Spacecraft Datensimulator für den DESTINY$^+$ Dust Analyser

Spacecraft Data Simulator for the DESTINY$^+$ Dust Analyser

Bachelor Thesis of

cand. aer. Rafael Kniese

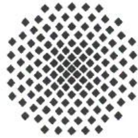
IRS-21-S-100

Examiner:
Priv.-Lect. Dr.-Ing. Ralf Srama

Supervisor:
Dipl.-Ing. (FH) Stephan Ingerl, M.Sc.

Institut für Raumfahrtsysteme, Universität Stuttgart
Institute of Space Systems, University of Stuttgart
August 2021

Exemplar für das Institut



UNIVERSITÄT STUTTGART
INSTITUTE OF SPACE SYSTEMS

Pfaffenwaldring 29 · 70569 Stuttgart · Phone +49 (0) 711 685-62375 · Fax: +49 (0) 711 685-63596 · www.irs.uni-stuttgart.de

Bachelor Thesis Work

of Mr. Rafael Kniese

Spacecraft Datensimulator für den Destiny+ Dust Analyser

Spacecraft data simulator for the Destiny+ Dust Analyser

Motivation:

DESTINY+ is a deep space mission of the space agency ISAS/JAXA (Japan). The launch of the electrically propelled space probe is planned for 2024. After swing-by maneuvers on the Moon, it is scheduled to reach the active asteroid Phaethon in 2028 and study it during the fly-by. In addition to testing new technologies, scientific issues will also be investigated. Part of the scientific payload is a new dust telescope called DESTINY^{+} Dust Analyser (DDA), which will characterize interplanetary and interstellar microparticles. In particular, the chemical composition is to be precisely determined by mass spectrometry. The dust telescope will be developed at the IRS and built and tested together with industry.

DDA communicates with the spacecraft via a single SpaceWire bus. Telecommands, telemetry and broadcasts data packets are exchanged via this bus. All packages have a certain data frame, data protocol and timing behavior. For DDA, the flight software is responsible for the SpaceWire communication. To improve the testability of this DDA data interface it is important to be able to simulate the spacecraft SpaceWire data bus. A simulator needs to be developed that emulates the behavior of the spacecraft.

The goal of this work is to develop a spacecraft data bus simulator by using a MK3 SpaceWire Brick (a SpaceWire to USB converter). The requirements for the simulator are to be elaborated and the necessary tools are to be selected. A concept for the simulator is to be established and developed with the help of the selected toolchain. The spacecraft data simulator will then be evaluated with the DDA flight software.

Task:

- Prepare a spacecraft data simulator concept
- Select a suitable toolchain
- Develop a spacecraft data simulator
- Evaluation of the spacecraft data simulator
- Documentation

Supervisor: Stephan Ingerl

Starting date: 12.04.2021

Submission until: 11.08.2021

Acknowledgement of receipt:
I hereby confirm that I read and understood the task of the bachelor thesis, the juridical regulations as well as the study- and exam regulations.

R. Srama

Date
PD Dr.-Ing. Ralf Srama
(Responsible Professor)

12. April 2021, Rafael Kniese

Date
Signature of the student

Legal Restrictions: The author/s of the bachelor thesis is/are not entitled to make any work and research results which he/she receives in the process of writing this thesis accessible to third parties without the permission of the named supervisors. The author/s shall respect restrictions related to research results for which copyright and related rights already exist (Federal Law Gazette I / S. 1273, Copyright Protection Act of 09.09.1965). The author has the right to publish his/her findings as long as they incorporate no findings from the supervising institutions and companies for which restrictions exist. The author must consider the rules and exam regulations issued by the university and faculty of the branch of study where the bachelor thesis was completed.

IRS Professors and Associate Professors:
Prof. Dr.-Ing. Stefanos Fasoulas (Managing Director) · Prof. Dr.-Ing. Sabine Klinkner (Deputy Director) · Prof. Dr. rer. nat. Alfred Krabbe · (Deputy Director) · Hon.-Prof. Dr.-Ing. Jens Eickhoff · Prof. Dr. rer. nat. Reinhold Ewald · PD Dr.-Ing. Georg Herdrich · Hon.-Prof. Dr. Volker Liebig · Hon.-Prof. Dr. rer.nat. Christoph Nöldeke · Prof. Dr.-Ing. Stefan Schlechtriem · PD Dr.-Ing. Ralf Srama

Declaration

I, ***Kniese***, **Rafael** hereby certify that I have written this ***Bachelor thesis*** independently with the support of the supervisor, and I did not use any resources apart from those specified. The thesis, or substantial components of it, has not been submitted as part of graded course work at this or any other educational institution.

I also declare that during the preparation of this thesis I have followed the appropriate regulations regarding copyright for the use of external content, according to the rules of good scientific and academic practice[1]. I have included unambiguous references for any external content (such as images, drawings, text passages etc.), and in cases for which approval is required for the use of this material, I have obtained the approval of the owner for the use of this content in my thesis. I am aware that I am responsible in the case of conscious negligence of these responsibilities.

Stuttgart, August 3, 2021, Rafael Kniese
....
Place, Date, Signature

I hereby agree that my ***Bachelor thesis*** with the following title:

Spacecraft data simulator for the Destiny+ Dust Analyser

is archived and publicly available in the library of the Institute of Space Systems of the University of Stuttgart ***without blocking period*** and that the thesis is available on the website of the institute as well as in the online catalogue of the library of the University of Stuttgart. The latter means that bibliographic data of the thesis (title, author, year of publication, etc.) is permanently and worldwide available.

After finishing the work, I will, for this purpose, deliver a further copy of the thesis along with the examination copy, as well as a digital version.

I transfer the proprietary of these additional copies to the University of Stuttgart. I concede that the thesis and the results generated within the scope of this work can be used free of cost and of temporal and geographical restrictions for the purpose of research and teaching to the institute of Space Systems. If there exist utilization right agreements related to the thesis from the institute or third parties, then these agreements also apply for the results developed in the scope of this thesis.

Stuttgart, August 3, 2021, Rafael Kniese
....
Place, Date, Signature

[1] Stated in the DFG recommendations for „Assurance of Good Scientific Practice "or in the statute of the University of Stuttgart for „Ensuring the Integrity of Scientific Practice and the Handling of Misconduct in Science "

Abstract

This bachelor thesis reports on the development of a data simulator for the dust telescope "DESTINY$^+$ Dust Analyser (DDA)". The DDA is part of DESTINY$^+$, a space mission to the asteroid 3200 Phaethon, the presumed parent body of the Geminids. It will analyze cosmic dust released by the asteroid to better understand its role as a source of organic material on earth.

The DDA will communicate with the rest of the satellite using the SpaceWire bus system and will be tested by the data simulator developed in this thesis, so that possible errors in the flight software can be detected and corrected as soon as possible. The need for this arises from the fact that the DDA is being realized at the Institute of Space Systems of the University of Stuttgart (IRS) together with the electronics supplier "von Hoerner & Sulger", while DESTINY$^+$ is a mission of the Japan Aerospace Exploration Agency (JAXA). The German Aerospace Center (DLR) is the German project sponsor. The geographical distance between the experiment and the spacecraft is too large to perform direct tests in the early stages of the mission.

The data simulator is implemented as two Graphical User Interfaces (GUIs) in C/ C++ in the Microsoft Visual Studio 2019 environment on Microsoft Windows 10. One sends data as broadcast or telecommand, the other receives telemetry. A SpaceWire Brick Mk3 from STAR-Dundee is used as the interface between the PC and the bus, which has two ports for SpaceWire cables and a USB port. Thus the data stream from and to the DDA can be controlled from a PC. The messages follow the packet protocol of the Consultative Committee for Space Data Systems (CCSDS). In the future, however, they will be adapted to the Remote Memory Access Protocol (RMAP).

Kurzfassung (German Abstract)

Diese Bachelorarbeit berichtet über die Entwicklung eines Datensimulators für das Staubteleskop „DESTINY$^+$ Dust Analyser (DDA)“. Der DDA ist Teil von DESTINY$^+$, einer Raumfahrtmission zum Asteroiden 3200 Phaethon, dem vermutlichen Urprungskörper der Geminiden. Er soll den vom Asteroiden freigesetzten kosmischen Staub analysieren, um dessen Rolle als Quelle organischen Materials auf der Erde besser zu verstehen.

Die Kommunikation des DDA’ mit dem restlichen Satelliten erfolgt mithilfe des SpaceWire-Bussystems und soll durch den in dieser Arbeit entwickelten Datensimulator getestet werden, damit mögliche Fehler in der Flugsoftware bereits frühzeitig erkannt und behoben werden können. Die Notwendigkeit dafür ergibt sich daraus, dass der DDA am Institut für Raumfahrtsysteme der Universität Stuttgart (IRS) zusammen dem Elektronikzulieferer „von Hoerner & Sulger“ realisiert wird, während DESTINY$^+$ eine Mission der Japan Aerospace Exploration Agency (JAXA) ist. Das Deutsches Zentrum für Luft- und Raumfahrt (DLR) ist der deutsche Projekträger. Die geographische Distanz zwischen Experiment und Raumfahrzeug ist zu groß, um bereits in frühen Stadien der Mission direkte Tests durchzuführen.

