GROUND SEGMENT SOFTWARE DESIGN AND DEVELOPMENT FOR
NANOSATELLITE SPACE MISSIONS

By

Jin Hyouk (Paul) Choi

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 © 2013 by Jin Hyouk (Paul) Choi
Abstract

Ground Segment Software Design and Development for NanoSatellite Space Missions

Jin Hyouk (Paul) Choi
Master of Applied Science
Graduate Department of Aerospace Science and Engineering
University of Toronto
2013

For spacecraft development, realizing strong supporting ground segment software is as important as designing the actual hardware component of the spacecraft. This thesis describes the author's contributions to the ground segment software design and development for nanosatellite space missions at the UTIAS Space Flight Laboratory. Particular emphasis is given to the ground segment software for the CanX-3 and CanX-4/-5 missions. For the CanX-3 mission, several software applications are explored, specifically ground control software for the payload on-board computer and star tracker, and mission planning software. For the CanX-4/-5 mission, its mission monitor and control software, and whole orbit data parser are discussed. For each software application, design considerations and decisions made during the development are explained. Furthermore, detailed discussions on their architectural and graphical user interface design and implementation are presented.
Acknowledgements

First of all, I would like to thank Dr. Robert Zee for giving me the wonderful opportunity to work at University of Toronto’s Space Flight Laboratory (SFL). I have learned priceless lessons and experiences that I could not have obtained elsewhere. Also, thank you to all my friends and colleagues at SFL: Daniel for all the help and guidance throughout the entire period of my M.A.Sc. program and both Cordell Grant and Grant Bonin as my project managers for the BRITE and CanX-4/-5 missions, respectively. And Vincent Tarantini and Laura Bradbury, thank for all of your valuable time spent for me. Last but certainly not least, thank to my loving parents and little sister. Without you and your support, I would have never amounted to anything of what I am today.
Table of Contents

Chapter 1 Introduction ................................................................. 1
  1.1 The CanX-3: BRIght-star Target Explorer ............................... 2
  1.2 CanX-4/-5: Formation Flying Demonstration ......................... 3
  1.3 Ground Segment Software .................................................... 4
  1.4 CanX-2 and Real On-orbit Operation Experience ..................... 5

Chapter 2 BRITE Payload Ground Control ................................. 7
  2.1 Software Requirements ........................................................ 9
  2.2 Software Architectural Design .............................................. 10
    2.2.1 Data Packet Flow Control ............................................. 11
    2.2.2 Task Management ...................................................... 11
  2.3 Software User Interface Design & Implementation .................. 12
    2.3.1 IOBC External Command Interface .................................. 13
    2.3.2 Imaging Operations and Controls .................................... 14
    2.3.3 Telemetry Data Manipulation ....................................... 16
    2.3.4 Peripheral Device Control .......................................... 17
    2.3.5 CCD Header Board & Control ...................................... 19

Chapter 3 BRITE Schedule ......................................................... 22
  3.1 Software Requirements ...................................................... 22
  3.2 Observation Planning, BRITE Target, and Setup File ............ 23
  3.3 TimeTag Scripts and TimeTag Uploader ............................... 23
  3.4 BRITE Schedule Design Considerations and Decisions .......... 24
    3.4.1 Spacecraft-Specific Operation Sequence for Target Observation 24
    3.4.2 Limitations on TimeTag Operation and Scripts .................. 26
    3.4.3 Generic TimeTag Command Operation Scheduler ................ 27
  3.5 Software Architecture Design & Implementation .................. 27
    3.5.1 Schedule Manager ..................................................... 28
    3.5.2 Command Creator .................................................... 29
    3.5.3 BRITE Command Module .......................................... 29
      3.5.3.1 Inheritance and Command Library .......................... 30
      3.5.3.2 Command Managers ............................................ 31
  3.6 Software Interface Design & Implementation ....................... 33
    3.6.1 Schedule Manager ..................................................... 34
    3.6.2 Command Creator .................................................... 35
List of Tables

Table 2.1: Additional BRITE PGC requirements. ................................................................. 9
Table 3.1: Primary BRITE Schedule Software Requirements.............................................. 23
Table 3.2: Required Actions of On-board Computers for a Target Observation. ............... 25
Table 4.1: STACI Software Requirements........................................................................... 42
Table 5.1: FFUserInterface Software Requirements............................................................ 61
Table 5.2: Variables in CFFExchangeStructureElement class and descriptions. ............. 67
Table 6.1: FFChop software requirements............................................................................ 77
Table 6.2: Settings in the output data configuration and descriptions. .............................. 81
List of Figures

Figure 1.1: Solid Model of the CanX-3 Nanosatellite [3]. ................................................................. 2
Figure 1.2: Solid Model of the CanX-4/-5 Nanosatellite [7]. .............................................................. 3
Figure 1.3: Simplified Ground Segment Architecture for Different Communication Types .............. 5
Figure 1.4: Solid Model of the CanX-2 Nanosatellite [14]. .............................................................. 6
Figure 2.1: BRITE Instrument: Telescope on the left and IOBC on the right. ..................................... 8
Figure 2.2: Screen capture of the initial version of BRITE PGC. ..................................................... 8
Figure 2.3: Simplified software architecture diagram for BRITE PGC .......................................... 10
Figure 2.4: Main application dialog design for BRITE PGC. .......................................................... 13
Figure 2.5: Manual command tab user interface with GPIO command selected. .............................. 14
Figure 2.6: Imaging operation user interface. ................................................................................... 15
Figure 2.7: Telemetry operation user interface showing the available A/D ICs. ............................... 17
Figure 2.8: Peripheral operation user interface showing the available A/D ICs. .............................. 18
Figure 2.9: Header board operation user interface with INIT_POINTER_FILE selected ................. 19
Figure 2.10: BRITE Header board application code uploader dialog with a sample code loaded ...... 20
Figure 3.1: Single Target Observation Sequence [24]. .................................................................... 25
Figure 3.2: Top-level architectural implementation for BRITE Schedule. ...................................... 28
Figure 3.3: Block diagram for BRITE command module. ............................................................... 30
Figure 3.4: An example of a BRITE Schedule file. ......................................................................... 32
Figure 3.5: BRITE Schedule’s root dialog design. .......................................................................... 33
Figure 3.6: Allowed clipboard operations through main menu and their accelerator-keys. ............. 34
Figure 3.7: Allowed clipboard operations through a context menu. ............................................. 34
Figure 3.8: BRITE Schedule menu for file input and output operations. ....................................... 35
Figure 3.9: BRITE Schedule menu for functional modules. .......................................................... 35
Figure 3.10: Command creator showing available commands for the selected OBC and software mode. .............................................................................................................................................. 36
Figure 3.11: Command creator showing a list of available options for a pin parameter. .................. 36
Figure 3.12: Command creator showing behavior of a cell with an invalid input. ........................... 37
Figure 3.13: The sequence of stages within the Target Operation Scheduler. ................................. 37
Figure 3.14: Target Field Selection user interface during the target observation scheduling operation. .................................................................................................................................................. 38
Figure 3.15: Observation Time Selection user interface during target observation scheduling operation. .................................................................................................................................................. 38
Figure 3.16: Confirmation and Generation interface during target observation scheduling operation. 39
Figure 3.17: Observation Settings user interface during target observation scheduling operation ...... 40
Figure 4.1: S3S Star Tracker from Sinclair Interplanetary. .......................................................... 42
Figure 4.2: Top-level software architecture for STACI ................................................................. 44
Figure 4.3: Simplified class relationship diagram for S3S Message handler module................................. 45
Figure 4.4: A flow chart of processing response packets from S3S star tracker. ................................. 46
Figure 4.5: Class diagram for the operational block and the corresponding task handlers. ................... 47
Figure 4.6: Screen capture of the main application dialog for STACI. .................................................. 48
Figure 4.7: Connection control user interface for BRITE PGC (left) and STACI (right). ....................... 49
Figure 4.8: Options user interface ........................................................................................................... 50
Figure 4.9: Manual commanding operation user interface (left) with its command selection list unfolded (right)......................................................................................................... 51
Figure 4.10: Built-in self-test operation user interface............................................................................. 52
Figure 4.11: Flow diagram for the self-test operation............................................................................. 52
Figure 4.12: Attitude determination operation user interface. ............................................................. 53
Figure 4.13: An example of ACS result file showing quaternion and angular velocity measurements. 54
Figure 4.14: Control-structure manipulation operation user interface. .............................................. 55
Figure 4.15: Flow diagram of the downloading operation for the EDAC memory and RAM............... 56
Figure 4.16: Flow diagram of the downloading operation for the NAND flash memory....................... 56
Figure 4.17: Data downloading operation user interface. ....................................................................... 57
Figure 4.18: Flow diagram of the writing process for the RAM.......................................................... 58
Figure 4.19: Flow diagram of the writing process for the NAND flash memory................................... 58
Figure 4.20: Firmware update operation user interface.......................................................................... 59
Figure 5.1: Command structure for GetExchange and SetExchange commands. .............................. 59
Figure 5.2: Top-level software architecture for FFUserInterface .......................................................... 65
Figure 5.3: Screen capture of the graphical user interface for FFUserInterface. ................................. 70
Figure 5.4: Listing user interface showing all updated exchange parameters in Fiona_output group. 71
Figure 5.5: FFExchange Parameter Editor with exchange parameter (DeputyAttitude) selected and showing its detailed information......................................................................................... 73
Figure 5.6: Safeguard user interface added as part of SetExchange operation.................................... 74
Figure 5.7: GPS operation user interface............................................................................................. 74
Figure 6.1: Data structure for a WOD group within WOD file. ............................................................. 78
Figure 6.2: Manual parser user interface with a raw WOD file loaded for parsing.............................. 81
Figure 6.3: Automatic parser user interface with its operation enabled.............................................. 82
Figure 6.4: ChoppingBoard showing input data structure of S-band WOD group in loaded sample file. ........................................................................................................................................ 84
Figure 6.5: ChoppingBoard showing output data structure of S-band WOD group in loaded sample file. ........................................................................................................................................ 85
**Acronyms and Abbreviations**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/D</td>
<td>Analog-to-Digital</td>
</tr>
<tr>
<td>ADCC</td>
<td>Attitude Determination and Control Computer</td>
</tr>
<tr>
<td>ADCS</td>
<td>Attitude Determination and Control System</td>
</tr>
<tr>
<td>ADU</td>
<td>Analog-to-Digital Units</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>BRITE</td>
<td>Bright Target Explorer</td>
</tr>
<tr>
<td>CanX</td>
<td>Canadian Advanced Nanospace eXperiment</td>
</tr>
<tr>
<td>CCD</td>
<td>Charge-Coupled Device</td>
</tr>
<tr>
<td>CNAPS</td>
<td>Canadian Nanosatellite Advanced Propulsion System</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial-Off-The-Shelf</td>
</tr>
<tr>
<td>CTAP</td>
<td>Coarse Three-Axis-Pointing</td>
</tr>
<tr>
<td>EDAC</td>
<td>Error Detection And Correction</td>
</tr>
<tr>
<td>FF</td>
<td>Formation Flying</td>
</tr>
<tr>
<td>FIONA</td>
<td>Formation flying Integrated On-board Nanosatellite Algorithm</td>
</tr>
<tr>
<td>FTAP</td>
<td>Fine Three-Axis-Pointing</td>
</tr>
<tr>
<td>GNB</td>
<td>Generic Nanosatellite Bus</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HKC</td>
<td>House Keeping Computer</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ICD</td>
<td>Interface Control Document</td>
</tr>
<tr>
<td>IOBC</td>
<td>Instrument On-Board Computer</td>
</tr>
<tr>
<td>ISL</td>
<td>Inter-Satellite Link</td>
</tr>
<tr>
<td>ISS</td>
<td>Inter-satellite Separation System</td>
</tr>
<tr>
<td>ITAR</td>
<td>International Traffic in Arms Regulations</td>
</tr>
<tr>
<td>KISS</td>
<td>Keep-It Simple and Straight</td>
</tr>
<tr>
<td>NSP</td>
<td>Nano-Satellite Protocol</td>
</tr>
<tr>
<td>OBC</td>
<td>On-Board Computer</td>
</tr>
<tr>
<td>PGC</td>
<td>Payload Ground Control</td>
</tr>
<tr>
<td>POBC</td>
<td>Payload On-Board Computer</td>
</tr>
<tr>
<td>RelNav</td>
<td>Relative Navigation algorithm</td>
</tr>
<tr>
<td>STACI</td>
<td>StarTracker All-in-one Control Interface</td>
</tr>
<tr>
<td>SDI</td>
<td>Single Document Interface</td>
</tr>
<tr>
<td>SFL</td>
<td>Space Flight Laboratory</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol / Internet Protocol</td>
</tr>
<tr>
<td>TIP</td>
<td>Terminal Interface Program</td>
</tr>
<tr>
<td>TNC</td>
<td>Terminal Node Controller</td>
</tr>
<tr>
<td>UTIAS</td>
<td>University of Toronto Institute for Aerospace Studies</td>
</tr>
<tr>
<td>WOD</td>
<td>Whole Orbit Data</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Space Flight Laboratory (SFL) at the University of Toronto Institute for Aerospace Studies (UTIAS) is a world-recognized research and development facility for small micro- and nano-class satellites. At SFL, a microspace philosophy is adopted, which means that wherever possible commercial-off-the-shelf (COTS) electronic components are used in order to keep the cost and lead times down. As electronics technology evolves and advances, COTS components have become substantially mature, reliable, and widely available. The viability of using state-of-the-art COTS components on complex and sophisticated spacecraft has been proven through a number of successful missions at SFL including MOST, CanX-2, NTS, and AISSat-1 [1].

Notably, a significant principle of the microspace philosophy [2] is to reduce the time and cost that are associated with spacecraft development. At SFL, a period of two years is typically expected for the complete spacecraft development cycle (including preliminary and detailed designs; manufacture, integration, and test; launch; and on-orbit operation). During this approximate two-year cycle, graduate students at SFL are given the opportunity to acquire valuable experience in the development of real spacecraft by working alongside full-time staff members.

For spacecraft development, realizing strong supporting ground segment software is as important as designing the actual spacecraft hardware. Ground software is required for testing and debugging purposes and for the successful commissioning and nominal operations of spacecraft on-orbit. This thesis describes the author’s contributions to the ground segment software design and development
for nanosatellite space missions at the Space Flight Laboratory. In particular, focus is placed on the
ground segment software for the CanX-3 and CanX-4/-5 missions. For the CanX-3 mission, the
payload control software (Chapter 2), mission planning (Chapter 3), and attitude-determination
device control (Chapter 4) are explored. For the CanX-4/-5 mission, mission monitor and control
software (Chapter 5) along with whole orbit data parser (Chapter 6) are discussed. In the following
sections, brief overviews of the CanX-3 and CanX-4/-5 missions are given along with a description of
the current SFL ground segment software designs to provide the background information required to
better understand the designs in the succeeding chapters.

