to write to memory-mapped hardware registers and
peripherals.

• **IOBC_Read**: A READ command is used to view the contents of on-board mem-
ory in quantities larger than is supported by the PEEK command. Ground software
is tasked with verifying that all requested packets were received and re-requesting
the missing ones. The packets are then recombined into a contiguous segment of
data. This command is typically used to download payload data.

• **IOBC_Write**: A WRITE command is used to write to the on-board memory
in quantities larger than is supported with the POKE command. The ground soft-
ware is tasked with splitting data into packets, writing them to the spacecraft, and
verifying that the operation was successful with the IOBC_WRITE_QUERY com-
mand. It is typically used to upload application code and mission specific scripts.
Unlike a POKE, not every WRITE command generates an acknowledgement by
the spacecraft to minimize communications overhead.

• **IOBC_Write_Query**: A WRITE_QUERY command is used to verify that
WRITE packets were successfully received and processed by the spacecraft. A
write log is maintained on the spacecraft for every processed write command. This
log is returned in a WRITE_QUERY response, and can be used by ground software
to determine if any WRITE packets need to be resent.

• **IOBC_Telemetry**: A TELEMETRY command is used to request telemetry
points from the spacecraft. Most often, these points correspond to analog sensors
that measure voltages, currents, and temperatures. Telemetry points can also rep-
resent software-based statistics, such as the number of invalid received packets, etc.
Telemetry is used to verify the state of health of the spacecraft.

• **IOBC_TxSelect**: Each on-board computer has a number of channels through
which it can receive and transmit packets. While all channels are monitored for
incoming commands, only one at a time can be used to transmit a response. The
TXSELECT, or “transmitter select” command is used to specify which channel
Appendix A. BRITE Instrument Software Commands

should be used. Depending on the computer, the output channel may be assigned to each of the two internal serial ports, the test port, or the radio transmitter.

- **IOBC.I2C**: The I²C command allows data to be sent over the Inter-Integrated Circuit peripheral of the instrument computer. On BRITE, this peripheral is used to communicate with the CCD header board to query CCD header board telemetry and to command the CCD trim heaters.

- **IOBC.GFS.OPEN**: The GFS.OPEN command is used to open a file in the GNB flash file system. The command supplies the file name and an attribute flag that dictates whether a file is to be opened for read, write, or append.

- **IOBC.GFS.CLOSE**: This command is used to close opened files in the GNB flash file system that are no longer being used. Closing a file allows a different file to be opened, but most importantly it protects files from being accidentally overwritten.

- **IOBC.GFS.READ**: The GFS.READ command allows files to be read out of the GNB flash file system and into the on-board RAM. The command specifies the file name, RAM destination address, and number of bytes to read. Once in RAM, a regular READ command can be used to download the data to the ground. A file must be opened before it can be read.

- **IOBC.GFS.WRITE**: The GFS.WRITE command allows files to be written from on-board RAM into the GNB flash file system. The command specifies a file name, RAM source address, and the number of bytes to be written. Before the GFS.WRITE command is issued, data can be uploaded into RAM using a regular WRITE command. A file must be opened prior to executing a GFS.WRITE.

- **IOBC.GFS.DELETE**: The GFS.DELETE command removes a file from the GNB flash file system when free flash memory is low. The name of the file is specified as part of the command.

- **IOBC.GFS.FORMAT**: The GFS_FORMAT command removes all existing files and file system tables from flash memory and creates a new, empty table. It is necessary to perform this action once before the file system can be used, or if an existing file system was corrupted.
• **IOBC\_GFS\_INFO:** The GFS\_INFO command retrieves a listing of files from the GNB flash file system. The listing contains file sizes, date of creation, and whether they are opened or closed.

• **IOBC\_GPIO:** The GPIO command provides control of the general purpose input/output pins of the main processor. It allows for pins to be configured as inputs or outputs, to read the states of input pins, and to set the states of output pins.

• **IOBC\_PERIPHERAL\_READ:** The PERIPHERAL\_READ command is part of an interface to the three-wire-serial driver. It is used to read data from TWS devices.

• **IOBC\_PERIPHERAL\_WRITE:** The PERIPHERAL\_WRITE command is part of an interface to the three-wire-serial driver. It is used to write data to TWS devices.

• **IOBC\_TIMESYNC\_START:** The TIMESYNC\_START command is used to initiate a time synchronization sequence by transitioning the IOBC into the TIMESYNC state, as shown in Figure 5.11.

• **IOBC\_TIMESYNC:** Once in the TIMESYNC state, this command is used to complete a time synchronization sequence. It contains the current time the onboard clock should be set to.

• **IOBC\_OBSERVATION\_START:** The OBSERVATION\_START command directs the IOBC to start an observation sequence. The command contains the name of the observation upload file which contains information such as the observation mode and number of exposures to be taken.

• **IOBC\_OBSERVATION\_STOP:** An OBSERVATION\_STOP command is used to terminate the current observation cycle.

• **IOBC\_EXPOSURE\_SET:** The EXPOSURE\_SET command is used to configure the exposure time of the telescope.

• **IOBC\_EXPOSURE\_START:** When an observation is performed in manual mode, an EXPOSURE\_START command is sent to initiate each exposure in the observation.
• **IOBC.IMAGE.READ**: This command is implemented for debugging purposes and is not expected to be part of the nominal operations procedure. If necessary for debugging of the instrument, an exposure can be manually initiated with a series of POKE and GPIO commands. The IMAGE.READ command can then be used to download the image. Two image read modes are implemented. In the first mode, the image is first loaded into SRAM and then downloaded with the READ command. As the SRAM is not large enough to store the full image, this is done in small segments. In the second mode the image is saved directly into flash, and is then downloaded through the GNB flash file system commands.
Bibliography