Der Datensimulator wird als zwei Graphical User Interfaces (GUIs) in C/C++ in der Umgebung Microsoft Visual Studio 2019 auf Microsoft Windows 10 umgesetzt. Eines sendet Daten als Broadcast oder Telecommand, das andere empfängt Telemetrie. Als Schnittstelle zwischen PC und Bus kommt ein SpaceWire Brick Mk3 der Firma STAR-Dundee zum Einsatz, der über zwei Anschlüsse für SpaceWire-Kabel und einen USB-Anschluss verfügt. Somit kann von einem PC der Datenstrom vom und zum DDA gesteuert werden. Die Nachrichten folgen dem Space Packet Protocol des Consultative Committee for Space Data Systems (CCSDS). Zukünftig werden sie jedoch an das Remote Memory Access Protocol (RMAP) angepasst.

Contents

1 Glossary **5**
1.1 Abbreviations 5
1.2 File Extensions 6

2 Introduction **7**
2.1 DESTINY$^+$ Mission Overview 7
2.2 The DESTINY$^+$ Dust Analyser 8
2.3 Data simulator 9
2.4 Thesis Overview and Timeline 9

3 Requirements **10**
3.1 SpaceWire 10
3.2 CCSDS Protocol 12
3.2.1 Packet Header 12
3.2.2 Telemetry 14
3.2.3 Telecommands 16
3.2.4 Broadcast 16
3.3 RMAP 17
3.3.1 Time Slots and Distribution 17
3.3.2 Read Commands 18
3.3.3 Write Commands 20

4 Implementation **22**
4.1 Setup 22
4.1.1 Hardware 22
4.1.2 Software 23
4.2 General 23
4.2.1 Environment Customization 23
4.2.2 Signals and Slots in Qt 23
4.2.3 MainWindow Class 24
4.2.4 Utilities 25
4.2.5 Standalone Executable 25
4.3 Send GUI 26
4.3.1 Read from file 26
4.3.2 Telecommands 27
4.3.3 Broadcast 27
4.4 Receive GUI 28
4.4.1 Telemetry 28
4.5 RMAP Example 29
4.6 Challenges 30
4.6.1 Development Environment 30
4.6.2 Multithreading 30
4.6.3 Broken Hardware 31
4.6.4 Protocol change to RMAP 31

5 Conclusion **32**
5.1 Summary 32
5.2 Evaluation 32
5.3 Outlook 33

6 References **34**
6.1 Articles and Conferences 34
6.2 Books and Guides 34
6.3 Other References 35

A Zusammenfassung (German Summary) **36**
A.1 Motivation 36
A.2 Anforderungen 36
A.3 Umsetzung 37
A.4 Fazit 37

B Files **38**
B.1 Send GUI 38
B.2 Receive GUI 39
B.3 RMAP Example 40

1 Glossary

1.1 Abbreviations

API Application Programming Interface

CCSDS Consultative Committee for Space Data Systems

DDA DESTINY$^+$ Dust Analyser
Please note that although this thesis is oriented toward American English, "DESTINY$^+$ Dust AnalySer" is a name in British English.

DESTINY$^+$ Demonstration and Experiment of Space Technology for INterplanetary voYage with Phaethon fLyby and dUst Science

DLR German Aerospace Center
Deutsches Zentrum für Luft- und Raumfahrt e. V.

ESA European Space Agency

GUI Graphical User Interface

HK Housekeeping Data

IEEE Institute of Electrical and Electronics Engineers

IRS Institute of Space Systems of the University of Stuttgart
Institut für Raumfahrtsysteme der Universität Stuttgart

JAXA Japan Aerospace Exploration Agency

LEO Low Earth Orbit

LVDS Low Voltage Differential Signaling

MDP Mission Data Processor

OS Operating System

P2P Point-to-Point

PC Personal Computer

RMAP Remote Memory Access Protocol

VS Microsoft Visual Studio 2019

1.2 File Extensions

Files with these extensions are submitted (Appendix B) and/or are addressed in the text of this thesis. The short descriptions on the right side are taken from Wikipedia [Wik21].

.c	C Source Code
.cpp	C++ Source Code
.csv	Comma-Separated Values, *to save Telemetry and open in a spreadsheet program*
.exe	Windows Executable
.h	C/C++ Header
.lib	C/C++ Library
.sln	Visual Studio Solution
.txt	Plain Text, *to load Data*
.ui	Qt User Interface
.vcxproj	Visual Studio C++ Project

2 Introduction

2.1 DESTINY$^+$ Mission Overview

"Demonstration and Experiment of Space Technology for INterplanetary voYage with Phaethon fLyby and dUst Science (DESTINY$^+$)" is a planned deep space mission by the Japan Aerospace Exploration Agency (JAXA). The two main goals of the mission are the demonstration of the usability of electrically propelled spacecrafts and the observation of the asteroid 3200 Phaethon. Therefore, the satellite is going to be equipped with four µ10 Ion Thrusters with a combined power of 1670 W and a total thrust of 40 mN. Furthermore, three scientific instruments are going to be installed. For the visual examination the Telescopic Camera for Phaethon (TCAP) and the Multiband Camera for Phaethon (MCAP) will detect light in different spherical angles and wavelengths. The DESTINY$^+$ Dust Analyser (DDA) will collect cosmic dust and determine certain properties of the particles. Additionally, new 20-µm-thick, light-weight Solar Array Paddles are going to generate a maximum of 4.7 kW of electric power [Ara17; Toy17].

DESTINY$^+$ is scheduled to launch into Low Earth Orbit (LEO) with an Epsilon rocket from Uchinora Space Center, Japan in 2024. Afterwards, the spacecraft is going to raise its orbit with the mentioned ion engines. After approximately 1.5 years the probe is going to perform a lunar gravity assist maneuver to reach an interplanetary cruise orbit. In 2028, DESTINY$^+$ is expected to arrive at 3200 Phaethon, where it will fly past at a distance of 500 km. Subsequently, the probe will be able to change its orbit again, which enables it to study another celestial body [Som20]. A sketch of the mission can be seen in Fig. 1.

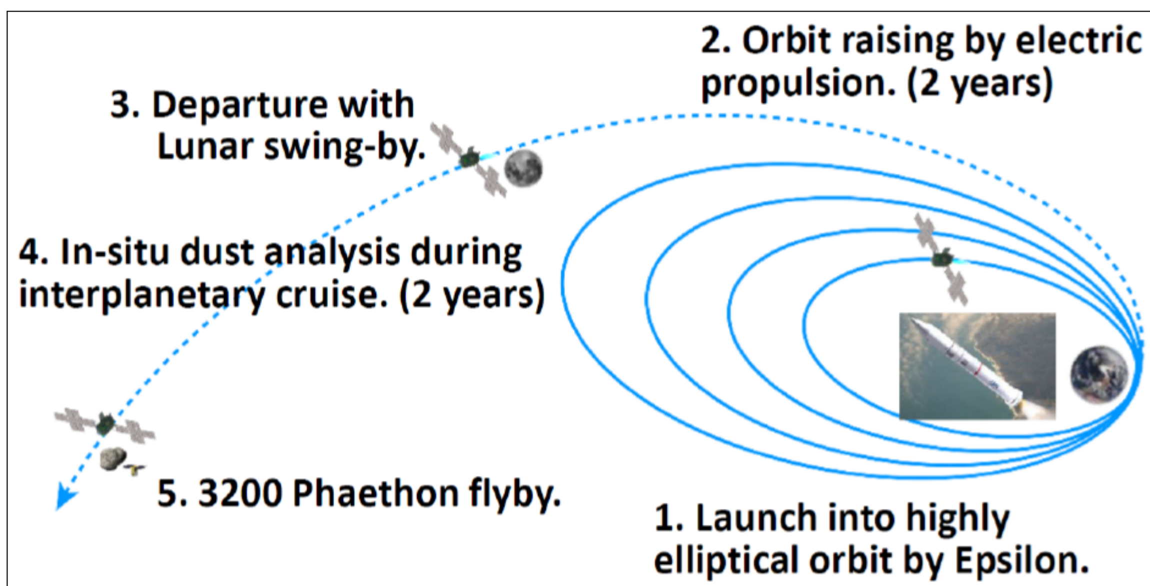


Figure 1: DESTINY$^+$ Mission overview with sketch of the flight path [Toy17]

2.2 The DESTINY+ Dust Analyser

The DESTINY+ Dust Analyser (DDA) is an in-situ dust analyzing instrument for the DESTINY+ mission. It is provided by the German Aerospace Center (DLR) and developed at the Institute of Space Systems of the University of Stuttgart (IRS). It is going to measure both interplanetary and interstellar particles during the cruise phase as well as the matter emitted by 3200 Phaethon during the flyby. The probe is going to quantify mass, speed, arrival direction, charge, flux, and composition of the corpuscles [Mas18].