1.1 The CanX-3: BRIght-star Target Explorer

As part of the Canadian Advanced Nanospace eXperiment (CanX) program [4] at the Space Flight
Laboratory (SFL), the CanX-3 mission has been developed and is currently awaiting the start of its
journey into space. The spacecraft, shown in Figure 1.1, uses the SFL Generic Nanosatellite Bus
(GNB) [5] and will perform on-orbit photometric star observations using a collection of six
nanosatellites. Throughout the mission, the satellites will make periodic observations of some of
the brightest stars in the sky in order to examine their brightness and temperature variation over time.
They will do this as frequently as possible. Because of its very specific objective, the CanX-3 mission
has also been named the BRIght Target Explorer (BRITE). The primary payload of the CanX-3 (or
BRITE) spacecraft is an SFL-developed optical instrument (telescope). The BRITE instrument uses a
10 megapixel monochrome charge-coupled device (CCD) optic sensor to capture photons from the
target stars in visible light spectrum. The imager’s exposure time is precisely controllable between the range of 100 milliseconds and 100 seconds with 0.01% tolerance. Originally, the BRITE project was initiated as a single nanosatellite mission. However, since its conception, the mission has drawn remarkable attention from several scientific communities around the world and caused the project to become a constellation of six nanosatellites, two from each of Austria, Poland, and Canada. The main difference between these versions of spacecraft is the optical configurations for the instrument that allow the capture of light in different spectral bands. Each pair of BRITE spacecraft consists of one that is fine-tuned for red light and one that is tuned for blue. The satellites are otherwise identical to each other. More detailed information on the BRITE mission and its instrument design can be found in [3] and [6], respectively.

1.2 CanX-4/-5: Formation Flying Demonstration

The primary objective of the CanX-4/-5 mission is the demonstration of on-orbit autonomous formation flying. For this demonstration, twin GNB nanosatellites (Figure 1.2), referred to as the chief and the deputy, are launched together and will perform four different formation flying configurations: 1000 m and 500 m along track orbits (ATO), and 100 m and 50 m projected circular orbits (PCO). During formation flying, several different sub-systems designed at SFL will be demonstrated and evaluated, including the Intersatellite Separation System (ISS) [8], the Inter-Satellite Link (ISL) [9] for on-orbit communication between spacecraft, and the Canadian Nanosatellite Advanced Propulsion System (CNAPS) [10] for maneuvering. Furthermore, to
accomplish autonomous formation flying, an advanced formation flying control algorithm (FIONA) [11] has also been developed in house at SFL and will be implemented in the CanX-4/-5 mission. The FIONA algorithm continuously calculates control solutions using relative position determined using the information from GPS receivers on both the chief and deputy spacecraft during formation flying. More information on the CanX-4/-5 mission can be found in [7].

1.3 Ground Segment Software

Ground segment software has been developed at UTIAS/SFL to support the various CanX missions. They have been carefully designed and implemented to fulfill their respective mission requirements. The ground segment software can be categorized into three groups according to their role and function as follows:

- **Control Software**: Software that is used to control spacecraft (or sub-systems of spacecraft). Control is accomplished by issuing the appropriate commands manually in real-time. The software is capable of autonomous operations, but it is performed under the operator’s supervision.

- **Planning Software**: Software that is used to generate the next set of instructions for a spacecraft. Planning software does not typically pass the next task to spacecraft. Instead, script files that contain a set of tasks are generated and passed to different software that takes care of the uploading process. The script file generation can be performed either autonomously using a predefined algorithm or manually by the operators.

- **Monitoring Software**: Software that is used to monitor parameters (i.e., telemetry) within the spacecraft (or a sub-system of the spacecraft). Monitoring software is capable of performing tasks autonomously without the presence of operators (i.e., during a spacecraft pass) and automatically sends the appropriate reports. A single function reduces the complexity of the software, resulting in increased reliability of the software throughout the mission.

The communication links between components of the ground segment software and between the ground segment software and on-board computers of the spacecraft are established using the NanoSatellite Protocol (NSP). NSP is a specially designed, simple, serial end-to-end communication protocol used at SFL for CanX nanosatellites. Currently, it has evolved into its fourth version. However, its third version is being used as a baseline for GNB based nanosatellite missions (i.e., CanX-3 and CanX-4/-5) because it was the most recent version when the projects were initiated. In
the context of this thesis, all commands and responses refer to data packets that are constructed under the third version of NSP, unless they are explicitly specified otherwise. More details about NSP can be found in [12].

Lastly, the ground segment software that serves the most important role is called the Terminal Interface Program (TIP). TIP acts as a communication gateway to spacecraft or NSP devices (devices that use NSP for communication). It allows other ground segment software to connect to it via the Transmission Control Protocol/Internet Protocol (TCP/IP) and acts as a network switch by routing NSP packets from clients to the appropriate destinations. For communication over the radio link (i.e., on-orbit operation), the packets are routed through the CanX Terminal Node Controller (TNC) to the ground station hardware (i.e., amplifier and antenna). For ground operations (i.e., testing and debugging), TIP delivers the packets to a target via a hardwired serial connection (test port). Hence, all commanding operations (sending commands and receiving responses) from ground segment software to NSP devices require TIP’s support even though it may not be mentioned explicitly. More information on TIP can be found in [13]. The diagram in Figure 1.3 illustrates the described ground segment software architecture for communications.

![Figure 1.3: Simplified Ground Segment Architecture for Different Communication Types.](image)

### 1.4 CanX-2 and Real On-orbit Operation Experience

CanX-2 is the second nanosatellite developed as part of CanX program at UTIAS/SFL. The main objective of the CanX-2 mission was to demonstrate formation flying technology that would be employed on the CanX-4 and CanX-5 missions. CanX-2 is a 30 cm x 10 cm x 10 cm spacecraft
weighing 3.5 kg. It carries a number of scientific experimental payloads from different university research groups across Canada including an atmospheric spectrometer for a pollution monitoring experiment, a COTS GPS receiver for radio occultation experiments, and a material sample for an atomic oxygen degradation experiment. On 28 April 2008, CanX-2 was successfully launched and became the first operational nanosatellite of the CanX program. It is still operational and is currently generating valuable on-orbit data for future nanosatellite missions. A solid model of CanX-2 spacecraft is shown in Figure 1.4. More information on the CanX-2 mission can be found in [14].

![Solid Model of the CanX-2 Nanosatellite](image-url)

**Figure 1.4:** Solid Model of the CanX-2 Nanosatellite [14].

Operation of CanX-2 is performed explicitly at the UTIAS/SFL ground station. The operation team is comprised of four to six members (depending on student and staffing levels). Satellite operations are scheduled into morning, evening, or weekend shifts with the operators rotating between the shifts on a weekly basis. Moreover, the team is composed of a combination of experienced staff members and masters students. Students are selected according to the relevance of their thesis work and are fully exposed to the real on-orbit satellite operation. During the period of Master’s study at SFL, the author was responsible for operating the CanX-2 satellite and thereby acquired extensive knowledge and experience that directly guided the development of the ground software described in this thesis. Specifically, the user interface designs were significantly influenced by the desire to improve their intuitiveness. All software was designed in such a way to be operator and operation-oriented.
Chapter 2

BRITE Payload Ground Control

The BRITE Payload Ground Control (PGC) is one type of ground segment software developed for the BRITE mission. It is responsible for controlling the instrument on the BRITE spacecraft. Strictly speaking, the BRITE PGC provides the user interfaces that are used to communicate with the instrument-on-board-computer (IOBC) on the BRITE spacecraft. The IOBC (shown in Figure 2.1) controls and allows operators to control the detector (also shown in Figure 2.1). The IOBC defines a set of 29 external commands [15] that can be sent through its universal asynchronous receiver and transmitter (UART) serial port that allows external components (such as the other on-board-computers or ground segment software) to have complete control over the BRITE detector. All external control commands sent to the IOBC conform to the NanoSatellite Protocol (NSP) format [12] and are encoded using the Keep-It Simple and Straight (KISS) frame [12].

At the time when the author started developing the BRITE PGC, the first iteration of the development had already been performed. The initial design objective was to provide a quick ground control interface for debugging and testing purposes. The application was already capable of dealing with a limited number of the IOBC external commands and of communicating with the IOBC in order to download images captured by the imager. However, as the development of the BRITE mission approaches its completion, IOBC debugging and testing operations have become more complex than those in the earlier development phase. The initial BRITE PGC design proved to be too primitive and inconvenient to handle those complex operations. As a result, additional functionaly
was required to be implemented into the BRITE PGC to make the interface more advanced, convenient, and flexible. A screen capture of the original BRITE PGC is presented in Figure 2.2.

Figure 2.1: BRITE Instrument: Telescope on the left and IOBC on the right.

Figure 2.2: Screen capture of the initial version of BRITE PGC.
2.1 Software Requirements

At the beginning of the BRITE PGC development process, a set of primitive application requirements were defined. In summary, the initial set of requirements defines the BRITE PGC as an essential ground control application to be used for the purpose of the BRITE IOBC debugging and testing. Therefore, a set of new requirements needed to be defined in order to provide the advanced functionality of the application. Those new requirements are defined along with the original ones in [16] and also shown below in Table 2.1.

| BP 6.2.4 | BRITE PGC shall be able to create a set of tasks that can be performed autonomously. BRITE PGC should not require any human intervention to complete the set of tasks. |
| BP 6.2.5 | BRITE PGC shall be capable of saving and reloading a set of tasks created by the operators. BRITE PGC shall be capable of adjusting and verifying the created set of tasks. |
| BP 6.2.6 | BRITE PGC shall be capable of uploading the BRITE CCD header board application code during flight. BRITE PGC should perform basic quality scans of the application code before uploading it to the header board. |
| BP 6.2.7 | BRITE PGC shall be capable of accessing the values of control files in BRITE CCD header board. |
| BP 6.2.8 | BRITE PGC shall allow a task to be paused or canceled. |
| BP 6.2.9 | BRITE PGC shall provide visual indicators as to the progress of a current task. |

Table 2.1: Additional BRITE PGC requirements.

The original motivation behind requirement BP 6.2.4 was to develop the interface to make the BRITE instrument test procedure automated since a part of its procedure requires it to capture and download multiple images – taking up to 20 minutes to capture and download a single image. For example, a task of receiving 20 images from the instrument would take about 7 hours and an operator must be present at the test bench during the testing. However, if the task can be done in an autonomous manner, the operator only needs to set up the commands and let it run overnight. This makes the process much more efficient. Later on, the idea was extended to general operations to
provide a more user-friendly system to the operators. The requirements BP 6.2.5, BP 6.2.8 and BP 6.2.9 have all been defined to give the operators flexibility during the management of tasks.

Requirement BP 6.2.7 has been defined to ensure that BRITE PGC can update the BRITE CCD header board’s application code while the spacecraft is on-orbit. There was no existing ground segment software that supported this for flight. Therefore, it was deemed appropriate to integrate this functionality into BRITE PGC since the CCD header board is a part of the BRITE instrument.

### 2.2 Software Architectural Design

As previously mentioned, the first iteration of the BRITE PGC development was relatively primitive. The original architectural implementation was discovered to be inadequate and thus could not be continuously used for further development phases. This was primarily due to its inherent lack of flexibility. As a result, the entire implementation was modified to integrate the new required functionality into BRITE PGC. Figure 2.3 has been prepared to show a simplified new software architecture diagram of the application.

![Software Architecture Diagram](image)

Figure 2.3: Simplified software architecture diagram for BRITE PGC.
2.2.1 Data Packet Flow Control

The major modification to the first version of the BRITE PGC was the flow of command packets within the application. In the original design, all the command packets and their corresponding responses from the IOBC were created and handled within the main application class. This method is advantageous if the application has a single purpose because the developer does not need to worry about passing variables between classes since they can be easily accessed within the application. Furthermore, none of the command packet handling routines needed to be accessed more than once for operations since the application did not have many events to handle. However, as the scope of the application became wider and more complex, many command packet handling routines were required to be accessed more than once during the operation, and those handling routines needed to be rearranged to generalize their use so that those routines could be repeatedly accessed by the application for all appropriate occasions. As a result, a new CommandHandler class has been implemented into BRITE PGC to complement the main application class to handle all necessary commands and their corresponding responses for the BRITE mission.

The CommandHandler class is made up of three major groups of functions: command packet preparation functions, command interpretation functions, and packet response functions. As the name implies, the command packet preparation functions are responsible for creating command packets in NSP format. The command interpretation functions are the set of functions that return the detailed information of the requested command in string form. Finally, the packet response functions decode the returned response packets from the spacecraft and pass the decoded data to the main application for further handling.

2.2.2 Task Management

To provide task management functionality for BRITE PGC for the purpose of satisfying the functional requirements (BP 6.2.4 and BP 6.2.5), the TaskManager class has also been implemented in parallel with the CommandHandler class. For its operation, the TaskManager class accepts tasks from the main application class and stores them into queue container objects within the class. Those tasks that pass from the main application are tagged with their priority level – high or normal – and stored into two different queue containers depending on their priority specification. This specification is important because it determines the way the TaskManager distinguishes instantaneous control.
tasks from autonomous tasks during the operation. Once any one of the task queues is filled with new tasks, the TaskManager accesses and executes the tasks in the normal priority queue as long as the main application class has already requested to execute the tasks. Otherwise, the tasks in the queues are held for the further manipulation. The tasks that are prioritized as high are executed right after they are added into the queue. Furthermore, the normal priority tasks in the queue can be saved into a file in such a way that they can later be accessed and loaded into the queue. This feature allows the operators to schedule and manage a sequential set of commands before the spacecraft is actually available for communication or while the spacecraft is busy executing on-going tasks – e.g., downloading a captured image. In addition, high priority tasks are always executed before the normal priority tasks if both queues are filled.

Furthermore, TaskManager has been implemented in such a way that it sits between the main application class and the previously described CommandHandler class. The TaskManager works as a broker between the Communication class and the CommandHandler to pass the prepared command packets and returned command responses back and forth within the application, as shown in Figure 2.3. Therefore, using such architecture, the main application class simply needs to create and add a task to the TaskManager in order to send a command to the IOBC.

2.3 Software User Interface Design & Implementation

As the functionality of the BRITE PGC grew during development, the handling of all the operations using a single dialog design became impossible and the original graphical user interface (GUI) design had to be modified. As a solution, all possible operations have been divided into five different groups based on their usage: manual command control, image control, telemetry control, peripheral device control, and header board control. Individual user interface dialogs for those operation groups were then implemented and imported into a tab control in the main application dialog as shown in Figure 2.4. Moreover, the connection control interface and the logging control interface have been placed in the main application dialog since they are commonly used components.
2.3.1 IOBC External Command Interface

To issue and handle the available IOBC external commands listed in the interface control document [15] as part of the requirements, a CManualCommandControlDlg class has been implemented and added into the main BRITE PGC design as one of the tab control members (with the label Manual Command). For the operation, the CManualCommandControlDlg class lists all supported IOBC commands in a list box to provide one-click access to operators. However, the most innovative part of the CManualCommandControlDlg class takes place when an operator selects an IOBC command in the list box – the class itself rearranges the user interface control objects and provides the possible argument data selections accordingly for the selected IOBC command. The intent of this assisted-command-selection function is to minimize the errors that can be caused by operators during operations. For example, Figure 2.5 shows the GPIO_WRITE command with its three arguments selected in the manual command tab. In addition, the assistant argument data selections are pre-defined and hardcoded in the source code.
Another important feature of the CManualCommandControlDlg class is that it provides an interface to add commands into the task queue for an autonomous operation, as discussed in Section 2.2.2. With the *Add to Queue List* check box set as seen in Figure 2.5, all commands will be appended in the normal priority task queue or will be transmitted to the IOBC right away if the check box is unset. By taking advantage of the CommandHandler class, sending or adding IOBC commands can be done in a simple manner. When such an event takes place, the CManualCommandControlDlg class only needs to create a BRITE command object with specified values in the manual control tab before it passes the object to the main application class. All further operations are done by the TaskManager class, and the result is then reported back to the CManualCommandControlDlg class as an event.

### 2.3.2 Imaging Operations and Controls

In the original BRITE PGC design, downloading a captured image from an IOBC was the only allowed imaging operation. All other operations that needed to be performed prior to or after the downloading process were executed individually using the manual command interface. Those extra operations were meant to initialize and setup hardware in the IOBC for an observation. Performing all of those operations manually every time when an observation needed to be made was extremely
inconvenient and inefficient. As a result, a user interface dedicated to the imaging operation was required and needed to be implemented into the new BRITE PGC design.

The Imaging Operation Interface, shown in Figure 2.6, has been developed in the CImageControlDlg class and was added into the tab control in the main application. During the development of the interface, an effort was made to identify all possible required tasks for the imaging operation. Once they were identified, all necessary controls for the imaging operation interface could be defined and the interface could be designed as a stand-alone module. Overall, five task groups of imaging operation were identified: power control, pointer control, exposure time control, exposure and download.

![Image Control Interface](image.png)

Figure 2.6: Imaging operation user interface.

In the IOBC, there are three power switches that need to be controlled during the imaging operation: the bias switch, CCD switch, and amplifier switch. To make a successful observation, all power switches must be turned on in the appropriate order and must be turned off after the observation is finished. Control of those switch states is achieved through a control register in the IOBC by poking appropriate values and in an opposite manner. Their states can be read back by peeking the appropriate control register. All the power switch control operations have been grouped as a power control task group. A control interface for them has been placed in the top-left side of the image control tab under the *Power Switches* section. By simply clicking the *Get Current State* button,
the state of all power switches are retrieved from the IOBC and the state of the Bias, CCD, and Amplifier buttons is updated accordingly. A button stays pressed when the corresponding power switch is on and unpressed when it is off. To toggle the switch states, the appropriate button needs only to be clicked on.

2.3.3 Telemetry Data Manipulation

To allow external devices to retrieve various telemetry points in the BR ITE instrument, the IOBC provides a command dedicated for the telemetry retrieving operation, namely IOBC_TELEMETRY. In the telemetry command packet, an identification number of an analog-to-digital (A/D) integrated circuit (IC) that has been placed in the IOBC hardware design is specified along with a 16-bit wide flag that indicates the applicable channels on the IC. When the IOBC receives a valid telemetry command, it replies with the corresponding telemetry reading that is in raw analog-to-digital units (ADU). Since those ADU readings are not very useful until they are converted to values in engineering units, an extra step is performed to make the appropriate conversions for further operations.

In general, the telemetry retrieval operation can be done using the manual command interface introduced in Section 2.3.1. However, since the manual command interface does not provide any method to post-process the data appearing in the response packets, all the time-expensive unit conversion tasks need to be performed manually by the operator. This process is inefficient and inconvenient. For long-term ground operation, an interface that does the conversion task in an autonomous manner is desired. As a solution, a dedicated telemetry data manipulation interface – CTelemetryControlDlg class – was developed as a part of the BRITE PGC design.

As illustrated in Figure 2.7, the majority of the space in the interface is occupied by a list control object in the middle of the Telemetry tab dialog. This list control is used as both input and output interfaces for most of the telemetry data manipulation operations. The list control displays a list of all collectable telemetry from the selected A/D IC in the device section. To get the telemetry readings, the Get Selected button simply needs to be clicked after selecting one or more desired telemetry points from the list control. Alternatively, the Get All button can be used to retrieve all telemetry readings in the list. Once the readings are retrieved from the IOBC successfully, the appropriate
numbers in the Value column are updated accordingly. If the Show Raw Value check box is set, the raw ADU counts are displayed. Otherwise, the converted values are displayed.

For the unit conversion process, a calculation algorithm has been implemented in such a way that the raw readings can be converted into engineering units in a flexible manner. The algorithm provides four different mathematical operations that are commonly used for telemetry conversion and allows operators to configure the type and coefficients of the mathematical operation for all telemetry points present in the list individually. Furthermore, during development, it was found that the number of configurations for all available telemetry was significant and it was unpleasant and inconvenient to transfer them between different ground station computers (particularly for testing and debugging purposes). Therefore, an implementation to allow the application to export and import the configuration has been employed into the design to improve convenience.

2.3.4 Peripheral Device Control

Among the electrical components comprising the BRITE instrument, there are two integrated-circuit (IC) chipsets that are defined as peripheral devices which require special controls in order to perform imaging operations. Each IC interacts directly with the optic sensor in the imager to maintain the desired performance and output during imaging operations. The behavior of the chipsets is defined in
their respective control registers. Therefore, to maintain precise control over imaging operations, a user interface (shown in Figure 2.8) has been implemented into the new BRITE PGC design that is dedicated to allow the user to manipulate the register values in both chipsets. The detailed information on the chipsets can be found in [17] and [18].

Figure 2.8: Peripheral operation user interface showing the available A/D ICs.

Similar to the telemetry retrieving operation, the IOBC provides specific commands for the manipulation of control register values via the PERIPHERAL_READ and PERIPHERAL_WRITE commands. For both commands, the identification number of the chipset along with the address and size of the considered register are specified within the command packet. If there is no additional data appended to the command, it is interpreted as the read command and the values at the target register are retrieved. However, if additional data are appended at the end of the packet, it is treated as the write command and the data is written into the target register.

In the user interface for the peripheral control operation, the details about the chipsets (i.e., address and size of the control registers) are not shown. Only the name of the chipsets and the registers to which they belong are exposed to operators for their control. Moreover, instead of displaying raw register values, a list control has been placed in the user interface to show the structure and parsed values of the selected control registers with the appropriate descriptions.
### 2.3.5 CCD Header Board & Control

The CCD Header board, located at the rear of the BRITE Imaging instrument assembly, serves as the mounting point for the Charge-Coupled Device (CCD) optical sensor and its electrical connections. Also, it contains a thermal control system designed to maintain the temperature of the CCD at the desired level during imaging operations. This thermal control is managed by a microcontroller and, similar to the IOBC, the header board controller defines a set of commands for its thermal management. In total, there are seven different commands [19] that are sent from the IOBC over an Inter-Integrated Circuit (I2C) communication bus.

The header board is operated using software that is divided into two parts: the base software and the application software. The base software handles the basic part of the header board operation such as command communication with the IOBC. It is hard-coded in the onboard flash memory, and cannot be modified. The application software manages all the temperature control tasks and passes necessary information back and forth between the header board and the IOBC during the operation. In contrast to the base software, the application software is modifiable and stored in a flight-reprogrammable section of the onboard flash memory.

![Header board operation user interface with INIT_POINTER_FILE selected.](image)

Initially, during the BRITE PGC development, the handling of all header board commands was accomplished through the manual command interface. This was done because the data in the response packets of all header board commands did not require any manipulation. However, later on, it was
determined that using the manual command interface to monitor and manage the approximately 50 different operation parameters was extremely inefficient and inconvenient. For this reason, requirement BP 6.2.6 has been defined to ensure that there is easy and quick access to the operation parameters (referred to as control files). To meet this requirement a dedicated control interface for the CCD header board was administered in the CHHeaderboardControlDlg class as shown in Figure 2.9. Furthermore, the ability to update the CCD Header board application code has been placed in the CHHeaderboardControlDlg class in order to satisfy requirement BP 6.2.7 since updating the software is one of the header board-related operations.

Similar to the other control interfaces, the header board control interface contains a list control object showing all accessible control files for the CCD header board. To get the most updated value or to set a new value, the Get Selected or the Set Selected button can be clicked, respectively, after selecting the desired control file for the operation from the list. For convenience, the Get All button has been placed in the interface to provide an easy, one-click solution for retrieving all control file values. The CCD header board application code update operation is initiated by clicking the Load button in the Application Code section at which point the user is prompted to choose the source file. Once the appropriate source file is selected, a BRITE header board application code uploader dialog appears as shown in Figure 2.10.

![Figure 2.10: BRITE Header board application code uploader dialog with a sample code loaded.](image-url)
The header board application source code conforms to the Intel Hexadecimal Object File format [20]. Within the source code, the hexadecimal representation of binary for the application software is stored in American Standard Code for Information Interchange (ASCII) alphanumeric characters along with a 16-bit wide field that specifies the starting load offset of the data bytes. Using this information, the uploader dialog loads the source code into memory and populates it into the code display control object upon initialization. By clicking the Upload button, the loaded source code is packed into a series of header board poke commands and these are then issued to the header board to update data at the appropriate address of the onboard flash memory. In order to provide an interface to confirm the result of the uploading process, a function that compares the application code written in the onboard memory against the loaded source code has also been implemented. During the verification operation, which is executed by clicking the Verify button, the written application code from the proper address is read back using a series of header board peek commands and compared byte by byte.

In addition, an input field defined as Number of bytes per packet has been added into the design to provide an option for controlling the packet size of the poke commands during the uploading operation. Such controllability is advantageous if the application code update operation is conducted for a spacecraft on orbit. This is because when the communication is made through radios, and the transmission failure rate is proportional to the size of packets for uplink. Being able to adjust the length of the packet provides the ability to compensate for unfavorable link conditions when performing an uploading operation.
Chapter 3

BRITE Schedule

BRITE Schedule is a mission-specific planning software tool that was developed for scheduling target observations for BRITE constellation spacecraft during the nominal phase of the mission. In summary, BRITE Schedule takes in information about target observations from an external source in order to initiate the scheduling process. As a result of the scheduling process, BRITE Schedule generates a series of spacecraft commands in various formats that are later uploaded and executed by the spacecraft in an autonomous manner to perform the observation. The current plan entails using a single instance of BRITE Schedule for scheduling target observations for each BRITE spacecraft.

3.1 Software Requirements

A subset of software requirements for BRITE Schedule is listed in Table 3.1 below. The complete set of requirements can be found in [21].

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS 6.1.2</td>
<td>BRITE Schedule shall be capable of interpreting data from BRITE Target representing the observation target’s time and position in inertial space to create a full set of command packets to perform the observation.</td>
</tr>
<tr>
<td>BS 6.1.4</td>
<td>BRITE Schedule shall be capable of interpreting and providing human-readable satellite command outputs from the observation data.</td>
</tr>
<tr>
<td>BS 6.2.1</td>
<td>BRITE Schedule shall be capable of selecting and interpreting BRITE Target setup files representing the observation target’s time and place to create a full set of</td>
</tr>
</tbody>
</table>
command packets to perform the observation.

| BS 6.2.2 | BRITE Schedule shall be capable of converting BRITE Target setup files into script files that can be read by TimeTag Uploader in order to create the TimeTag packets which are sent to the satellite. |
| BS 6.2.4 | BRITE Schedule shall be capable of generating script files that can be read by TimeTag Uploader, independent of a BRITE Target setup file. In addition, BRITE Schedule shall be capable of editing existing script files, including those generated from BRITE Target setup files. |
| BS 6.2.7 | BRITE Schedule shall not have any direct uplink or downlink with the satellite. |

Table 3.1: Primary BRITE Schedule Software Requirements.

### 3.2 Observation Planning, BRITE Target, and Setup File

The very first step of preparing a successful target observation is to define a set of tasks for the observation. This planning operation is performed at a ground station. During the planning operation, a target star field is identified for an observation and setup details are defined. These details include the exposure time of the imager, the star location in the captured image, and the image processing method. To allow scientists and operators to perform all necessary planning operations in a comprehensive manner, the BRITE Science team in Austria has developed a graphic-user-interface based ground segment program called BRITE Target. Most importantly, BRITE Target generates two observation files containing information about the planned observation: one in human readable Extensible Markup Language (XML) format and one in binary format. In particular, the information that creates a full set of spacecraft commands for the observation is stored in the observation setup file in XML format. Thus, as stated in the requirements BS 6.1.2 and BS 6.2.1, BRITE Schedule accesses the XML file extensively during the later stage of target observation preparation (observation scheduling).

### 3.3 TimeTag Scripts and TimeTag Uploader

Overall, there are two different types of timetag script necessary for the target observation operation. The first type generated by BRITE Schedule is a binary script file that contains a series of time
tagged spacecraft commands in relative times. This type of script is defined as a spacecraft timetag script because it is uploaded to a spacecraft file system and the spacecraft itself handles the interpretation and dispatching of tasks in order to execute the commands in the script. Detailed information on the spacecraft timetag script can be found in [22]. The second type of timetag script is the timetag uploader script. In contrast to the first type, the contents of the timetag uploader script file are human-readable and its commands are expressed in an absolute time frame. Furthermore, the script files cannot be interpreted by the spacecraft itself. In order to schedule the list of spacecraft commands in the script, they must be imported and uploaded to a spacecraft using the ground segment program TimeTag Uploader. More information on the implementation of TimeTag Uploader and the timetag uploader script can be found in [23].

After the observation scheduling is accomplished using BRITE Schedule, a set of timetag scripts of both types are generated depending on the schedules and options that were specified during the scheduling process. Detailed information on the types of script file and their contents will be discussed in Section 3.4.2.

3.4 BRITE Schedule Design Considerations and Decisions

3.4.1 Spacecraft-Specific Operation Sequence for Target Observation

In order for the BRITE spacecraft to perform a successful target observation, a series of actions must be followed. Currently, the sequence assumes that the BRITE instrument will observe one target field per orbit with less than 100% duty cycle since the target field is not visible over the entire orbit. Under this assumption, it has been found that the spacecraft needs to go under different stages of operations (initialization, imaging, and deinitialization) during the course of an orbit and also needs to control all three on-board computers (OBCs) to complete the observation. Table 3.2 has been prepared to show the actions that must be performed by the OBCs during the different operation stages. This is shown diagrammatically in Figure 3.1 to provide a better understanding.
Figure 3.1: Single Target Observation Sequence [24].

| Initialization Operation | 1. Turn on the star tracker  
|                          | 2. Set desired pointing target  
|                          | 3. Enter Fine Three-Axis Pointing (FTAP) mode  
| Attitude Determination and Control Computer (ADCC) |  
| House Keeping Computer (HKC) | 1. Turn on instrument on-board computer  
| Instrument On-board Computer (IOBC) | 1. Load and execute application code  

| Imaging Operation | 1. Set the desired exposure time of the imager for observation  
|                  | 2. Turn on the power switches for the imager  
|                  | 3. Start the observation  
|                  | 4. Stop the observation  
|                  | 5. Turn off the power switches for the imager  
| Instrument On-board Computer |  

| Deinitialization | 1. Exit application code  
| Instrument On-board Computer |  
| House Keeping Computer | 1. Turn off the IOBC  
| Attitude Determination and Control Computer | 1. Enter Coarse Three-Axis Pointing (CTAP) mode  

Table 3.2: Required Actions of On-board Computers for a Target Observation.
3.4.2 Limitations on TimeTag Operation and Scripts

When the scheduling process is initiated within BRITE Schedule, the list of actions in Table 3.2 is translated into a set of corresponding time-tagged spacecraft commands for a single target observation operation. For BRITE missions, the maximum number of timetag commands that can be stored in the spacecraft buffer is set to 512. Currently, there are about 25 commands that need to be executed in order to perform a single target observation operation. When absolute-time tags are used for the commands, the number of observations that can be scheduled at a time is limited to 20, which translates into less than two days of observations (assuming 16 orbits per day). This is undesirable since about 500 timetag commands (20 observations each with 25 commands) must be uploaded to the spacecraft daily in order to achieve non-stop target observation. If part or the entire communication link is being used for other operations, the commands may not be able to be uploaded and the desired observations will not be achieved.