The shape of the device is going to be a hollow cylinder with electric grids on the inside. In Fig. 2 the basic mechanism is displayed. Dust particles with less than 3.5 keV of charge are rejected. All remaining dust will cross the Trajectory Sensors at (1) and (2). By scaling time and changes in the electric fields, speeds and flight paths can be calculated. The object then crashes on the Target (3) and ionizes. Due to the Acceleration Grid, the ions will then reverse and be reflected by the Reflectron. Again, the time will be clocked to evaluate the mass of the bodies [IRS19]. The DDA is going to collect particles within a 90° cone and the error is going to be less then 10° in direction. Moreover, velocity can be determined for particles reaching speeds of up to $100\,\mathrm{km\,s^{-1}}$ fast with an accuracy of 10%. Mass resolution will be between 10^{-16} g and 10^{-6} g [Mas18].

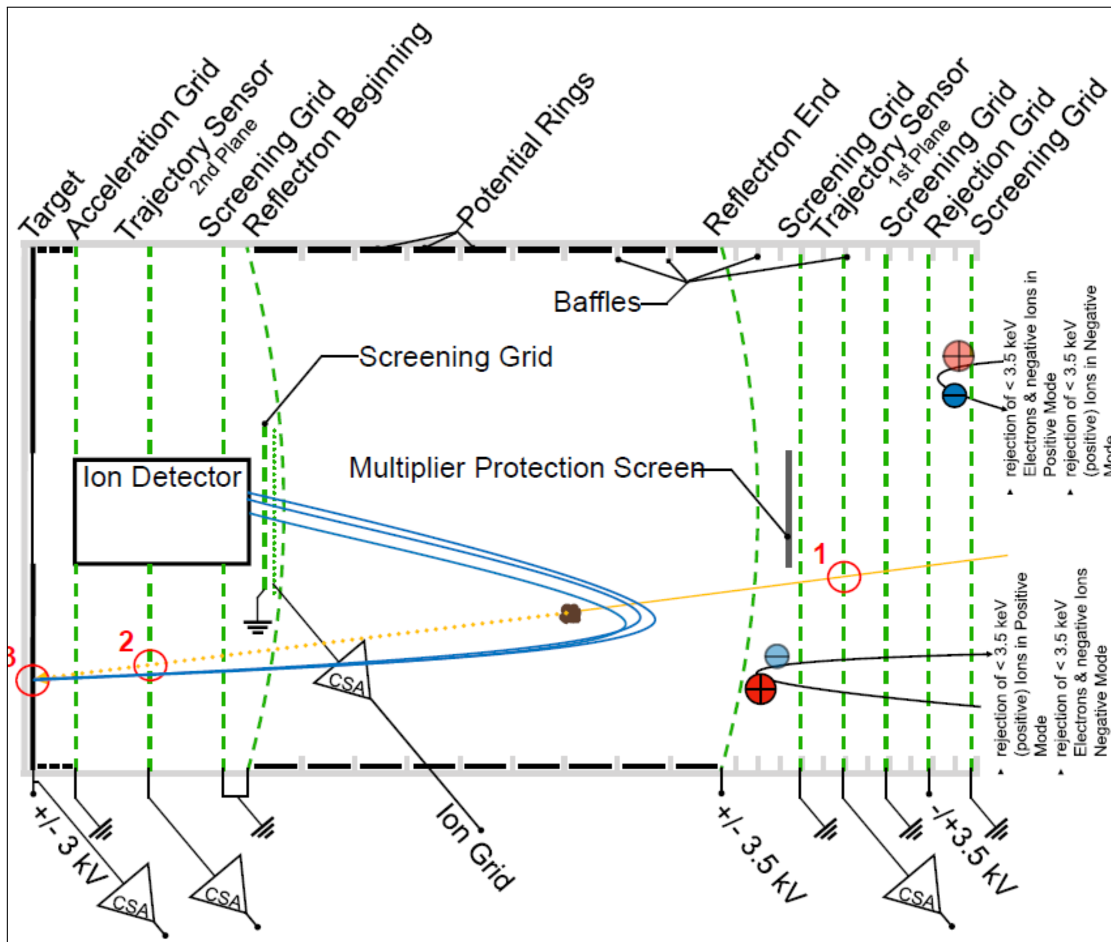


Figure 2: Reaction on a particle entering the DDA [IRS19]

2.3 Data simulator

Testing the DDA without a simulator is difficult, because it is manufactured by the IRS on behalf of the DLR in Stuttgart, Germany, while the rest of the satellite is created in Japan by JAXA. Therefore, the DDA data simulator was developed and this document reports on the creation process.

The simulator emulates the spacecraft's Mission Data Processor (MDP) in terms of its communication behavior. Being built in that way, the link between the DDA and the rest of DESTINY$^+$ can be tested as well as the DDA's reaction to incoming or outgoing messages. In short, the data simulator creates a communication environment similar to that in flight, which helps in the development of the dust experiment.

At first, the simulator was expected to only test the flight software of the DDA. Now, the IRS plans to use it for several tests of the whole dust telescope. Among other things, the DDA will be exposed to particles at the institute's own Stuttgart Dust Accelerator. During these tests the DDA will be controlled through this data simulator. This is supposed to reduce the occurrence of errors in the merging of instrument and spacecraft, which will take place shortly before launch.

2.4 Thesis Overview and Timeline

The work is divided into specifying and elaborating the requirements (Chapter 3) and the implementation of them (Chapter 4). Hence, the thesis is structured into these two major chapters. However, chronologically, the project started with reading the basics about SpaceWire (Chapter 3.1) and the SpaceWire Brick Mk3 (Chapter 4.1.1).

Subsequently, the right hardware and software environment has been chosen (Chapter 4.1) and the necessary skills for developing software and Graphical User Interfaces (GUIs) in C++ have been learned (Chapter 4.2). The functions of the Brick's Application Programming Interface (API) were tested with the basic applications provided by the manufacturer STAR-Dundee. Thereafter, the example code of the company was extended with a simple GUI, which can transmit and receive SpaceWire packets.

Then, the requirements of the simulator were evaluated (Chapter 3), so that the parameters could be adapted onto the GUI. The coding was continued and the parsing according to protocol of the Consultative Committee for Space Data Systems (CCSDS) was started. The work had to pause because the definition was incomplete and then changed to the Remote Memory Access Protocol (RMAP). Plus, a defect of the Brick Mk3 was discovered (Chapter 4.6). In this recess, the writing of this thesis started and the implementation was continued with assumptions.

After programming was finished, the code was structured and commented. The thesis' text was completed and proofread. As soon as JAXA and the IRS expanded and changed their specifications, an example code of RMAP was added (Chapter 4.5). Due to the short remaining time, a complete conversion could not be realized.

3 Requirements

The DDA communicates with the spacecraft via a single SpaceWire link (Chapter 3.1). The data simulator shall be able to emulate the MDP's behavior, i.e. sending and reading packets to and from a Personal Computer via a GUI. Furthermore, the packets should be led by a header. There are three types of messages to be processed: Telemetry is transmitted by the DDA, whereas telecommands and broadcast are transmitted by the simulator (Table 1). Hence, the GUI must have buttons to start and stop sending and receiving.

This work was commenced under the hypothesis that the communication would follow the protocol of CCSDS because the European Space Agency (ESA) recommends it for sending Telemetry and Telecommands (Chapter 3.2). The other common protocol is RMAP, which aims "to support reading from and writing to memory in a remote SpaceWire node" [ECS10]. Shortly before submission of the thesis the desired protocol was set to RMAP (Chapter 3.3).

	Simulator	**DDA**
Telemetry	receiving	sending
Telecommands	sending	receiving
Broadcast	sending	receiving

Table 1: The three message types used by the DDA

3.1 SpaceWire

SpaceWire is a bus system for spacecrafts administered by ESA. Other space agencies use the standard as well, among them the US-American NASA, the Russian RKA, and JAXA. SpaceWire was published in 2003 and provides high-speed data exchange of up to 200 Mbits/s. The bus can be used Point-to-Point (P2P) or with routers creating a network. SpaceWire is based on the IEEE-1355 standard, which has been modified for space usage. It works with full-duplex and bidirectional P2P-Links between two nodes or a node and a router. A node is equipment "using the services of a SpaceWire link or network" [Par12, p. 77]. The system ensures compatibility between all nodes, even though they might be from different manufacturers.

The DDA uses a P2P link. If interested in networks, routing, and addressing, please cf. Parkes [Par12]. SpaceWire works with Low Voltage Differential Signaling (LVDS). LVDS sends every signal via two strands, with voltage of the same absolute value but opposite signs, switching between 0 and 1. As illustrated in Fig. 3, a voltage between Vin+ -250 and -400 mV means logical 0 and $+250$ to $+400$ mV signals logical 1. For Vin- it is the other way around. Upon receiving, the signals are subtracted from each other. The noise will then cancel itself out. Other advantages are: Nearly "constant current which decreases switching noise", up to ± 1 V tolerance in ground difference, low electromagnetic emissions because of LVDS, fail-safe operation, and low power consumption [Par12, p. 51].

Two signals (data and strobe) are sent in each direction differentially, which equals a total of eight strands per cable. The connectors have an additional ninth pin for the inner shield. The structure of both cable and connector are shown in Fig. 4 and 5.