To deal with this problem, a design decision was made to use relative time-tagged commands since it provides an efficient way to manage the spacecraft’s timetag command buffer. Unlike absolute time-tagged commands, the execution time of the command is not defined during creation. Only a time offset between a base time (which is provided later as part of the operation) is defined for each command. The exact execution time is then later determined by the spacecraft when the command is dispatched into the timetag command buffer. Furthermore, a sequence of relative time-tagged commands that share the same base time can be grouped together and stored in a single file defined as a timetag command script file. All commands stored in a script file are loaded into the timetag command buffer at the same time and the offsets are added with the provided base time when the script file is loaded into the buffer. In addition, all script files are stored in the 256MB FLASH memory of the spacecraft and can be called multiple times as needed.

By applying relative time-tagged commands, scheduling more than 2 weeks worth of target observations at a time was made possible (assuming 16 orbits per day and about 25 commands for a single observation). There are two distinct timetag command scripts generated by BRITE Schedule that are used to perform target observations. The first script contains commands associated with actions that need to be taken around the time when a target field becomes visible to the spacecraft (referred to as ‘target rise time’). The second script is for the associated actions around the time when the target field goes out of the spacecraft’s view (referred to as ‘target set time’). By default, each
script contains a subset of the commands listed in Table 3.2. All commands for the initialization and
part of the imaging operation (up to Step 3) are placed in the script file that references the target rise
time. The other commands are placed in the second script. Furthermore, BRITE Schedule generates a
timetag uploader script that contains a series of commands that load those two timetag command
scripts for the observations in the appropriate manner. Currently, the timetag uploader commands
are generated in such way that the commands in the script files are dispatched into the timetag
command buffer five minutes before the expected execution in order to minimize the number of
commands in the buffer at a given time.

3.4.3 Generic TimeTag Command Operation Scheduler

While BRITE Schedule was being developed as a specific target observation scheduler, many distinct
components were created to fulfill the software requirements. For example, a set of libraries for
managing the commands for all OBCs in the BRITE spacecraft were implemented along with an
architectural component that provided an efficient means of organizing lists of commands created
from these libraries as a container for scheduling. Furthermore, additional features were developed for
the libraries and the container which exported the list of commands to the timetag command scripts
and the timetag uploader scripts. With the wide range of functionality implemented in BRITE
Schedule, it was identified that the timetag schedules for any generic mission could also be generated
using BRITE Schedule if minor tweaks were made to the user interface design. Since the
modifications were minor and relatively easy to carry out while yielding significant generality to the
software, it was decided to go ahead with the modifications to the BRITE Schedule design. As a
result, the functionality of BRITE Schedule has been extended and has become a generic timetag
command operation scheduler.

3.5 Software Architecture Design & Implementation

The architectural implementation of BRITE Schedule has been defined as shown in Figure 3.2. The
application is comprised of four different functional blocks: schedule manager, command creator,
BRITE command module, and target observation scheduler. Each block is made up of one class or
more in order to serve specific roles within the application. A dedicated user interface design is linked
to each functional block (except for the BRITE command library) in order to provide a means of interaction and to obtain necessary information from the operators.

Figure 3.2: Top-level architectural implementation for BRITE Schedule.

The functional blocks were further grouped into two different scheduling groups: generic operation scheduling and mission-specific operation scheduling. The main reason for this separation in the design was to incorporate the design decision of extending the generality of BRITE Schedule (as discussed in Section 3.4.3) and to handle the two groups separately using dedicated algorithms. Consequently, the separation made the overall implementation of the BRITE Schedule design modular. If any additional specific schedulers are identified or required during the nominal mission phase, they can be seamlessly integrated into BRITE Schedule without affecting the existing scheduler.

3.5.1 Schedule Manager

A schedule manager is a set of class implementations that provides a means of managing a list of spacecraft commands under the scheduling process. At the core of the schedule managing operation is the CSchedule class which has been developed in order to serve as a smart container object that holds a series of spacecraft commands in C++ object form (Section 3.5.3). The reason why the schedule container is described as smart is because the object itself can perform almost all of the file input and output (I/O) operations that are required during the operation of BRITE Schedule. There are three different file formats that the CSchedule object can handle: a timetag command script (Section 3.3), a
timetag uploader script (Section 3.3), and a BRITE schedule file. For the output operation, the list of spacecraft commands can be exported to any of the allowed formats without supplemental information. The output file format is controlled by supplying the appropriate format type parameter into function. The files generated by the CSchedule object can be easily loaded back to the object again. In addition, all the file I/O operations are supported by a CBRITEScheduleDlg class, which is a graphical user interface (GUI) part of the schedule manager design. Detailed information on the schedule manager’s GUI implementation can be found in Section 3.6.1.

### 3.5.2 Command Creator

As the name indicates, the command creator is the functional block that is in charge of creating various BRITE spacecraft command objects which are used within the application and passing the created object to the schedule manager. At the front end, a simple user interface design was developed which allows operators to input the necessary information. As the information is being entered or when the command creation process is initiated, the command creator converses with a command library (which is a module connected at the back end) in order to process the requested actions. Detailed information on the command creator’s user interface can be found in Section 3.6.2.

### 3.5.3 BRITE Command Module

During the BRITE Schedule development, a significant amount of time was allocated to construct a generic command managing architecture which would provide an efficient way to manage spacecraft commands used within the application. Throughout the development, user experience and feedback obtained from the BRITE Payload Ground Control’s command handler implementation were taken into consideration in order to increase the efficiency and robustness of the library design. The block diagram of the current BRITE command module implementation is shown in Figure 3.3.

---

1 A BRITE schedule file is a specially designed format which stores contents of the CSchedule class object for BRITE Schedule. The contents are stored in human-readable format and editable using common text editor software. An example of the file is prepared in Figure 3.4.
3.5.3.1 Inheritance and Command Library

The development of the command library takes advantage of the C++ inheritance where Classes are defined using a parent-child relationship. When a class is inherited from another class, the inherited class is referred to as the ‘child’ or ‘derived’ class and may become the ‘parent’ or ‘base’ class for yet another class. In addition to its own components, a child class inherits all the functions and data members from its parent. The child class may access all inherited components unless otherwise defined in the parent class definition. Furthermore, any child object may be accessed using a pointer of the same type as its parent class. This is very advantageous as it provides a flexible way to manage the objects of the child classes that have the same parent.

For the current command library design, the singleton inheritance hierarchy architecture has been employed. The CNSPPacket class is defined at the top of the hierarchy and serves as the parent for the BRITE commands as shown at the right side of Figure 3.3. Most importantly, the CNSPPacket provides the blueprints for its children in such a way that enables ‘smart-operations’. A smart-operation refers to the ability of a child class to indicate and validate the number and type of its required input parameters. The child class can also generate its own command packets in the Nano-Satellite Protocol (NSP) [12] format using its received inputs. For each BRITE command, a dedicated
child class has been created; the full set of these form the BRITE command library which is used by BRITE Schedule.

3.5.3.2 Command Managers

After the elements of the command library were developed, the command manager class was added to complete the BRITE command module design. Within the BRITE command module, the command manager classes act as a wrapper providing an interoperable software interface for clients to access the command library functions. As illustrated in Figure 3.3, there are multiple command manager classes placed in the BRITE command module: CHKCBootloaderCmdManager, CHKCAplicationCmdManager, CADCCBootloaderCmdManager, CADCCApplicationCmdManager, CIOBCBootloaderCmdManager, and CIOBCApplicationCmdManager. They were defined to reflect the two software modes for each of the three different on-board computers (OBCs). Each command manager class represents a specific software mode for a specific OBC and is linked to a subset of commands from the library.

Most importantly, a command manager provides a flexible software interface to dynamically create objects of spacecraft commands. There are three different ways of creating command objects for clients using a command manager: (1) by an index number of the command in the available command list; (2) by a unique command number; (3) by the name of the command defined in the available command list. When a command object is created using one of the methods specified above, the command manager returns the CNSPCommand type object to the client for further operation. The first method brings a convenient way of accessing the linked commands when a client lists the available commands in the command manager and creates a command object accordingly. In the BRITE Schedule design, this method was heavily used by the command creator where it allowed operators to browse through all available BRITE commands and add the commands to a schedule list in the schedule manager. The second method, which allowed clients to create commands using unique command numbers, was implemented in order to permit BRITE Schedule to convert commands from the NSP format into the command object form that the schedule manager can work with. This feature is convenient when already-generated timetag scripts are imported back to BRITE Schedule. The usage of the third method is similar to the second one, except that this method is called when BRITE Schedule loads a set of commands from a BRITE Schedule file. Within a BRITE Schedule file, commands are expressed in human-readable format as shown in Figure 3.4. The command definition
part of each entry – the actual name of the command – is used to specify the type of command along with supplemental information if applicable. By including the ability to create command objects using a command name in the command manager design, parsing and loading processes of the command components in BRITE Schedule files were easily achieved.

Figure 3.4: An example of a BRITE Schedule file.
3.6 Software Interface Design & Implementation

The BRITE Schedule graphical user interface (GUI) was designed in such a way that operators can easily begin to use the software with minimal training. The names of menus and their sub-components follow the most commonly used naming conventions. The interface also makes use of accelerator-key definitions to improve user-efficiency. Furthermore, the number of control components that the user can access within a single dialog is minimized in order to preserve the simplicity of the GUI. Similar to other ground segment software tools developed at the Space Flight Laboratory (SFL), BRITE Schedule conforms to a single document interface (SDI) form – each instance of the software handles only one schedule of BRITE timetag commands at a time. Hence, the root dialog of BRITE Schedule has been designed in such a way as to assist operators in managing timetag commands during the scheduling process. Figure 3.5 shows the root dialog design of BRITE Schedule, which is brought up after execution.

![Figure 3.5: BRITE Schedule’s root dialog design.](image)

---

2 Accelerator-keys, also known as hot-keys, are used as a shortcut to execute a function that would normally require multiple user actions within (e.g., selecting items, moving the mouse, and selecting from a menu). Accelerator-keys are usually defined as a combination of a letter key and a modifier key (Ctrl and Alt keys).
3.6.1 Schedule Manager

List control was implemented in the schedule manager user interface design in order to enable the displaying and managing of timetag commands in the CSchedule object (Section 3.5.1). Furthermore, several clipboard operation functions were implemented within BRITE Schedule since it will be mostly accessed during the nominal phase of the mission. An efficient way of manipulating the list of timetag commands for the scheduling is highly desired. Currently, common clipboard operations such as cut, copy, paste and drag-and-drop are permitted and they are accessible through accelerator-keys, a menu, and a context menu\(^3\) as shown below in Figure 3.6 and Figure 3.7.

\(^3\) A context menu is a pop-up menu that appears when a right-click mouse operation is asserted. It contains a set of actions that can be performed on the selected object.
Moreover, the schedule manager user interface serves as the primary interface for executing other available functions within BRITE Schedule. The functions are called by selecting them from the appropriate drop-down menu. For example, the file input and output (I/O) operations that are implemented in the CSchedule object are accessible through the File and Build menus as shown in Figure 3.8. The Tools menu, shown in Figure 3.9, allows the user to access the command creator, the target observation scheduler, and the settings manager.

![Figure 3.8: BRITE Schedule menu for file input and output operations.](image)

![Figure 3.9: BRITE Schedule menu for functional modules.](image)

### 3.6.2 Command Creator

The command creator graphical user interface allows operators to create BRITE commands by interacting with the BRITE command module controller (as defined in Section 3.5.3). The interface connects to the available command managers and provides an organized way of accessing all the associated commands for the chosen manager, as shown in Figure 3.10. Furthermore, when a command in the list is selected, all required parameters for the selected command are listed and held for operators to enter the appropriate values.
When operators attempt to enter a parameter value, the interface accesses the appropriate command in the library and prepares a list of options that the operator can choose from. Figure 3.11 shows an example of the command creator displaying the available option list for a pin parameter of an IOBC_GPIO command. Moreover, every parameter input undergoes a validation process during which it is compared against a rule defined within the specified command implementation. If the validation fails, the failure is indicated by highlighting the appropriate cell in red as shown in Figure 3.12. This validation feature was implemented in order to minimize possible operational failures caused by an operator’s mistake as well as to catch errors as early as possible during the operation process. It is always desirable for errors to be identified and corrected during the planning stage.
3.6.3 Target Observation Scheduler

The Target Observation Scheduler is a group of GUI implementations that allows operators to generate the timetag scripts for scheduling target observations (Section 3.4.2). As previously mentioned, the functional implementation for mission-specific operation scheduling was completely separated into a modular block called target observation scheduler. In order to provide an intuitive way of generating the timetag scripts for the target observations, the scheduling operation was separated into multiple stages, each with their own GUI. The stages were designed in such a way as to keep the total number of actions in each stage to a minimum while grouping similar actions together. In addition, an effort was made to ensure that the flow between stages was as intuitive and self-explanatory as possible. In total, four different stages were identified for the complete scheduling cycle: Target Field Selection, Observation Time Selection, Confirmation and Generation, and Observation Settings (see Figure 3.13).

When the Target Observation Scheduler is executed, the Target Field Selection GUI, which is the entry point for the target observation scheduling process, appears to the operator (Figure 3.14). The Target Field Selection interface allows operators to browse through directories on the system and select a target observation setup file which is required to complete the scheduling process.
Furthermore, during the setup file selection, important parameters are parsed on-the-fly and displayed within the interface to provide operators with a quick summary of the observation.

Figure 3.14: Target Field Selection user interface during the target observation scheduling operation.

Figure 3.15: Observation Time Selection user interface during target observation scheduling operation.
The Observation Time Selection interface (shown in Figure 3.15) is used for the second step of the target observation scheduling process. Within this GUI, the specific times for a target observation are selected. All of the available target visible times are displayed in a list within the interface. Since there may be a very large number of the visible times (a few thousand), a feature that allows the user to filter items based on different criteria was developed. Currently, the items can be filtered by a range of dates and by the minimum duration of time that a target stays either visible or invisible to the spacecraft. Furthermore, an option which allows the operator to limit the number of filtered items has also been implemented. This provides a quick and easy way to incorporate the maximum number of schedulable observations determined by the limitation on the spacecraft’s timetag buffer size (see Section 3.4.2 for details on timetag scripts).

Figure 3.16: Confirmation and Generation interface during target observation scheduling operation.

Once the desired target visible times are specified, a set of timetag scripts for the observation may be generated. This is done in the third GUI of the Target Observation Scheduler. (See Section 3.4.2 for different types of timetag scripts). By default, the file names for all three timetag scripts are automatically generated following a predefined naming convention as illustrated in Figure 3.16; however, the operator may also choose to specify the file names manually. The default file names for the two timetag command scripts include the generation timestamp (month and day in numeric format) along with a prefix of a character ‘S’ and a digit which indicates the script number. In a similar manner, a fixed string ‘Schedule’ is used followed by the complete script generation timestamp (year, month, day, hour, minute and second) for the timetag uploader script. Notice that a less
accurate timestamp string is used for the timetag command scripts in comparison to the ones for timetag command scripts under ‘Schedule’. This is because there is a limitation on the length of the file name\(^4\). Once the *Finish* button in the third GUI (Figure 3.16) is clicked, the Target Observation Scheduler will generate the scripts and place them into the location that is specified in the output directory field.

The Observation Settings GUI, which is accessed by selecting the *Advanced* button from the Confirmation and Generation stage, has been implemented to allow the fine-tuning of timing in relation to the operations performed during target observations. This interface allows the operator to specify many different settings that are accessed by the target observation scheduler during the script generation process. The Observation Settings stage is accessed from the last step of the scheduling process since the majority of the settings do not need to be changed frequently. Also, the Observation Settings interface is only displayed if the user intentionally selects to open it in order to avoid any confusion that may arise by forcing the operator to go through the Settings interface. Figure 3.17 shows the GUI of the Observation Settings stage.

\[\text{Figure 3.17: Observation Settings user interface during target observation scheduling operation.}\]

\(^4\) The current implementation of the on-board computer file system requires file names to be less than 12 characters – this restriction does not impact the ground segment software. Detailed information on the OBC’s file system implementation can be found in [37].
Chapter 4

Star Tracker All-in-one Control Interface

In BRITE spacecraft, a star tracker has been integrated as part of the Attitude Determination and Control System (ADCS) to obtain accurate information on the spacecraft’s attitude needed to achieve the required pointing performance during target observation operation. Currently, there are two different star trackers used in the BRITE constellation satellites. In the initial phase of the BRITE program, there were only two BRITE satellites (UniBRITE and BRITE-Austria) under development, and a Miniature Star Tracker (MST) from Comtech AeroAstro Inc. was used. Using the MST for subsequent satellites was not financially feasible. Furthermore, because the device was subject to International Traffic in Arms Regulations (ITAR) – limiting the flexibility of the mission and launch campaign – a design decision was made to stop using the MST. Instead, the newly designed S3S star tracker that was built by Sinclair Interplanetary in conjunction with UTIAS/SFL and Ryerson University was chosen as a solution. Because the S3S star tracker was newly developed, there were no supporting tools that could be used during assembly. As a result, Star Tracker All-in-one Control Interface (STACI) was developed at UTIAS/SFL to provide the necessary control interfaces during testing, debugging, and both pre-flight and on-orbit operations for the S3S star tracker.


4.1 Software Requirements

A subset of software requirements for the STACI is listed in Table 4.1 below. The complete list of requirements can be found in [25].

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACI 1.1</td>
<td>The STACI shall provide operators the means to issue all commands that a S3S star tracker is capable of executing.</td>
</tr>
<tr>
<td>STACI 1.4</td>
<td>The STACI shall be capable of performing the built-in-self test for a S3S star tracker and generating the corresponding test report.</td>
</tr>
<tr>
<td>STACI 1.5</td>
<td>The STACI shall be capable of downloading data from the star tracker, including images taken for attitude determination operations.</td>
</tr>
<tr>
<td>STACI 1.6</td>
<td>The STACI shall be capable of updating the star tracker’s flight application code along with star catalogs.</td>
</tr>
<tr>
<td>STACI 1.7</td>
<td>The STACI shall be capable of manipulating the control parameters for attitude determination operations.</td>
</tr>
</tbody>
</table>

Table 4.1: STACI Software Requirements.
4.2 Design Considerations and Decisions

4.2.1 Non-Standard Nano-Satellite Protocol Message Length

Like all electronics developed at UTIAS/SFL, the S3S star trackers uses Nano-Satellite Protocol (NSP) messages for communication with external devices. However, during the development, it was found that the version of NSP used on the S3S star tracker was slightly different from the standard one used at UTIAS/SFL. The maximum length of the NSP response message without the frame encapsulation was twice as long as what the standard NSP allowed. Taking this abnormality into account, the size of the memory buffers used to handle NSP messages within the application were doubled along with the appropriate modification to the functions that use those buffers for operation. The details on the standard NSP can be found in [12].

4.2.2 Operational Task Groups

Upon examination of the interface control document (ICD) for the S3S star tracker [26], it was found that managing the star tracker for the operations outlined in the requirements (described in Table 4.1) could not be accomplished in a straightforward manner. Most of the operations required a number of preparation steps in order to set the device into a state where the required operations could be conducted. Moreover, there was a lot of repetition involved with the preparation steps. As a result, the operations required by STACI were grouped accordingly into the six different operational task groups shown below, and the implementation for the each group was made separately in the STACI design:

- Manual Command
- Self-Test
- Attitude Determination
- Control-Structure
- Download
- Firmware Update
4.3 Software Architecture Design and Implementation

The overall software architecture for STACI has been designed in a similar manner to that of BRITE Payload Ground Control (PGC). This is because the purpose of both applications is very similar and both applications have to be developed to provide a control interface for a device during testing, debugging, as well as pre-flight and on-orbit operations. The only difference between these two applications is the target device to control.

STACI is composed of two major blocks, as shown in Figure 4.2: an operational block for the task group handlers and a functional block with three separate modular components that serve specific tasks (i.e., S3S Message Handler, Communication Handler, and Logging Handler). Within the application, the modular components in the functional block act as a slave for the operational block and they interact and exchange data only with the operation block during operations.

![Figure 4.2: Top-level software architecture for STACI](image)

### 4.3.1 S3S Message Handler

The S3S message handler module is a combination of classes in charge of creating, managing and interpreting all NSP messages (i.e., command and response packets) used in STACI for the communication. The initial blueprint of the message handler design was inherited from the command module developed for BRITE Schedule (see Section 3.5.3). Since the BRITE command module does not contain implementations for interpreting response packets, the concept of interpreting response
packets introduced in the BRITE PGC design (see Section 2.2.1) has been infused into the inherited design to complete the structure of the current message handler design for STACI. Figure 4.3 has been prepared below to illustrate the simplified class relationship diagram of the S3S message handler module.

![Figure 4.3: Simplified class relationship diagram for S3S Message handler module.](image)

As shown in Figure 4.3, the notable change in the command module design from the one for BRITE Schedule is the response interpretation section located on the right of the diagram. For the core of the response interpretation algorithm, a CNSPResponse class was defined. As all response packets are in NSP format, the CNSPResponse class was inherited from the CNSPPacket class and two CStringArray members, “response_data_name_” and “response_data_”, were defined in the class to store the names and values of all the interpreted (parsed) data from the response. When STACI receives a response packet from the S3S star tracker, an instance of CNSPResponse object is created and initialized using the received raw packet. It is then parsed using the corresponding object used for the command, passed to the appropriate task handlers in the operational block, and handled...
appropriately. A flow chart of the described response packet interpretation process is given in Figure 4.4.

![Flow Chart](image)

Figure 4.4: A flow chart of processing response packets from S3S star tracker.

### 4.3.2 Communication Handler Module

The communication handler module is a set of classes that allows STACI to communicate with the S3S star tracker through transmission control protocol/internet protocol (TCP/IP). The framework developed earlier for ground segment software GNBControl was adopted into the STACI design with minor tweaks. This was because the author was already familiar with its functionalities and it was mature enough since it had gone through iterations of testing and debugging processes. This also allowed the overall software development time to be shortened. More information on the GNBControl and its communication handler implementation can be found in [27].

### 4.3.3 Logging Handler Module

The logging handler module is a single class implementation that is in charge of logging text-based messages into a file. The handler itself takes care of all necessary tasks for the file management (i.e., creating and writing). A new log file is created every time the STACI is executed, and the handler keeps only one log file at a time. However, it can be changed to a new one later on within the application by request. For the messages for logging, any members within the application can issue them. When the logging handler receives the messages, it writes them into the file in the order they are received. In addition, the messages are appended on a per line basis.
4.3.4 Operational Block

The class diagram for the operational block is shown in Figure 4.5. It is comprised of six distinct classes, all of which handle different types of task groups defined earlier in Section 4.2.2. Among those task groups, the manual command task group has been chosen to be the master over the rest of the task groups and its implementation was placed in the main application class. Such a design decision was made because during the development, it was found that the manual commanding task was a necessary operation and needed to be used commonly to accomplish the other required tasks.

![Class diagram for the operational block and the corresponding task handlers.](image)

Furthermore, within the operational block, the manual command task handler acts as a data bridge between the other task handlers and the modules in the functional block introduced in the previous sections. Only the manual command task handler has the ability to interact with all the modules, and all data from and to the task handlers must pass through the manual command task handler. This type of architecture is advantageous because it provides a means of keeping track of data traffic within the application. No dedicated functional module needs to be developed, and consequently it makes the overall design simpler.

Lastly, the task handlers in the operational block have been implemented in such a way that each contains all necessary functions with its own graphical user interface (GUI) design for handling the assigned task. Hence, all class implementations in the block are independent from one another and are not affected by the implementation of others. Moreover, the addition of more task groups into STACI
that may be required in the future can be easily accomplished while the implementation of other modular blocks remain untouched. The detailed GUI designs and implementations for the task groups are described in the following sections.

### 4.4 Software Implementation and Interface Design

The base user interface structure for STACI has been designed in a similar manner to the one developed for BRITE Payload Ground Control (PGC). A tab interface control was used to link all user interfaces designed for the task groups to the main application dialog. Furthermore, all other interfaces that are used commonly during the operations have been placed within the main application dialog. Figure 4.6 shows the layout of the main application dialog.

![Figure 4.6: Screen capture of the main application dialog for STACI.](image)

The main application class, CSTACIDlg, which contains the user interface implementation of the main dialog, is composed of four sub-sections of control interfaces: connection control (top-left), log display (bottom), manual command (mid-left), and task-specific control (top-right). All instances of controls placed in this dialog are declared as a member in the CSTACIDlg class except the ones that are linked in the tab control. For those interfaces linked to the tab control, their instances are created as an object during the initialization of the CSTACIDlg class, and their pointers are stored in the CSTACIDlg for later access by the tab control. Similarly, instances of the modules in the functional block (Figure 4.2) are also accessed using pointers within the main application class as needed.
4.4.1 Connection Control Interface

For the implementation of the connection control interface, the one developed for the BRITE PGC design has been reused with minor changes in the appearance of the buttons. Instead of using plain text to describe the function of the buttons (i.e., Connect and Disconnect), self-explanatory icons were used in STACI with corresponding tool tips for each of the buttons, as shown in Figure 4.7 (A button denoted as Options… is also in the figure to show another example of using the icon button. The option button is used to open the option interface which is explained in the following section). Such a design change was made to reserve and to have efficient use of the main dialog space for other controls.

![Connection Control Interface](image)

Figure 4.7: Connection control user interface for BRITE PGC (left) and STACI (right).

4.4.2 Options Interface

To allow operators to manage configurations that define the software behavior of STACI, a dedicated interface design, Options, has been created, as shown in Figure 4.8. The configurations specified within the options interface are stored in the Windows registry. The advantage of using the Windows registry as data storage is that there are no files involved in storing the configurations. Consequently, it makes accessing those configurations fast and easy because no implementation for file input and output (I/O) is required – storing and retrieving the configurations can be done simply by calling a single function. Furthermore, the functions used for accessing the Windows registry are part of Windows native Application Programming Interface (API). Therefore, accessing the configurations can be done within any class (e.g., task handler classes) as needed and, consequently, the software architecture of STACI can be constructed simpler than it would be using separate files for storing the configurations.
CHAPTER 4. STAR TRACKER ALL-IN-ONE CONTROL INTERFACE

However, there is a downside to using the Windows registry as well. Since the configurations are stored in a pre-defined absolute location of the registry, only a single set of configurations is definable and it is shared among all instances of STACI in the same environment (e.g., computer). This is disadvantageous if there is more than a single S3S star tracker to be controlled with different configurations using a single ground station computer. Regardless, this is not an issue for the BRITE mission because there is only one star tracker to control in each BRITE spacecraft. However, if multiple star trackers need to be controlled in the future, an implementation will be made to allow for multiple sets of configurations.

4.4.3 Manual Commanding Operation

As previously discussed, the functional implementation of the manual command task group handler has been made in the CSTACIDlg class along with its interface design. The manual command task group is a set of tasks defined to perform manual commanding operations to satisfy the software requirement STACI 1.1. The task group handles the commanding operation on a single command basis. Such a decision has been made to make the manual command task group generic, allowing it to be used under any circumstances to support other task groups that require additional control commands for their successful operation.

Since the functionality of the manual command handler does not require extensive manipulation with both the command and response packets, a simple interface allowing operators to create various S3S star tracker NSP commands with the destination selection option was developed, as shown in Figure 4.9. The core interface design and algorithm of creating and validating NSP commands (i.e.,

![Options user interface](image)
CNSPCommand objects) developed for the BRITE Schedule command creator was reused with the
NSP command library plugin prepared for S3S star trackers. This was because at the time STACI's
manual commanding interface was being developed, the algorithm was well-tested and mature enough
to be used for other applications. This also allowed for the overall software development time to be
reduced.

Figure 4.9: Manual commanding operation user interface (left) with its command selection list
unfolded (right).

4.4.4 Self-Test Operation

To handle the necessary tasks to perform the star tracker's built-in self-test functionality as per
requirement STACI 1.4 (Table 4.1), CSelfTestTab class was created with its user interface, as shown
in Figure 4.10. The built-in self-test cycle is commenced by sending a COMBINATION command to
the star tracker with an argument byte, defined as “Go Code”. Its six least significant bits specify a
behavior of the star tracker during the self-test process. To provide a convenient way of determining
the Go Code, a set of radio buttons for selecting each bit with the appropriate descriptions are placed
into the user interface.
Once the test is initiated, the device runs through pre-defined self-test sequences, captures snapshots of various telemetry, and returns the captured measurements via the response packets at the end of the process. As soon as the self-test task handler receives the results, the results (which are in binary format) are parsed into human-readable format and are processed according to the options specified within the self-test user interface. Furthermore, polling functionality is included in the self-test task handler. This is because the self-test operation needs to be continuously performed during thermal functional acceptance testing, as part of SFL’s quality assurance procedures, and it is desired that this be done in an autonomous manner. The detailed self-test operation flow diagram is prepared in Figure 4.11.

Figure 4.10: Built-in self-test operation user interface.