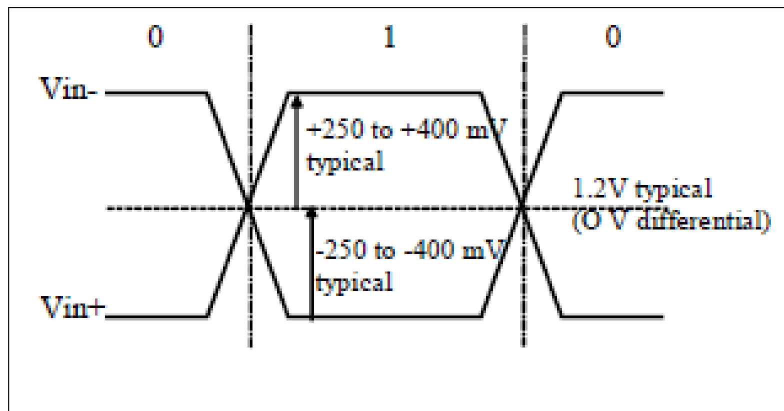


Figure 3: Low Voltage Differential Signaling [Par12, p. 50]

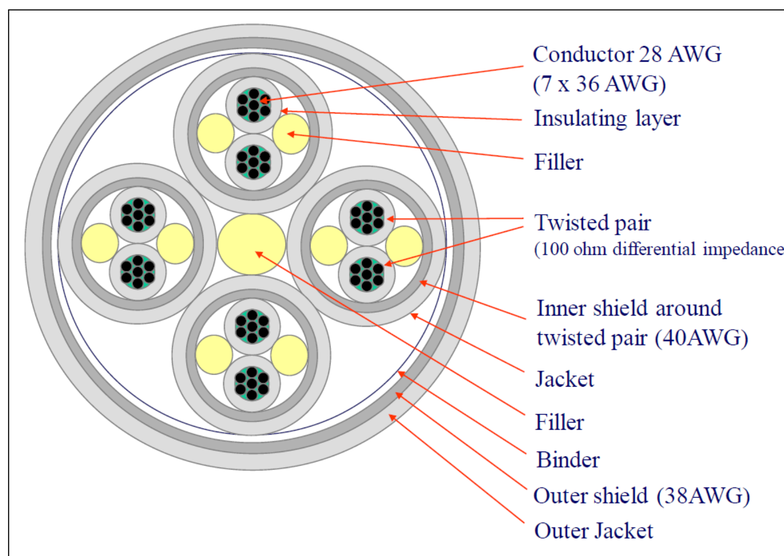


Figure 4: SpaceWire cable structure [Par12, p. 46]

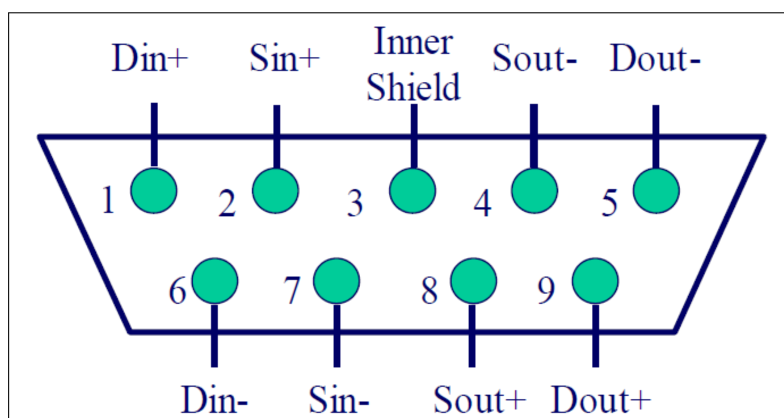


Figure 5: SpaceWire connector pin-out [Par12, p. 47]

The bus system can be used very freely because raw bytes terminated by an End of Packet (EOP) (if no error occurred) or Error End of Packet (EEP) token (if an error occurred) are sent. Therefore, additional standards are defined to increase compatibility. The most commonly used are both the protocol of the CCSDS (Chapter 3.2) and RMAP by ESA (Chapter 3.3).

3.2 CCSDS Protocol

The DDA had been expected to follow the CCSDS recommended standards of packet protocol [CCS20] and time codes [CCS10] before the change to RMAP. The CCSDS however is a multinational organization with 11 member agencies including the DDA and DESTINY+ project sponsors DLR and JAXA, as well as the space agencies of Europe, Russia, China, and the United States [CCSMA]. It issues recommended standards and practices to standardize space data systems and was founded in 1982.

3.2.1 Packet Header

Telemetry and Telecommand packets use the same header protocol. It includes basic information about the delivered message, such as length and time. Additionally, it includes flags and a counter, that show the order of packages sent in a sequence (Table 2). Sequences will occur if a message is larger than the capacity of a single packet. This will be the case when transmitting telemetry messages.

An explanation of the time code in the Secondary Header should be given here (Table 2). Originally, the word "Epoch" has been attributed to the Unix time, which counts the seconds since January 1, 1970, 00:00:00 UTC [IEE17]. The advantage is that only one integer is needed compared to the display of years, months, days, hours, minutes, and seconds. The CCSDS's counter has started on January 1, 1958 [CCS10, p. 3-2]. The basic time unit (i.e. seconds) is saved into 32 bits, which allow for approximately 136 years of time, until the definition would suffer an overflow. This would be in 2094, long after the mission's end.

Moreover, one byte is set aside for passing fractional time, with the syntax shown in Table 3. The conversion from milliseconds to the fractional time unit can be implemented as follows in Source Code 1. Because the fractional time is conveyed with eight bits, the decimals are lost after the division. Therefore, the resolution is $\frac{1}{256}\,\mathrm{s} \approx 3.9\,\mathrm{ms}$.

```c
unsigned char fractional_time = msec_time * 256 / 1000;
                    /* 165 [FTU] = 645 [msec]* 256 / 1000 */
```

Source Code 1: Conversion from msec to fractional time unit incl. an example

DDA Packet Header in accordance with CCSDS				
Header	**Name**		**Bits**	**Binary Value**
Primary Header	Packet Version Number		3	000
	Packet Identification	Packet Type	1	0 = Telemetry 1 = Telecommand
		Sec. Header Flag	1	1 = Sec. Header available
		Application Process Indentifier	11	
	Packet Sequence Control	Sequence Flags	2	00 = Continuation of sequence 01 = First packet of sequence 10 = Last packet of sequence 11 = Unsegmented data
		Packet Sequence Count or Packet Name	14	var, increased with every packet
	Packet Data Length		16	(Total number of octets in the Packet Data Field) - 1
Secondary Header	P-Field	P-Field Extension	1	0 = No extension
		Time Code Identification	3	001 = Epoch January 1, 1958
		(Number of octets of the basic time unit) - 1	2	11 = 4 octets
		Number of octets of the fractional time unit	2	01 = 1 octet
	T-Field	Basic time unit	32	var = Number of seconds past the epoch
		Fractional time unit	8	var = msec fraction byte

Table 2: DDA Packet Header in accordance with CCSDS [CCS10; CCS20]

Bit	1	2	3	4	5	6	7	8
Value [s]	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
Fraction [s]	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$
Expanded [s]	$\frac{128}{256}$	$\frac{64}{256}$	$\frac{32}{256}$	$\frac{16}{256}$	$\frac{8}{256}$	$\frac{4}{256}$	$\frac{2}{256}$	$\frac{1}{256}$
Example	1	0	1	0	0	1	0	1
Value [s]	2^{-1}	0	2^{-3}	0	0	2^{-6}	0	2^{-8}
Fraction [s]	$\frac{1}{2}$	0	$\frac{1}{8}$	0	0	$\frac{1}{64}$	0	$\frac{1}{256}$
Expanded [s]	$\frac{128}{256}$	0	$\frac{32}{256}$	0	0	$\frac{4}{256}$	0	$\frac{1}{256}$
Σ over fractions	$= \frac{165}{256}\,\mathrm{s} \approx 645\,\mathrm{ms}$							

Table 3: Definition and Example (10100101) of the fractional time unit

3.2.2 Telemetry

Telemetry is data generated by the DDA, e.g. the parameters of an impact of a dust particle. The experiment's preprocessor compresses the information, splits it into multiple SpaceWire packets and provides them with a header. Afterwards, the packets are sent to the MDP because the satellite's downlink system is located outside the DDA [Toy17]. Telemetry can be received on earth and evaluated by the IRS.

As mentioned above, telemetry is data sent by the DDA, thus received by the simulator. Therefore, the GUI must be able to receive telemetry packets, parse the message according to protocol, and save it into a *.csv* file. This practice guarantees that the information can be accessed and edited easily in a spreadsheet application. It is necessary to enable sequencing because on the one hand the amount of data generated by one dust particle impact is about 400 kbit and on the other hand an assumable DDA Packet transports a maximum of 1 kByte = 8192 bit of data. This leads to a sequence of approximately 49 packets per measurement.