Figure 4.11: Flow diagram for the self-test operation.
4.4.5 Attitude Determination Operation

An attitude determination operation task group handler has been defined to encapsulate tasks required to obtain attitude measurements from the S3S star tracker. Its implementation has been made in a COperationTab class. In a similar manner to the self-test operation, the operation is triggered by sending the COMBINATION command to the star tracker. However, in this case, the self-test bit in the Go Code, which differentiates the operation from the self-test, is masked out. Furthermore, an additional bitmask byte that defines the type of desired measurements to retrieve is appended to the command packet. To provide an easy way to specify the type of desired measurements in the bitmask, check-box controls for each type of measurement with the appropriate description are placed in the user interface, as shown in Figure 4.12.

![Figure 4.12: Attitude determination operation user interface.](image)

Once response packets containing the desired attitude measurements are received from the star tracker, the measurements are parsed and subjected to further processing. The parsed measurements are processed in three different steps. First, all the parsed measurements are printed in the text box at the bottom of the user interface. Next, a file containing the same contents printed in the text box is created – if requested by the operator – by setting the Create Result file option. Lastly, if the parsed measurements include quaternions or angular velocities (ACS measurements), they are, also by request, saved into a separate file defined as the ACS result file. In addition, if polling, the ACS measurements are appended at the end of the previously created ACS result file until the new file is created. The attitude determination interface is to perform unit-level acceptance tests on S3S star...
trackers, and it is desired to have a single file containing all ACS measurements during the test. Figure 4.13 shows an example of collected ACS measurements in a file that was generated while performing the acceptance test on one of the S3S star trackers at UTIAS/SFL.

**CHAPTER 4. STAR TRACKER ALL-IN-ONE CONTROL INTERFACE**

**4.4.6 Control-Structure Manipulation**

For the attitude determination operation, S3S star trackers use settings stored in its memory space. The settings are defined as a control structure. When the star tracker initializes, the control structure stored in its NAND flash memory is copied into the error detection and correction (EDAC) protected memory, and the copy is used for the determination operation. The settings in the structure are easily readable and modifiable by accessing the appropriate addresses of the memory using the PEEK and POKE commands respectively. In STACI, the tasks required for checking and adjusting settings in the control structure are handled by a control structure manipulation task group handler and its implementation has been made in the CControlStructureTab class.

The user interface for the control-structure manipulation task handler has been designed in a simple and straightforward manner as shown in Figure 4.14. A list control is placed in the middle of the interface to show the list of settings in the control structure along with their values. To retrieve the most up-to-date values in the control structure and set new parameter values in the star tracker, two dedicated buttons – *Get All* and *Set Selected* respectively – were included. Detailed information on the control structure can be found in [26].

![Figure 4.13: An example of ACS result file showing quaternion and angular velocity measurements.](image)

**Figure 4.13:** An example of ACS result file showing quaternion and angular velocity measurements.
4.4.7 Data Downloading Operation

A download task handler was created in a CDownloadTab class. Its original motivation was to download images taken by the star tracker for the attitude determination operation. However, later on, a decision was made to extend the scope of the download handler to become a generic data downloader, and the extended requirement STACI 1.5 (Table 4.1) was defined because the images were nothing more than data placed in the specific memory locations with fixed sizes. Furthermore, the implementation for the extended requirement could be easily accomplished with relatively short development time while the gain was significant.

To run appropriately, the downloading operation requires a number of input parameters: a location, a start address, and the size of the data to be downloaded. Once the downloading operation is initiated, the download task handler manages the command creation and response handling processes appropriately according to the specified location of the data. In the S3S star tracker, three different types of memory exist: EDAC protected memory, random-access memory (RAM), and NAND flash memory. To download data located in the EDAC protected memory and RAM, the READ_EDAC and PEEK commands are used. A flow diagram describing the downloading process of data from EDAC protected memory and RAM is prepared in Figure 4.15.
For downloading data in the NAND flash memory, the previously introduced downloading algorithm could not be reused because the NAND flash memory is accessed on a page-by-page basis and the page worth of data must be copied into the star tracker’s internal NAND page buffer prior to downloading any of the data in the subject page. Therefore, an extra step was included in the algorithm. As a result, a separate downloading algorithm for NAND flash memory was developed in the download task handler design and it is illustrated in Figure 4.16 with the additional stages highlighted.

Having all the downloading algorithms developed, the task of downloading star tracker images can be easily done because retrieving those images is nothing more than downloading known-sized
data at specific locations. Therefore, in the download task handler user interface design shown in Figure 4.17, a section has been reserved for the image downloading operation along with the combo box containing the selections of the possible captured star tracker image locations within the memory to provide operators with an easy means of downloading images. In addition, the function that displays the images in the downloading process was added into the handler design to provide operators with the ability to verify the quality of the image while downloading is in progress, akin to the one developed for BRITE PGC.

![Image downloading operation section](image.png)

**Figure 4.17: Data downloading operation user interface.**

### 4.4.8 Firmware Update

Lastly, to deal with data uploading operations required to update flight software code and various catalogs (e.g., star maps) used for attitude determination operations on the S3S star tracker, the firmware update task handler has been implemented in STACI. The uploading is realized by writing a block of given source data into the appropriate location of the device memory. If all the data cannot be uploaded in a single packet, multiple packets are sent automatically. For the memory location, it varies depending on the type of source data: flight software is written into RAM, and catalog data is written into NAND flash memory.

Since the desired operation for the firmware update task handler is very similar to the data downloading process, with the exception that the handler needs to write instead of read, it was decided to reuse the algorithm developed for the download task handler (Section 4.4.7). However, all the read-related commands in the algorithm have been replaced with the corresponding commands for writing. Specifically, the PEEK command has been replaced with a POKE command for the RAM
writing operation, and READ_NAND_BUFFER and READ_NAND_PAGE commands have been substituted with WRITE_NAND_BUFFER and WRITE_NAND_PAGE commands respectively for NAND writing operation. Furthermore, the functional block that is used to store the downloaded data has been removed from the algorithm. The flow diagrams that describe the writing processes for the RAM and NAND flash memory are shown in Figure 4.18 and Figure 4.19 respectively.

**Figure 4.18:** Flow diagram of the writing process for the RAM.

**Figure 4.19:** Flow diagram of the writing process for the NAND flash memory.

The user interface for the firmware update task handler is shown in Figure 4.20. Its main function is to provide interfaces with the ability to select or enter three pieces of information that the writing algorithm requires to run: the source data (e.g., a file), the type of source data, and the target address to which the data needs to be written. For the target address, a list of pre-defined addresses is supplied to provide an intuitive selection method for both target memory types. Furthermore, a part
of the source data sometimes needs to be uploaded. To handle this situation, an interface that allows specifying the range of source data has been added into the design. To increase the flexibility of the write operation even more, the number of data bytes that can be uploaded via a single write command packet has been defined as a user-configurable parameter. Lastly, when the actual writing process on the device fails, the device sends back a not-acknowledged (Nack) response for the requested write command. The mechanism that detects this situation and aborts the on-going writing operation has been implemented into the algorithm along with the option (Stop on Nack) check box, which can toggle this error detection in the user interface shown below.

Figure 4.20: Firmware update operation user interface.
Chapter 5

Formation Flying User Interface

Formation Flying User Interface (FFUserInterface) is the ground segment software developed for the CanX-4/-5 mission at the Space Flight Laboratory. FFUserInterface serves as essential control software for the accomplishment of the formation flying objective. The software gives operators real-time access to all on-board parameters that are used for performing formation flying for both the CanX-4 and CanX-5 satellites. These parameters can be monitored and manipulated as required. In addition, FFUserInterface provides an interface to allow operators to interact with the GPS receivers on the satellites. Various commands can be sent to the receivers either manually (one at a time), or via a list of commands (contained in a script file) which can be sent to the receivers autonomously.

FFUserInterface has been developed with the intention of supporting all phases of the mission: development, commissioning, and nominal on-orbit operations. The first implementation of FFUserInterface has been released and was used as part of a hardware-in-the-loop test\(^5\) for the CanX-4/-5 formation flying simulation. With the ability to probe parameters and make real-time changes to their values during simulation, debugging was much easier. During the hardware-in-the-loop test, for

\(^5\) A CanX-4/-5 hardware-in-the-loop test is the verification and validation process of the navigation and control algorithm for the formation flying using the flight representative hardware setup. The test is conducted using various realistic test scenarios in the closed-loop with external truth model inputs. More information on the hardware-in-loop test can be found in [33].

60
example, the software helped to identify and correct a number of logical bugs that were present in the formation flying algorithm.

## 5.1 Software Requirements

A subset of the software requirements for FFUserInterface is listed in Table 5.1 below. Note that the acronym FUI in Table 5.1 refers to the Formation Flying User Interface. The complete set of requirements can be found in [28].

| FUI 1.1 | The FUI shall permit an operator to view all variables associated with the Exchange structures (see section 5.2). |
| FUI 1.3 | The FUI shall allow an operator to view a “snapshot” (one moment in time) of the current Exchange structure downloaded from the spacecraft. |
| FUI 1.7 | The FUI shall allow operators to view the current status of both CanX-4 and CanX-5. |
| FUI 1.9 | The FUI shall permit operators to create, edit, save, and re-load complete Exchange structures. |
| FUI 1.10 | The FUI shall permit operators to send groups of Set Exchange [AD6] commands in real time. There shall be a safeguard in place to prevent accidental or erroneous settings. |
| FUI 1.14 | The FUI shall be capable of controlling GPS receivers on both CanX-4 and CanX-5. |
| FUI 1.15 | The FUI shall allow operators to send arbitrary GPS command scripts (within the range of valid GPS commands) to the receivers. |
| FUI 1.20 | The FUI shall be capable of exporting Exchange structures to Formation Flying Analysis and Control Tool Set (FFACTS) as initial conditions for simulation. |

Table 5.1: FFUserInterface Software Requirements.

## 5.2 Parameters for Formation Flying

According to requirement FUI 1.1 (Table 5.1), FFUserInterface needs to be able to access variables in the Exchange structure. The Exchange structure is a set of parameters that are required to perform two distinct software algorithms running on CanX-4/-5 spacecraft for successful formation flying. The
two algorithms are the Formation Flying Integrated On-Board Nanosatellite Algorithm (FIONA) and
the Relative Navigation algorithm (RelNav). Each algorithm requires its own data structure for
execution referred to as the Formation Flying Exchange (FFExchange) structure and the GPS
Exchange (GPSExchange) structure, respectively. They are subcomponents of the overall Exchange
structure. Currently, there are 612 defined parameters in total that are accessible via the Exchange
structure. A detailed explanation of the algorithms and their data structures can be found in [29] and
[30] respectively.

5.3 GetExchange and SetExchange Commands

To identify the parameters in the exchange structure during various operations, each parameter is
assigned a unique identification number. The unique parameter ID numbers are referenced by
FFUserInterface when retrieving or adjusting the parameters’ values. The complete list of parameters
and their identification numbers within the exchange structure can be found in [30]. To allow the
ground segment software to retrieve and adjust parameter values in the exchange structure on the
CanX-4/-5 satellites, the Payload On-Board Computer (POBC) provides two different commands for
each operation, defined as GetExchange and SetExchange, respectively. Similar to other OBC
commands, both commands are defined in the NanoSatellite Protocol (NSP) format [12].

![Command structure for GetExchange and SetExchange commands.]

For both GetExchange and SetExchange commands, the parameters of interest are specified in a
parameter bitmask section as illustrated in Figure 5.1. Currently, the bitmask is 77 bytes (616 bits)
wide and each bit is used to define the specific parameter in the exchange structure. The byte index
number within the bitmask grows from the beginning of the command towards the end. Within each byte, the bits are numbered from the least significant to the most significant. For the GetExchange command, there is no limitation on the number of parameter bits that can be set within a single request packet. To respond to the GetExchange command, the POBC returns a packet containing the values of the requested parameters along with a bitmask that indicates the parameter numbers present in the response. If all requested parameters values cannot fit in a maximum-length NSP packet, the POBC returns the values in multiple response packets. For the SetExchange command, the parameters of interest are specified in the bitmask and the corresponding parameter values are appended after the bitmask section in ascending order of parameter number. The total size of the command packet must be smaller than the allowable NSP packet size. Otherwise, the request needs to be split into multiple SetExchange command packets and delivered to the POBC. To respond to the SetExchange command, the POBC returns a response packet with a bitmask indicating the parameters for which the POBC successfully received new values.

It is important to note that the new parameter values delivered to the POBC using SetExchange commands do not immediately affect the formation flying algorithms. Any new values are queued up in a buffer and held until a trigger signal is received by the POBC, at which point it will perform the actual “set” operation. This mechanism exists in order to deal with a situation where there is a group of parameters that need to be set simultaneously but it is not possible to set them all with a single SetExchange packet. For the trigger signal, a latch bit is defined in the parameter bitmask. The last parameter number that the bitmask can have is assigned to the latch bit (currently set to be 615). When the “set” operation is performed, the new values in the queue are cleared.

5.4 FFUserInterface Design Consideration and Decisions

5.4.1 GPS Receiver Command Format

The GPS receiver used in the CanX-4 and CanX-5 satellites is a commercial off-the-shelf OEMV-G1 module manufactured by NovAtel. To provide end users with flexible communication methods, NovAtel has designed the GPS module in such a way that it can communicate using many different command formats. The receiver supports commands in three different formats: abbreviated ASCII, ASCII, and binary. Since commands in abbreviated ASCII format are both human-readable and self-
explanatory, abbreviated ASCII command format was used for the GPS command operations within FFUserInterface. Furthermore, all development work with the GPS receiver, prior to the FFUserInterface development, had been done using the abbreviated ASCII format. Hence the operators were already familiar with this format. In addition, using abbreviated ASCII gave operators the ability to reuse already-created GPS scripts by FFUserInterface. This feature satisfies the software requirement FUI 1.15. Detailed information on the GPS receiver and the full list of its commands can be found in [31].

5.4.2 Implementation Recycling

Up until the author started developing FFUserInterface, many distinct software functional modules had been developed for various other projects including: BRITE Payload Control (PGC), BRITE Schedule, and Startracker All-in-one Interface (STACI). While the modules were being developed, an effort was made to design the modules in such a way as to make them reusable. The major advantage of this approach is that the development time of software projects can be significantly reduced by reusing pre-existing modules. The maturity of the existing modules is also carried over. A significant amount of the pre-existing modules were adapted and integrated during the development of FFUserInterface. Although minor modifications were required, the effort was worthwhile as it made the overall process faster. Information on recycled modules and their functional roles within the application can be found in the following section.