The packets themselves consist of a Primary and a Secondary Header (Table 2), plus the Packet Data Field. Table 4 shows the definition of the DDA Packet Data Field. The content of the Coded Data Sets (Table 4, p. 16) has not yet been defined because the instruments specification have not been finished, so their size cannot be determined.

Telemetry Packet Data Field according to the IRS				
Field	**Name**		**Bits**	**Binary Value**
Source Data Field	Grouping Data Length Field		16	0000 + var = first four bits reserved + number of packets within the group minus one (max. 4096)
	Compression Technique Identification Field		8	var = Compression technique all zero = No compression
	Reference Sample Interval Field (r)		8	var = Number of data sets counted from one data set containing a reference sample up to but not including the next consecutive data set containing a reference sample
	Preprocessor Subfield	Header	2	00 = Preprocessor
		Preprocessor Status	1	0 = Absent 1 = Present
		Predictor Type	3	000 = Bypass predictor 001 = Unit delay predictor 111 = Application-specific predictor
		Mapper Type	2	00 = Prediction error mapper 11 = Application-specific mapper

Continuation **Telemetry Packet Data Field according to the IRS**				
Field	**Name**		**Bits**	**Binary Value**
		Block Size (J)	2	Number of Samples per Block 00 – J=8 01 – J=16 10 – J=32 or J=64 11 – Application-specific
		Data Sense	1	0 = Two's complement 1 = Positive; mandatory if preprocessor is bypassed
		Input data sample resolution (n)	5	var = Input data sample resolution minus one (max. 32)
	Entropy Coder Subfield	Header	2	01 = Entropy Coder
		Data Resolution Range	2	00 = Spare 01 for $n \leq 8$ 10 for $8 < n \leq 16$ 11 for $16 < n \leq 32$
		Number of CDS per packet (m)	12	var = Number of CDSs per packet minus one
	Extended Parameter Subfield	Header	2	11 = Extended Parameters
		Reserved	2	00
		Block Size	4	Number of Samples per Block 0000 – J=8 0001 – J=16 0010 – J=32 0011 – J=64 1111 – Application Specific
		Reserved	1	0
		Restricted Code Options Flag	1	0 = Basic set of code options 1 = Restricted set of code options
		Reserved	2	00
		Reference Sample Interval Extension	4	var = (r-1)/256 as an integer, i.e. the largest integer less than or equal to (r–1)/256 shall be encoded
	Instrument Configuration Subfield	Header	2	10 = Instrument configuration
		Event Number	16	var = Consecutive number of the current event
		To be determined (TBD) by the IRS	*TBD*	Unique instrument configuration parameters

Continuation **Telemetry Packet Data Field according to the IRS**			
Field	**Name**	**Bits**	**Binary Value**
Coded Data Sets	CDS 1	*TBD*	Lossless compressed, lossy compressed or raw coded data sets that contain a science or housekeeping data frame
	CDS 2	*TBD*	
	CDS n	*TBD*	
	Fill bits	var	Fill bits if necessary, as the packet length is fixed

Table 4: Definition of the Source Data Field of the DDA Telemetry packet

3.2.3 Telecommands

Telecommands are packets which present information to the DDA, e.g. starting or stopping the experiment, turning the sensor, etc. They are led by the same header as telemetry, specified in Chapter 3.2.1. The actual content has not been defined yet. Therefore, the GUI shall have the possibility to enter the bytes sent after the header manually. This is realized either through a free text field or a *.txt*-file.

3.2.4 Broadcast

Broadcast is data distributed by the spacecraft to all science instruments via a single SpaceWire packet each. The different assumable broadcast types are currently:

- Periodically every n seconds
  - Spacecraft time to synchronize DDA's internal time
  - Spacecraft's attitude (accuracy $\pm 0.1^{\circ}$)
    - Quaternion
    - Three angles to the sun
- On mode change
  - Safe mode
  - TLM mode
  - OP mode
  - Articulation allowed/forbidden
  - Low power mode
  - Ion engine on/off
  - Thruster firing is/was active

Just as telecommands, broadcast has not been fixed yet. Hence, the GUI should also have a free text field and a *.txt*-file, where the user can enter the contents of the broadcast. Additionally, the repetition period can be determined.

3.3 RMAP

RMAP is a protocol designed by ESA to configure SpaceWire networks and control units, as well as gather data and status information from the applications [ECS10]. JAXA changed the packet protocol to RMAP in the later course of this thesis.

The two modes used by DESTINY$^+$ are the “Read Command” for all data transferred from DDA to spacecraft (Chapter 3.3.2) and the “Write Command” for information transmitted from satellite to DDA (Chapter 3.3.3). The third mode “Read-Modify-Write” is not used. In addition, DESTINY$^+$ defines “Time Slots” and “Time Distribution” (Chapter 3.3.1).

The following Chapters 3.3.3 and 3.3.2 aim to simplify and merge the information of the RMAP standard [ECS10] and the definitions of the DESTINY$^+$ Project Team [Des21] and give an overview of the relevant specifications for the data simulator. For further information please consider those documents. The values assigned to the different header bytes are listed in Table 5.

RMAP Header Values 1 byte per field (exceptions are marked)	
Name	**Value**
Target Logical Address/ Initiator Logical Address	0x80 = MDP (Spacecraft) 0xB0 = DDA
Protocol Identifier	0x01 = RMAP
Instruction	Cf. Fig. 6,7,8,9
Key	0x00
Status	0x00 = Command executed successfully 0x01 - 0x0C = Various error code
Transaction Identifier	01b = Mission data 00b = Other content Rest of bits = Irrelevant
Adress Field (5 bytes)	TBD
Data Length (3 bytes)	var = Number of octets in the Data Field
CRC	var = Check value

Table 5: Values in RMAP Header [Des21; ECS10]

3.3.1 Time Slots and Distribution

“Fine Time”, i.e. fractional time, is distributed by Time-Code, while “Coarse Time”, i.e. whole seconds, are sent via a Write Command (Chapter 3.3.3). The DDA can recreate the master time from those to broadcasts.

Fine Time Time-Code is an unconventional SpaceWire packet, which starts with an Escape Token (ESC) and conveys six bits of data. Therefore, the time resolution is $\frac{1}{2^6}\,\mathrm{s} = \frac{1}{64}\,\mathrm{s} = 15.625\,\mathrm{ms}$.

Coarse Time Coarse Time is distributed by a single Write Command with four octets of data without Reply. The integer resembles a time in seconds after a certain Epoch which is not defined yet. The CCSDS however recommends January 1, 1958 [CCS10, p. 3-2].

Time Slots JAXA defines slots to limit delay time and prevent blockages. Time-Codes divisible by four limit one Mission Time Slot each. As a result, every data transfer must be completed after $4 \cdot 15.625\,\text{ms} = 62.5\,\text{ms}$, otherwise the transmission is aborted and an Error End of Packet (EEP) token is sent.

3.3.2 Read Commands

Read Commands can be used to request Housekeeping Data (HK) or Mission Data, i.e. they replace the telemetry packets in Chapter 3.2.2. The spacecraft asks for data through the Command and the Reply of the DDA carries the data. The syntax of the Read Command is shown in Fig. 6 and of the Read Reply in Fig. 7. Because DESTINY$^+$ works with logical addressing, the Target SpaceWire Address and the Reply Address are not part of the packets. Thus, Read Commands are 16 and the Read Reply Header is 12 bytes long.

HK Collection and Reply HK is collected in raw data. A total of 144 bytes is gathered per packet. The first 16 bytes are called "Essential HK" and are prioritized at downlink. What type of and how much HK data is generated by the DDA, has yet to be determined.

Mission Data Collection and Reply Mission Data is collected by Read Reply as well. These packages contain data, which will be defined by the IRS. Additionally, the data field is led by a CCSDS primary and secondary header, similar to Table 2. Despite that, the assigned values differ (e.g. the Time Field does not constitute the transmission time but the time of the dust impact) and the secondary header has additional fields.

The maximum amount of user data transmittable in one packet is 2032 bytes (= 2048 − 15 (CCSDS header) − 1 (reserved)), whereas the data is split into messages of 1002 bytes for downlink on the spacecraft. Therefore, it is recommended that one SpaceWire packet does not carry more than 1002 bytes of data plus headers.

Memory Dump (optional) It is possible to read the DDA's memory directly through Memory Dump. A basic Read Command asks for the data at a certain address. The Reply provides up to 512 bytes of data. If the amount of data exceeds that, the MPD will send more than one Read Command.