5.5 Software Architectural Design

The software architecture of FFUserInterface was developed using the same framework as was used for STACI (see Section 4.3). Figure 5.2 shows the top-level software architecture for FFUserInterface. The architecture is divided into two major blocks: the operational block and the functional block. Each block is comprised of a different set of classes, where each set serves a specific functional role within the application. The implementations in the operational block handle specific tasks that follow from the software requirements, along with a dedicated graphical user interface (GUI). For FFUserInterface, the implementations for managing the exchange structure and controlling the GPS receiver have been placed in the operational block under the formation flying control task group and the GPS operation task group respectively, as shown in Figure 5.2. In the functional block, there are
three modules that have been defined in the FFUserInterface design: the payload on-board computer (POBC) message handler, the communication handler, and the log handler. The POBC message handler is a group of classes that enables the application to manage the NSP command packets necessary for operation. The communication handler is in charge of sending these NSP commands and retrieving their responses to/from the POBC through TCP/IP. The received responses are then passed to the task groups in the operational block accordingly and handled in an appropriate manner. While the application is operational, all communication activities and actions are logged into a file for tracking purposes [32], as per requirement GSS-SYS-12. The logging process is performed by the log handler.

![Figure 5.2: Top-level software architecture for FFUserInterface](image)

For the three modules in the functional block, the designs developed for STACI were reused with minor modifications as mentioned in Section 5.4.2. Module designs specifically from STACI were used since at the point in time when FFUserInterface was being developed, STACI was the last project the author had worked on and it contained the most up-to-date implementations for each module. For the POBC message handler module design, extra time had to be allocated to ensure that the module was fully functional. A command library in the adapted message handler only had S3S startracker commands defined. All the necessary CanX-4/-5 POBC command definitions needed to be added to the command library. More information on the general message handler design can be found in Section 4.3.1.
5.5.1 Formation Flying Control Task Group

The formation flying control task group is a combination of classes that was developed to encapsulate software operations for managing and controlling the exchange structure used on the POBC for the CanX-4/-5 satellites. There are a total of three fundamental functional tasks that are required to be performed in the task group: setting, listing, and retrieving parameter values in the exchange structure. Unfortunately, detailed development could not be done in a straight-forward manner due to the complexity of the exchange structure architecture. Major challenges included designing a graphical user interface (GUI) as well as supporting architecture that provided an efficient way of managing a large number of parameters and their associated data. As a result, a data container class was created to store all the data required for handling a single exchange parameter during a software operation. A wrapper class was also created to manage the collection of the data containers. Both the data container and the wrapper classes are part of the formation flying control task group. Detailed information on the GUI can be found in Section 5.6.1.

5.5.1.1 A Data Container for Exchange Parameter

According to requirement FUI 1.9 (Table 5.1), the definition of the exchange structure and its parameters may be changed at a later date. The implementation of the data container class must therefore be flexible enough to adapt to these types of future changes without requiring modifications to the source code. Hence, the definition of the variables used to hold the data for each exchange parameter could not be hardcoded. Instead, they are handled dynamically during the runtime. Furthermore, multiple variables are used in order to have complete control over a single exchange parameter for the software operation. This is because some exchange parameters are composed of multiple elements. Extra variables are required to store the definition of those elements (e.g., data type and size). Moreover, the software itself had to reserve some working variables in order to perform the operations desired in the software requirements. For example, variables that store information on allowable bounds, possible discrete values, and a description for the associated exchange parameter needed to be placed in the class. As a result, the CFFExchangeStructureElement class was designed to encompass all required variables for single exchange parameter. Table 5.2 shows the defined variables along with their descriptions.
<table>
<thead>
<tr>
<th>#</th>
<th>Variable Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m_last_updated_time_</td>
<td>CTime</td>
<td>Stores the last time that m_parameter_value_ variable is updated.</td>
</tr>
<tr>
<td>2</td>
<td>str_parameter_name_</td>
<td>CString</td>
<td>Stores the name of the parameter.</td>
</tr>
<tr>
<td>3</td>
<td>str_parameter_location_</td>
<td>CString</td>
<td>Stores the location of the parameter in an exchange structure.</td>
</tr>
<tr>
<td>4</td>
<td>str_parameter_type_</td>
<td>CString</td>
<td>Stores the structure type of the parameter.</td>
</tr>
<tr>
<td>5</td>
<td>b_parameter_readonly_</td>
<td>Boolean</td>
<td>Indicates whether the parameter is read-only.</td>
</tr>
<tr>
<td>6</td>
<td>m_parameter_type_</td>
<td>CUIntArray</td>
<td>Stores the types for each element in the parameter.</td>
</tr>
<tr>
<td>7</td>
<td>m_parameter_size_</td>
<td>CUIntArray</td>
<td>Stores the sizes for each element in the parameter.</td>
</tr>
<tr>
<td>8</td>
<td>m_parameter_value_</td>
<td>CByteArray</td>
<td>Stores the last retrieved element values in the parameter from a spacecraft.</td>
</tr>
<tr>
<td>9</td>
<td>m_parameter_new_value_</td>
<td>CByteArray</td>
<td>Stores the temporary element values in the parameter for software operations.</td>
</tr>
<tr>
<td>10</td>
<td>m_parameter_status_</td>
<td>CUIntArray</td>
<td>Stores the status for each element in the parameter; it is a result of a sanity check performed when the element values are retrieved from a spacecraft. It indicates whether or not the values are within the expected range.</td>
</tr>
<tr>
<td>11</td>
<td>m_parameter_boundary_</td>
<td>CStringArray</td>
<td>Stores the allowable bounds or discrete values for each element in the parameter.</td>
</tr>
<tr>
<td>12</td>
<td>m_parameter_description_</td>
<td>CStringArray</td>
<td>Stores the descriptions about each element in the parameter.</td>
</tr>
</tbody>
</table>

Table 5.2: Variables in CFFExchangeStructureElement class and descriptions.
There are two important design decisions that should be highlighted regarding the variables listed in Table 5.2. Firstly, array type objects are used for variables holding exchange parameter data (variables with a prefix of the fixed string ‘m_parameter_’ in their name, items #6 to #12 in Table 5.2). This is because the size of arrays are not required to be defined when they are created – they can be expanded or reduced dynamically. Furthermore, for parameters with multiple members, the information is inserted into the appropriate arrays in the sequential order that is specified in the exchange parameter definition. The second decision is to have an extra copy of the variable that holds values for the exchange parameter: m_parameter_new_value_. The purpose of these extra parameters is to reserve a dedicated data buffer that the software can use to store new values for a specific exchange parameter during the SetExchange operation. Moreover, by having the buffer, the most up-to-date parameter values retrieved from the spacecraft could stay untouched and be accessed anytime as necessary during software operation.

5.5.1.2 Exchange Parameter Manager

As previously mentioned, the parameter definitions in the exchange structure could not be hardcoded as part of the software source code. This information must be provided from an external source and has to be loaded into FFUserInterface during software initialization. To deal with the loading process, an exchange structure manager class is implemented in the formation flying control task group. While the manager loads the definition file, parameter data container objects are dynamically created along with the variables within the objects. The references for all created objects are then kept in a container that is placed in the manager. These references are used throughout software operations. When the references are being inserted into the container, the manager handles them in such a way that the location index of the references in the container matches the actual parameter number. This has been done to allow the data containers to be accessed directly using the actual parameter numbers without having to execute a search algorithm. Eliminating the search algorithm is beneficial because the parameter data containers are frequently accessed during software operation due to the characteristics of the application. By eliminating the need for a search algorithm, the accessing time is greatly reduced.
5.5.2 GPS Operation Task Group

The operations to control the GPS receivers on-board the CanX-4/-5 satellites are handled by the GPS operation task group in FFUserInterface. Unlike the formation flying control task group, the implementation of the GPS operation task group was made in a single class. This was because functions which were required to satisfy the software requirements within the task group were simple and it was possible to combine them with GUI handling functions. The controlling operation is accomplished by sending the appropriate GPS commands to the receivers (see Section 5.4.1 for more details). The receivers do not use the NanoSatellite Protocol (NSP) for communication. All GPS commands, however, must be encoded in NSP format prior to transmission. The NSP-formatted GPS commands are later converted back to the original format and delivered to the receiver by the POBC on-board the satellite.

5.6 Software User Interface Design & Implementation

The graphical user interface (GUI) for FFUserInterface was developed using similar design practices as those used for the development of the other ground segment control software introduced in the previous chapters. For the base of the GUI design, a single document interface (SDI) framework was employed. The control components were laid on top of this framework. The components that are used commonly during nominal operations were placed in the main dialog window. Furthermore, a large portion of space was reserved in the middle of the main window for a tab control. The tab control provides a way to link separately developed GUIs to the main dialog window. The GUIs for the operational blocks (formation flying control task group and GPS operation task group) were connected via the tab control. Icon buttons were used to optimize limited window space. The latest GUI design for FFUserInterface is shown in Figure 5.3.
5.6.1 Formation Flying Control Task Group

The fundamental principle for the formation flying control task group GUI design was to provide a convenient method of manipulating parameter values within the exchange structure. To do so, a GUI that is able to provide an effective way of listing and sorting the large number of parameters existing in the exchange structure needed to be designed as the first step. A design decision was made to make this listing interface the main entry point to other interfaces in the task group. This is because the listing operation was identified as being essential to both retrieving and setting operations. Making the listing interface the main interface allowed for a smooth operation flow. The GUI design for the listing operation is linked to the tab control in the main dialog window, and is expanded in Figure 5.4 for legibility.
CHAPTER 5. FORMATION FLYING USER INTERFACE

Figure 5.4: Listing user interface showing all updated exchange parameters in Fiona_output group.

In the middle of the interface is a list control which is used to display brief information on the filtered exchange parameters according to the selection in the Group and Show combo-box controls (they are placed directly above the list control in Figure 5.4). As the name implies, the Group combo-box control allows operators to toggle the list to show a set of the parameters defined as a group definition. The group definitions are user definable and they can be managed through the interface FFExchange group editor. Currently, there is no limitation on the maximum number of groups. The group definitions are stored into the loaded exchange structure definition file for operations. This design decision was made because group definitions are be specific to the working exchange structure. Exchange parameters in the list can be further filtered using the criteria defined in the Show combo-box control. This control provides options to screen out parameters based on their values at the point of time when the filter is applied. At present, there are four different filters: never updated, updated, out of bounds, and bounded. In addition, the criteria in the Show combo-box are pre-defined as part of the application source code, unlike group definitions. Next, searching functionality was included as part of the listing interface. During development, it was found that in some cases the size of the parameter list was still significantly large even though a number of items would be screened out.
(using the filters). It was difficult for operators to find desired parameters within the list by their name, especially when the names are similar to each other. In order to improve convenience and efficiency, the ability to perform a text-based search for parameters was implemented. Lastly, the parameter list was implemented in such a way that each entry of the list was able to represent the status of the corresponding parameter by varying its colour. This colour code uses four different colours: white for not updated, yellow for querying, green for updated and valid (i.e., values are within bounds), and red for updated and invalid (i.e., values are out of bounds).

5.6.2 Get Operation

Within the formation flying control task group user interface design, there are two access points that are reserved to allow operators the ability to perform retrieving operations for exchange parameter values. The first and primary access point is placed in the listing interface dialog. This design decision is derived from requirement FUI 1.3 and allows the operator to capture a snapshot value of FFExchange parameters. This is useful for monitoring the formation flying algorithm status. By having access on the listing interface dialog window where the list of items is configurable and colour-coded, monitoring can be done more effectively. Secondly, the retrieval operation can be performed through the FFExchange parameter editor. The FFExchange parameter editor is a pop-up dialog window that is implemented in order to allow operators to browse detailed information on selected exchange parameters. This is distinctly different from the list interface that provides a way to check only a snapshot parameters value status. Using the retrieval operation, changes in the parameter values can be monitored within the editor window. Furthermore, it has become more convenient for operators as they are no longer required to continually switch between windows to get updated parameter values. The current design of the FFExchange parameter editor is shown below in Figure 5.5.
5.6.3 Set Exchange Parameter Operation

In order to set new exchange parameter values on the spacecraft, the desired values need to be entered and buffered into the application prior to performing the actual commanding process. The first step of the set exchange parameter operation was to design a GUI that would allow operators to access and enter values for all individual components in the exchange parameter. Fortunately, by the time the design requirement for the set operation was identified, an FFExchange parameter editor that could handle the majority of the desired actions was already developed. Few extra user interfaces and functions needed to be added to the existing parameter editor; as a result, the first development step for the set operation was realized very quickly.

During development, it was found that the process of entering each parameter value manually was inconvenient and time consuming, which is clearly not desirable for nominal mission operations. The functions designed to handle the input process in an autonomous manner had to be implemented as part of the parameter editor. As a result, two different user interfaces were developed. First, an input field was included to allow operators to enter arrays of values in a long text format. The text format used by MATLAB was employed because currently all formation flying simulations are
planned to be performed using MATLAB on the ground. Therefore, operators can simply copy and paste the desired values (e.g., control parameter values) into the application ensuring a fast and convenient process. Secondly, the interface that allows saving and loading a set of new exchange parameter values have been placed in the parameter editor design. This addresses the situation where operators need to set the same group of parameter values repeatedly. For instance, this functionality was used frequently during hardware-in-the-loop testing [33] as it provided a convenient way to setup initial conditions for various simulations for the onboard formation flying algorithm.

The set exchange commanding operations update the actual exchange parameter values on the spacecraft with the buffered ones. This is accomplished through an additional dialog window (shown in Figure 5.6) that is linked to the FFExchange parameter editor. Although it seems like an extra step in overall SetExchange operation was being introduced here, this was done on purpose to safeguard against erroneous parameter values being sent to the spacecraft from accidental operator actions. This is because some parameters define the behavior of the onboard formation flying algorithm and erroneously altering those settings can possibly put the formation flying operations in an unrecoverable state.

![Figure 5.6: Safeguard user interface added as part of SetExchange operation.](image-url)
5.6.4 GPS Operation Task Group

In comparison to the complexity of the formation flying control task group's user interface design which is comprised of multiple components, the user interface for the GPS operation task group is very simple. A single dialog window (directly linked to the tab control in the main dialog window) was used to capture all required user interfaces as shown in Figure 5.7. The fundamental operation performed in the GPS operation dialog is sending GPS commands to the spacecraft. Using the interface in the upper half of the dialog, arbitrary GPS commands can be entered and sent manually. However, only one command can be managed at a time. To handle the case where multiple commands need to be entered and sent as a group, a command queue was created in the bottom half of the dialog. Using this command queue, a series of GPS commands can be built up and sent sequentially in an autonomous manner. Furthermore, the command queue was implemented in such a way that its contents can be saved and reloaded through a file called a ‘GPS command script’. It is a simple human-readable text file and the commands are stored on every line of the file in the same order as they appear in the command queue. Moreover, since the script files are reusable, they are convenient for performing repetitive GPS operations.

![Figure 5.7: GPS operation user interface.](image-url)
Chapter 6

GNB Whole Orbit Data Parser

All operational spacecraft developed at SFL periodically collect telemetry from each subsystem and store them into a file (called whole orbit data, or WOD) autonomously. These WOD files are later downlinked to the ground station during contact. They are then analyzed to obtain a significant amount of on-orbit information which is used for different purposes (e.g., spacecraft health monitoring and attitude control system performance verification). When a WOD file is generated using telemetry from different subsystems, each set is placed in a pre-defined structure and is appended so that it can be easily extracted at the ground station.