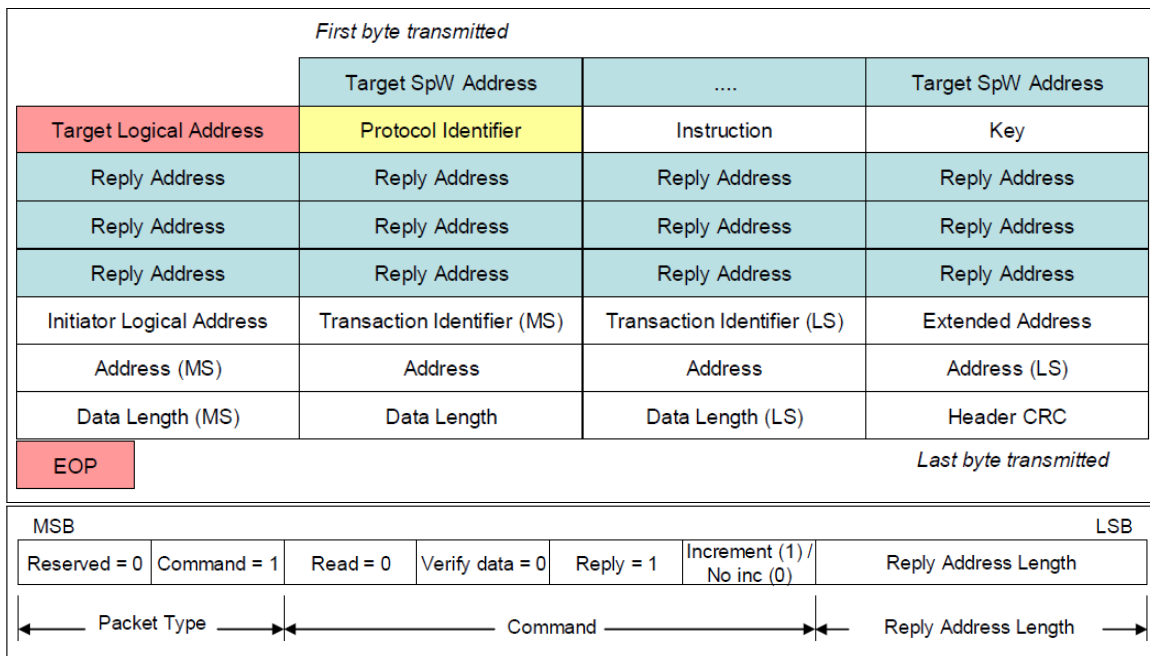


Figure 6: RMAP Read Command Protocol (top) and Instruction Field (bottom) [ECS10, p. 40]

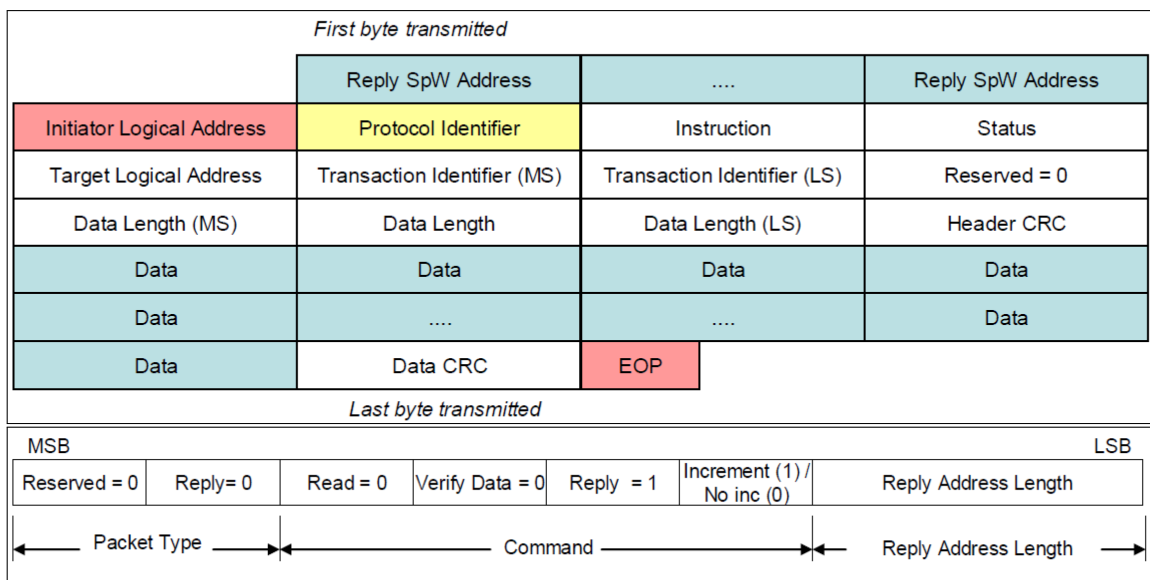


Figure 7: RMAP Read Reply Protocol (top) and Instruction Field (bottom) [ECS10, p. 42]

3.3.3 Write Commands

As mentioned above, Write Commands are used for packets carrying information for the DDA, therefore replacing the assumed telecommand and broadcast in the CCSDS protocol. The syntax of an RMAP Write Command is shown in Fig. 8. Because DESTINY$^+$ works with logical addressing, the Target SpaceWire Address and the Reply Address are not part of the packets. Thus, the length of the header is 16 bytes.

There are optional, short Replies which confirm the reception of the message. Their headers are similar to the Command's, with initiator and target logical address exchanged. Replies use logical addressing too, hence they are eight bytes long (Fig. 9). JAXA split the Write Commands into different packages as follows.

Coarse Time Distribution Coarse Time will be sent by a Write Command without Reply. For more information please confer Chapter 3.3.1.

Command Distribution and Reply The Command Distribution contains a Telecommand as data. It asks for a Reply. The data itself (max. 512 bytes) starts with a CCSDS primary header (Table 2) but has no secondary header. The purpose of commands can be found in Chapter 3.2.3.

Data Distribution Data Distribution replaces the makeshift broadcast (Chapter 3.2.4). It does not require Replies. Similar to Command Distribution, the RMAP header is followed by an CCSDS packet but also including a secondary header. Its contents have not yet been defined completely.

Memory Load (optional) The memory load is a possibility to write directly onto the dust telescope's memory. This enables updates and bug-fixes. It is realized through a simple Write Command, without Reply. If the amount of update data is bigger than 512 bytes, the information is split into more than one package.

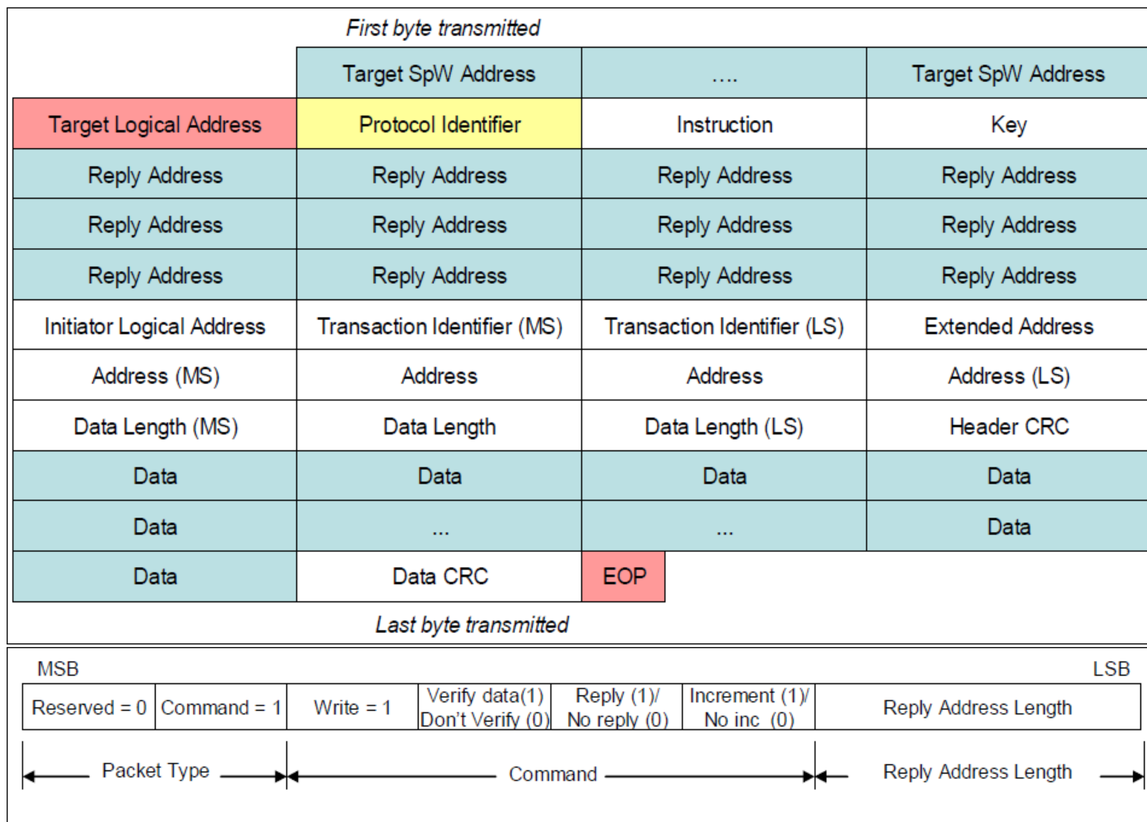


Figure 8: RMAP Write Command Protocol (top) and Instruction Field (bottom) [ECS10, p. 24]

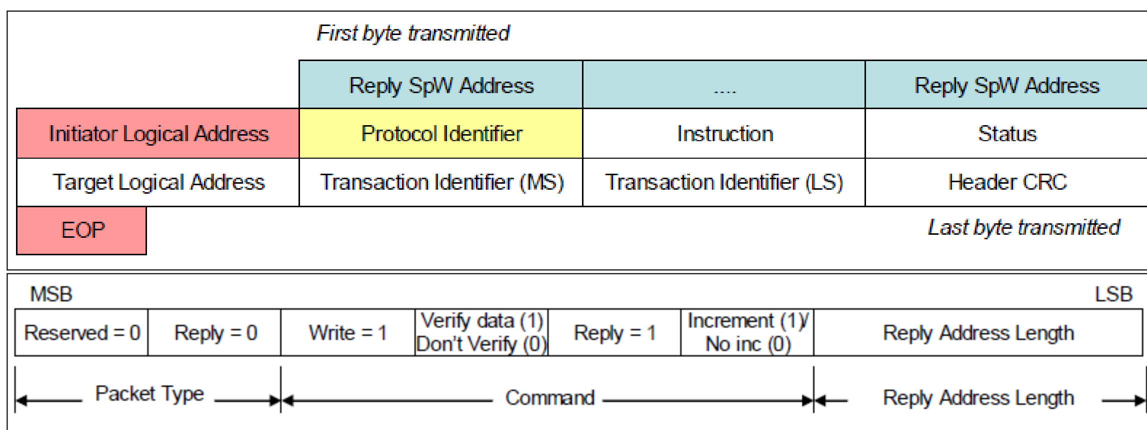


Figure 9: RMAP Write Reply Protocol (top) and Instruction Field (bottom) [ECS10, p. 27]

4 Implementation

The implementation is the main subject of this thesis. The setup is specified in Chapter 4.1. Splitting the tasks between sending and receiving and creating two GUIs, was considered useful in the process. Thus, the report on developing is split into general files and tasks that are used similarly in both Send and Receive GUI (Chapter 4.2) and code that was developed for sending (Chapter 4.3) or receiving (Chapter 4.4). The documentation of the RMAP example (Chapter 4.5) and the challenges are enclosed as well (Chapter 4.6).

4.1 Setup

4.1.1 Hardware

The company "STAR-Dundee Ltd." from Dundee, Scotland provides USB to SpaceWire conversion hardware. Their "SpaceWire Brick Mk3" [StaMk] is equipped with two SpaceWire ports. Therefore, a loop can be created to test the simulator before attaching it to the DDA. Because of a hardware defect, the company sent a "Brick Mk4" as a replacement, which is the backward compatible successor. The Brick is controlled on a Personal Computer (PC) via USB. The hardware setup is illustrated in Fig. 10.

Figure 10: Hardware setup; top: USB cable to PC; center: Brick Mk4; bottom: SpaceWire loop

4.1.2 Software

The Operating System (OS) installed on the PC used is Windows 10 [MicWi]. Coding has been taking place in Microsoft Visual Studio 2019 (VS), an integrated development environment by Microsoft [MicVS]. The VS extension "Qt Visual Studio Tools" enables the creation of GUIs with the Qt toolkit (Version 6) by The Qt Company [Qt21].

The data simulator is based on the "STAR-System" API of STAR-Dundee's Bricks, which uses C/C++ [StaSy]. The manufacturer provides a documentation, function libraries, and example code for all customers. The functions of the API access the properties of the brick, e.g. open and close ports for sending/receiving, create and destroy packages, etc.

The developed functions are based on the examples *advanced_send_example.c* and *advanced_receive_example.c*, which use the API. The programs could be tested step by step because of the "Transmit"/"Receive" applications included in the STAR-System: Status and content of the sent messages could be seen in the "Receive" application. Packets obtained from the "Transmit" application inspect the other direction.

4.2 General

4.2.1 Environment Customization

Before the examples and the data simulator are compilable, certain adjustments to VS have to be made. As mentioned above, the Qt extension has to be installed and a Qt *‹MainWindow›* project has to be created. In addition, both the examples and the data simulator use STAR-Dundee's libraries and header files. Therefore, their folders are pasted into the project. The directory *star* contains all *.h* files, while *lib* consists of the *.lib* files for 32-bit and 64-bit OSs.

To access this data, the project settings have to be adapted. The compiler needs an additional include path to the header directory and the linker to the libraries. For this, the project must be opened and right clicked on. Then to *"Properties → C/C++ → General → Additional Include Directories"* has to be navigated. In this field the header files can be added with *$SolutionDir$star*. The libraries are included similarly under *"Properties → Linker → General → Additional Library Directories"*. The commands are *$SolutionDir$lib\x86-64* and *$SolutionDir$lib\x86-32*.

If linking errors still occur, the language has to be set to "ISO C++ 17" under *"Properties → General → C++ Language Standard"* and/or include the untraceable *.lib* files directly under *"Properties → Linker → Input → Additional Dependencies"*.

4.2.2 Signals and Slots in Qt

Signals and slots are the solution to communicate between function classes in Qt. The derived class cannot access functions or variables from the parent. Therefore, when wanting to release information or to call a function, a signal in the derived class and a slot for it in the parent class are defined. An example can be seen in Source Code 2.

```
1  /* derived.h */
2  class Derived /* ... */
3  {
4  signals:
5    void errorExample(void);
6  }
7
8
9  /* derived.cpp */
10 void Derived::a_function(void)
11 {
12   /* ... */
13   emit errorExample();
14   /* ... */
15 }
16
17
18 /* mainwindow.h */
19 class MainWindow /* ... */
20 {
21 private slots:
22   void on_errorExample(void);
23 /* ... */
24 }
25
26
27 /* mainwindow.cpp */
28 MainWindow::MainWindow(QWidget* parent)
29 : QMainWindow(parent)
30 {
31   /* Opening the GUI */
32   ui.setupUi(this);
33
34   /* Declare "this" as parent of "Derived" */
35   derived_item = new Derived(this);
36
37   /* Connect the error signals */
38   connect(derived_item, SIGNAL(errorExample(void)), this, SLOT(↩
       on_errorExample(void)));
39 }
40
41 void MainWindow::on_errorExample(void)
42 {
43   /* Action performed when emitting errorExample in derived.cpp */
44 }
```

Source Code 2: Example of signals and slots in Qt with MainWindow as parent

4.2.3 MainWindow Class

The source code needed for the GUI is defined via Qt. There are several possibilities from which *QMainWindow* was selected as the Qt class. It provides all functions needed to display the GUI. The four files *mainwindow.h*, *mainwindow.cpp*, *main.cpp*, and *mainwindow.ui* are created automatically when opening a Qt project in VS.

mainwindow.h Following the C/C++ manner, the header file contains the declarations of all functions, global variables, and the GUI itself. Moreover, it encloses the signals and slots, which allow Qt to communicate between classes (cf. Chapter 4.2.2).

mainwindow.cpp *mainwindow.cpp* defines all declarations of *mainwindow.h*, i.e. setting up and destroying the GUI as well as connecting signals and slots (cf. Chapter 4.2.2). The consequences of clicking buttons on the window will be defined in functions, with the syntax of *void on_<buttonName>_clicked(void)*. Besides, all changes on the GUI are set here, e.g. when opening a channel, the "Close" button will be enabled, while the "Open" button will be disabled. The slots, i.e. errors emitted from another class, are defined as well.

main.cpp The file consists of the main function, which only has four lines of code. They declare the *QApplication* and the *MainWindow*. In other words, they initialize and show the application, while the execution happens in the background and in the *mainwindow.cpp* file.

mainwindow.ui *mainwindow.ui* is the file which defines the graphical appearance of the application towards the user. In VS it can be edited graphically, VS translates it into code.

4.2.4 Utilities

There are additional utilities provided by StarDundee. Not all of the functions are used but they are all included, in case the IRS wants to make adjustments to this work.

utilities.h The utility functions are declared here.

utilities.cpp The utilities are defined here. They include functions which create device and channel lists, ask the user to select a device or channel on the console, and write the input string into bytes. Furthermore, the packet contents and version information can be printed.

4.2.5 Standalone Executable

Running the data simulator without opening VS for it is not possible without further measures. When trying to open the executable, error messages are displayed. Certain Qt Dynamic Link Libraries (*.dll*) are not found. Fortunately, the Qt distribution has a tool called *windeployqt.exe* which can create a standalone version of Qt project *.exe* files. Qt provides a documentation for it [QtWin].

The easiest possible solution is opening the Windows command prompt and navigating to *windeployqt.exe*'s path which is similar to *C:\Qt\6.0.3\msvc2019_64\bin* while version number and hard disk drive designation may differ. Afterwards, *"windeployqt --release $The path to your GUI .exe file$"* is entered. Now a standalone should be created. When wanting to start it from another directory, consider using Windows Shortcuts because changing the path afterwards might result in errors.