To provide a means of parsing the WOD files, a program called “GNBChopper” has been developed as part of the author’s thesis work at SFL. In short, GNBChopper takes a file that contains WOD expressed in binary (raw) format as an input and produces files with the corresponding raw WOD expressed in human-readable (parsed) format. While the raw WOD files are being parsed, GNBChopper automatically converts the raw telemetry from analog-to-digital units (ADU) to engineering units as required. As the name of the software implies, GNBChopper can only handle the raw WOD files generated from Generic Nanosatellite Bus (GNB) spacecraft. It is being used in various GNB missions and for different purposes including testing and debugging for the CanX-4/-5 and BRITE missions, as well as for the on-orbit operation of AISSat-1 [34].
6.1 Software Requirements

Originally, GNBChopper had been designed as CanX-4/-5 mission ground segment software and it was given the name Formation Flying Chopper (FFChop). FFChop was primarily required to parse a series of formation flying exchange (FFExchange) structure values from debugging files. These debugging files were generated during the debugging phase of the development of the formation flying algorithm (FIONA). Further information about FFExchange and FIONA can be found in [30] and [29] respectively. Since the files were generated for the purpose of debugging, the content of the files was not always the same. Therefore, FFChop had been developed with the capability of parsing files with different user-defined structures. Furthermore, during the development of FFChop, it was found that the debugging files were constructed using the same structure as the GNB-WOD files, and therefore FFChop could also be used to parse telemetry in the WOD files. As a result, it was decided to extend the scope of FFChop to become a generic WOD file parser for GNB based missions. Thus, FFChop has become GNBChopper. The complete set of requirements that were used for FFChop development is given in Table 6.1 below.

<table>
<thead>
<tr>
<th>FFChop.1</th>
<th>FF Chop shall be capable of parsing raw FIONA, RelNav, GPS and ADCS WOD and placing it into text files with a pre-defined, user-configurable structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFChop.2</td>
<td>FF Chop shall be capable of generating output files with configurable, time-dependent start and end points.</td>
</tr>
<tr>
<td>FFChop.3</td>
<td>FF Chop shall not be responsible for parsing/interpreting raw GPS data, but shall be able to download it and transform the downloaded data into a format that is suitable for input to NovAtel tools.</td>
</tr>
</tbody>
</table>

Table 6.1: FFChop software requirements.

6.2 Structure of WOD file and WOD Group

When telemetry from subsystems in a GNB spacecraft is collected and stored in a WOD file, it is encapsulated into a structure referred to as a WOD group structure. This pre-defined structure allows telemetry to be easily parsed from the WOD file for later use. The WOD group structure includes four additional pieces of information along with the actual telemetry as described in Figure 6.1 (numbers in the figure represent the number of bytes for the corresponding information).
The beginning of the structure is a two-byte-long header that indicates the start of the WOD group entry in a given WOD file. It is defined to always be 0xFFFF (hexadecimal). The following elements in the structure are the type and time information of the entry, which are two bytes and eight bytes long, respectively. The type information is a unique identification number that the spacecraft uses to distinguish one subsystem from another when it collects telemetry from various subsystems. The time field specifies the retrieval time for the given set of telemetry. It is represented in 64 bit-long J2000 time in milliseconds. Next is the actual telemetry that is collected from the subsystems. Its size varies among the subsystems. The final element is an error-detection code. It provides a means of verifying the integrity of the data in the group entry (16-bit cyclic redundancy check (CRC16) code is employed to achieve the error detection for the data in the type, time, and telemetry data fields). The complete WOD file is simply a concatenated sequence of this WOD group structure.

6.3 Software Design and Implementation

6.3.1 Parsing Engine

To take care of the entire parsing operation process in an efficient manner, a dedicated functional algorithm (engine) has been developed as part of the application. For simplification of the design, the engine has been designed in such a way that it parses only one WOD group at a time in the given raw WOD file. For the situation where multiple groups are desired to be parsed, the engine is executed in a recursive manner.
When the parsing engine is initiated, it walks through the given raw WOD file from one group header to the other looking for the desired WOD group entry. All the valid entries are then stored in the memory reserve for the rest of the operation. Once the scanning process has completed, all valid entries in the memory are sorted chronologically using the data given in their time field. Even though it is assumed that the entries in the raw WOD files are appended in chronological order when they are produced in the spacecraft, the sorting process is still performed as a means of reassurance. For the last step, the telemetry in the parsed group entries are appended into the appropriate output file on a per line basis. All output files that are generated during the parsing operation follow the specific naming convention defined for all GNB missions. The details of the naming convention will be discussed in Section 6.3.2.

As the parsing engine scans through the raw WOD files to extract the desired group entries, it walks through the files on a byte-by-byte basis. This process requires remarkable computing power and as the size of the WOD files get larger it may cause the application to become unresponsive. To deal with this situation, a multithreading mechanism has been employed and the parsing engine has been designed to always be run on a thread separate from the main application thread.

### 6.3.2 Output File Management

All output files produced by GNBChopper during the parsing operation are named following the convention defined in [35]. The filename is composed of the type number and type name of the WOD group that the file contains and the date used by the time stamps in the raw WOD file. Therefore, if a raw WOD file contains data spanning multiple days, multiple output files for each WOD group will be produced. The file creation for the output files occurs dynamically during the parsing operation when the parsing engine tries to append the parsed data. If the engine cannot find the file that has the appropriate name for the data, a new file is created. Otherwise, the data is appended to an existing file. Furthermore, to provide a convenient and practical way to manage the output files, archiving functionality has been implemented into the parsing engine. When this functionality is disabled, all newly generated output files are placed in the root directory that is specified in the application options. When the function is enabled, the parsing engine automatically creates separate directories in the root directory according to date, and places the files into them accordingly.
6.3.3 Configuration for Parsing Operation

To meet software requirements FFChop.2 and FFChop.3 (Table 6.1), that desire the parsing operation to be highly user-configurable, a set of settings has been defined that store all information necessary to configure the behaviour of the parsing engine. This set is referred to as the parsing configuration and is composed of two sub-configurations: the input data structure and the output data structure.

As its name implies, the input data structure configuration is comprised of information that GNBChopper requires to extract the WOD groups out from given raw WOD files. A set of WOD group definitions is stored in the input data structure configuration. For each group definition, the type identification number and ‘structural information’ of the telemetry that appears in the corresponding WOD group entry are specified. ‘Structural information’ refers to binary data in the telemetry field that are concatenated in a pre-defined sequence without data representation information (i.e., size and type). The data representation information is essential in order to obtain the original values during the parsing operation. This information must be supplied externally to GNBChopper and be specified in the input data structure configuration.

Next, information that GNBChopper requires while producing its output files is specified in the output data structure configuration. A unique copy of this configuration is assigned to each WOD group definition defined in the input data structure configuration. Each configuration contains seven different settings that are used to define the format of the output file and the structure of its contents. Table 6.2 shows all the settings and their respective descriptions.

<table>
<thead>
<tr>
<th>#</th>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>File extension</td>
<td>The extension of the output file containing parsed telemetry of the assigned WOD group.</td>
</tr>
<tr>
<td>2</td>
<td>Header</td>
<td>The text that is included at the top of the output file before the telemetry entries.</td>
</tr>
<tr>
<td>3</td>
<td>Delimiter</td>
<td>The character that is used as the delimiter for the parsed telemetry appended in the output file.</td>
</tr>
<tr>
<td></td>
<td>Settings in the output data configuration and descriptions.</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Data column definition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The settings for the type and the order of parsed telemetry presented in the output file. A single line of text containing the telemetry is prepared. Furthermore, each entry is separated using the character specified in the delimiter setting.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Include column title</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A boolean type setting. When it is set, a line of text that contains the titles for the columns in the column definition is inserted following the header. The column titles are separated using the character specified in the delimiter setting.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Unit conversion definition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The type and coefficients for the unit conversion process that is performed on the raw telemetry. A unique unit conversion definition is assigned to each data column.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Precision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The number of decimal places for the parsed telemetry readings. Each data column is assigned its own precision setting.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Settings in the output data configuration and descriptions.

Figure 6.2: Manual parser user interface with a raw WOD file loaded for parsing.
6.4 Operation and User Interface

6.4.1 Parsing Operation

GNBChopper offers two different user interfaces to handle the parsing operation: the manual parser and the automatic parser. The fundamental difference between those two user interfaces is the way that raw WOD files are selected and parsed. The manual parser, as shown in Figure 6.2, allows only one raw WOD file to be selected and parsed at a time. For multiple input files, operators have to manually handle all of them in a repetitious manner which can be inconvenient. However, there are advantages of using the manual parser. When a raw WOD file is selected and successfully loaded into the manual parser, detailed information about the loaded WOD file is listed in the interface, as shown in Figure 6.2. Data counts for each WOD group present in the loaded WOD file are displayed along with the corresponding timestamps for the first and last data entries. Furthermore, the interface lets operators specify the desired WOD groups for the parsing operation.

![Figure 6.3: Automatic parser user interface with its operation enabled.](image)

In contrast to the manual parser, the automatic parser has been designed to allow operators to parse multiple raw WOD files in an autonomous manner. For the parsing operation, a specific directory that contains raw WOD files is given to the automatic parser rather than a list of the specific filenames. Once the automatic operation is enabled (achieved by setting the Enable Automatic Operation check box, as shown Figure 6.3), the parser scans the given directory and processes all the raw WOD files one by one if any are found. The parsed input files are then moved to the archive directory that is specified in the application options. Once all files present in the given directory are
completely parsed and moved, the automatic parser continues to scan the directory in the anticipation that new WOD files will be added. Any new files added to the directory are immediately processed and moved to the archive directory. This behavior is especially advantageous when GNBChopper is combined with other ground segment software. For instance, there is a real-time WOD downloader developed at SFL which is used to retrieve raw WOD files from a spacecraft and place them into a certain directory in an autonomous manner. If the download directory is set as the same one that the GNBChopper scans for the automatic parsing operation, any downloaded WOD files will be parsed at the same time that they are downloaded. Therefore, all the processes from WOD retrieval to parsing can be done in an autonomous manner resulting in increased autonomy and efficiency for ground station operations.

### 6.4.2 Configuration file and ChoppingBoard

To provide an easy way to create and manage the parsing configurations introduced in Section 6.3.3, a dedicated graphical user interface (GUI) has been implemented in the GNBChopper design. The motivation for having this GUI arose from the inconvenience that the author had experienced while using other SFL developed ground segment software which perpetually required extensive configuration. The software did not have any associated interfaces that could be used to generate the configurations – they had to be generated externally using third party software (e.g., a plain text editor). Furthermore, significant time and effort had to be spent to generate the proper configuration files as there were no available documents that explicitly described the correct formats.

The configuration managing tool implemented for the GNBChopper is called ChoppingBoard. ChoppingBoard takes care of all the file handling processes required while creating, importing, and exporting the configuration files. Thereby, operators are totally insulated from the complex formatting process. When ChoppingBoard creates or exports the configuration files, it embeds a unique number referred to as a version identifier. The version identifier specifies the structure of the configuration that is used to store the settings in the file. The main reason for using the version identifier is to keep track of the changes made in the configuration structure since the first release of the software. Typically, changes are made while introducing or enhancing new or existing functionality to fulfill additional software requirements. The definition of the all different configuration structure versions are hardcoded in the ChoppingBoard implementation. The
appropriate function calls are executed according to the version identifier number as the configuration files are imported into ChoppingBoard for operation. Furthermore, all the configurations are backwards compatible – i.e., all the older versions of the configuration file are converted to the most-up-to-date version during the loading process.

The user interfaces in ChoppingBoard have been constructed in such a way that they can provide a convenient workspace to manage the configurations for a single WOD group. There are two dedicated tab interfaces in ChoppingBoard that allow the operator to manage all the settings in the input data configuration (Figure 6.4) and the output data configuration (Figure 6.5). In addition, to allow operators to browse all the available (created and defined) WOD groups and to choose one for editing, a list control interface has been placed in the top-left corner of the ChoppingBoard interface, as shown in Figure 6.4. It is important to note that when the configuration file is imported and its settings are being managed, all changes are made to a virtual copy of the file which is kept in memory. Operators have to manually export the modified configuration settings into the file in order for the changes to take effect. This approach is advantageous because it provides a method to revert the settings back to the original states in case any issues arise during operation (e.g., operational mistakes or abnormal software behavior). Furthermore, the design provides a safeguard mechanism to prevent the same configuration file from being accessed at the same time for both the parsing operation and the editing operation.

![ChoppingBoard interface](image)

**Figure 6.4:** ChoppingBoard showing input data structure of S-band WOD group in loaded sample file.
Figure 6.5: ChoppingBoard showing output data structure of S-band WOD group in loaded sample file.
Chapter 7

Conclusion

Five different ground segment software programs have been presented. The author has developed these for two nanosatellite space missions at the UTIAS Space Flight Laboratory. For the CanX-3 mission, BRITE Payload Control (PGC), BRITE Schedule, and the Star Tracker All-in-one Control Interface (STACI) have all been described. For the CanX-4/-5 mission, the Formation Flying User Interface (FFUserInterface) and GNB Chopper have been discussed. Each chapter covers the design details for specific software. The chapters have been presented in the same order that the software has been developed. Furthermore, their architectural and graphical user interface design and development have been discussed along with the design considerations and resulting decisions made during development.

Each piece of ground segment software has undergone a verification and validation. All designs and implementations (i.e., algorithms, user interfaces, and functions) have been carefully tested and debugged using the actual hardware units (e.g., BRITE instrument and S3S star tracker) and datasets (e.g., raw WOD files). In this manner, each software program has been verified against its requirements. In addition, due to the highly dynamic and iterative nature of the software development process, the verification and validation has been performed in parallel with actual software development.

The full development cycle for all five pieces of software has been completed. Each one has demonstrated that it meets its corresponding requirements. The software programs are currently
being used to perform various tasks at SFL. BRITE PGC and STACI have been used to perform acceptance tests for the BRITE instrument and the S3S star tracker, respectively. FFUserInterface has been used while testing and debugging the formation flying algorithm (FIONA) running on the payload computer and has successfully identified numerous software bugs that have previously gone unnoticed. GNBChopper has been used among various GNB missions to parse raw whole orbit data files for various purposes including testing and debugging for the CanX-3 and CanX-4/-5 missions and also for actual on-orbit operations for the AISSat-1 mission. At present, BRITE Schedule has not yet been used for real applications since it is a mission-specific, on-orbit task scheduler. However, the software has been fully tested and verified on the ground and is awaiting the first BRITE spacecraft to be launched into space.

Ground segment software design and development for nanosatellite space missions is not a straightforward process. A developer cannot simply implement a ready-made tool that easily communicates with hardware units. Many different design criteria need to be carefully identified, examined, and considered. A series of design decisions must then be made in order to resolve unique challenges that arise during development. In addition, many iterations of testing, debugging, and process modification must be performed in order to make the implementations more robust, reliable, convenient, and intuitive. It is hoped that the author’s work described in this document contributes to the success of the CanX-3 and CanX-4/5 missions as well as to the building of a strong foundation for ground segment software that can be used for future nanosatellite space missions at UTIAS/SFL.
Bibliography


[18] Analog Devices, Complete 14-Bit 30 MSPS Complete 14-Bit 30 MSPS, Rev.0 AD9824.