4.3 Send GUI

The send application consists of three columns. On the left, a short manual and possible errors are displayed. Opening and closing a channel can be performed too. In the middle, telecommands can be sent. The user can select whether a header is needed and whether they want to enter the message by hand or read it from a *.txt* file. The message can be entered and transmitted below. On the right, broadcast can be started and stopped. The layout is basically the same as in the middle, except for the time interval which can be set for broadcast as well. The final version of the send GUI is shown in Fig. 11.

mainwindow.h is led by two macros which let the IRS change the length of the header (`COMM_HEADER_LEN`) and the message itself (`MAX_LEN`) easily. Note that if the length of the header is changed, the assignment of the header bytes in *send_telecommand.cpp* must be adjusted as well. Otherwise, zeros are added when extended or information is lost when shortened. The following subsections list all source files and their functions, which complete the ones specified in Chapter 4.2.

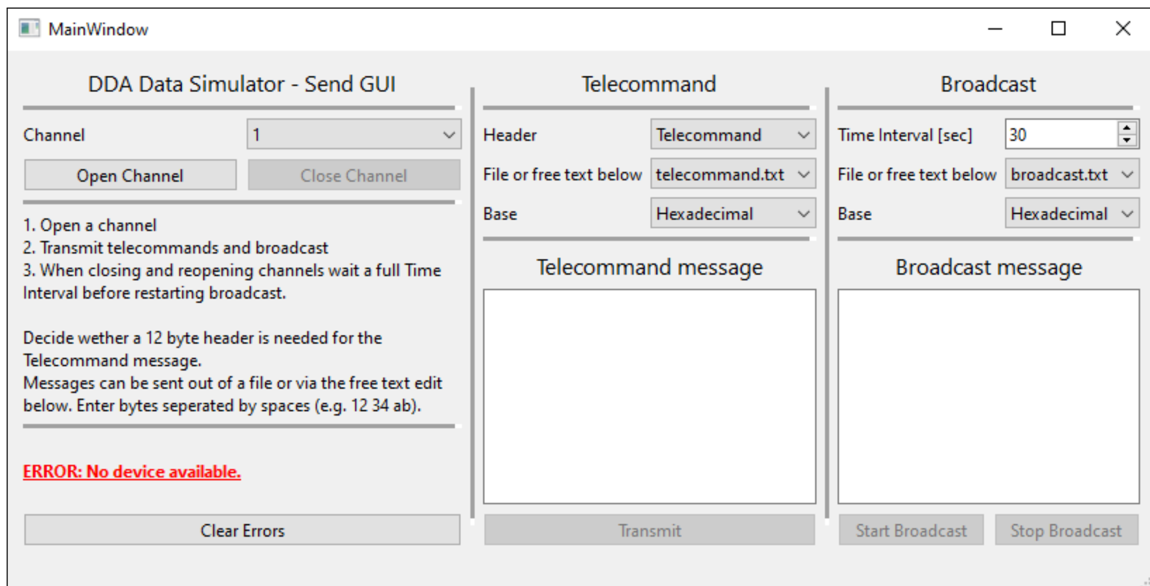


Figure 11: Send GUI of the data simulator with error displayed

4.3.1 Read from file

readFromFile.h In this header the global *readFromFile()* function is declared, so that it is accessible from both *Broadcast_Thread* and *MainWindow*.

readFromFile.cpp The *readFromFile()* function opens a *.txt* file and reads the first line in it. It is called by the send/broadcast functions when needed.

4.3.2 Telecommands

Sending telecommands is realized in the *MainWindow* class. When the "Transmit" button is pressed, one of the following functions are called.

send_telecommand.cpp This file contains the *MainWindow::send_telecommand()* function, which sends messages with a telecommand header. The time is read from the OS. The message input is converted into a C-array, considering the hexadecimal or binary base because the API works with C. For the actual sending process, a packet and a transfer operation are created and destroyed after completion.

send_vanilla.cpp The *MainWindow::send_vanilla()* function is similar to *MainWindow::send_telecommand()* but skips the creation of a header. However, the code differs slightly in several places. Because the clarity would suffer, a common function has been waived.

4.3.3 Broadcast

Broadcast is implemented as an additional thread, so that it can run independently in the background. Otherwise, no telemetry could be sent and the GUI would freeze while broadcasting. Thus, a new class must be created with the properties of ‹*QThread*›. When the "Start Broadcast" button is pressed, all important variables are initialized and the thread is started.

broadcast_thread.h The header file declares the class *Broadcast_Thread*, its variables, and the error signals to the *MainWindow* class.

broadcast_thread.cpp This file contains the function *Broadcast_Thread::run()*, which is a function of the ‹*QThread*› library. This part of code executes the actual broadcast as long as *stop* is false. The boolean is used as a termination criterion for a while loop. Therefore, stopping the broadcast can take up to one full "Time Interval".

4.4 Receive GUI

The receive GUI is significantly easier, cf. Fig. 12. The user can only start and stop receiving. Received messages are saved into a file after distinguishing if the telemetry header is present or not. The following subsection lists all source files and their functions, which complete the ones specified in Chapter 4.2. In *mainwindow.h*, there are three macros, where the two filenames (`FILE_TELEMETRY` and `FILE_NO_HEADER`) and the wait time (`WAIT_TIME`) can be edited without searching the right place in the code.

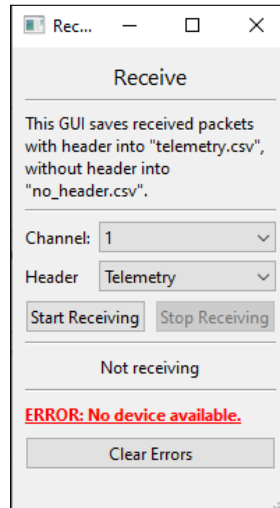


Figure 12: Receive GUI of the data simulator with error displayed.

4.4.1 Telemetry

Similar to broadcast, the Receive GUI is controlled by two threads. This is necessary because otherwise the GUI would freeze while waiting for messages. The derived class has been called *Receive_Thread* and consists of the following three files.

receive_thread.h This header contains the class definition and all declarations, i.e. variables, functions, and signals.

receive_thread.cpp This file contains the receiving task, performed by the *Receive_Thread::run()* function. The thread waits for messages, but for a maximum of `WAIT_TIME`. The default value of the macro is 50 ms. Then, a while loop will check if the “Stop Receiving” button has been clicked. No messages will be lost because the Bricks Mk3 and Mk4 have buffers. If a message is received, the function will stop waiting and the content will be written into a file by the *writeToFile()* function.

writeToFile.cpp The *Receive_Thread::writeToFile()* function performs several tasks with the aim of preparing the data for further use in a spreadsheet program. At first, it converts the received data into a string of binary numbers and sets a semicolon between bytes. Then, depending on the user's input and the validity of the first byte, the message is either parsed into the different header bits and saved into `FILE_TELEMETRY` or unparsed into `FILE_NO_HEADER`. The default file *telemetry.csv* has a table header. Opening it in a spreadsheet program shows the different header sections as columns including their labels (Fig. 13).

	Q	R	S	T
1	_14	_15	_16	User Data Field
2	Basic time unit 3/4	Basic time unit 4/4	Fractional time unit	Grouping Data Length Field 1/2
3	11010110	11101100	00111000	00000000
4	11010111	00001010	00100000	00000000
5	11010111	00001010	00111010	00000000
6	11010111	00001010	01010011	00000000
7	11010111	00001010	01101110	00000000
8	11010111	00001010	10001000	00000000

Figure 13: *telemetry.csv* opened in Microsoft Excel, incl. labels and example headers.

4.5 RMAP Example

As the deadline for the thesis approached, the switch to RMAP had not been known for long. Hence, a complete reconstruction of the GUI was not possible, instead a VS project without GUI was created, which can send and receive RMAP packets. Thus, the continuation of this work is easier.

The example is again based on the *advanced_send_example.c* and the *advanced_receive_example.c*, which use the STAR-API. Additionally, the RMAP functions, e.g. a check value (CRC) calculator, are used. Therefore, the headers (*star*) and libraries (*lib*) folders are included as well as the utilities (Chapter 4.2.4) and the *readFromFile()* function (Chapter 4.3.1).

RMAPexample.h The header contains all relevant macros and function declarations for the project.

main.cpp The main function calls either the *RMAPSend()*, the *RMAPReceive()* function or both, depending on which is commented out.

RMAPSend.cpp This file contains the *RMAPSend()* function which creates an example packet from the data loaded out of *RMAP.txt*, including header and data CRCs and sends it via SpaceWire.

RMAPReceive.cpp This file contains the *RMAPReceive()* function which prints the received SpaceWire packet and checks its CRCs.

4.6 Challenges

As in almost every project, certain challenges arose. Finding the right development environment turned out to be challenging. The difficulty of freezing GUIs could be solved relatively fast by using multithreading. The broken hardware was a bit more complicated to replace and the sudden change to RMAP has not been completed yet.

4.6.1 Development Environment

At the beginning of the work a suitable coding environment had to be selected. Initially the software provided by Qt [Qt21] was used. When handling the external libraries and source files of StarDundee, this tool was inappropriate. Next, the environment Eclipse by Oracle [Ecl21] was tested. The linker issued an error, stating that it could not find the header file *sal.h*. Through internet research it could be identified as an internal file of VS. Subsequently, the program was changed again. VS itself had some initial difficulties as well concerning the linker and compiler settings (cf. Chapter 4.2.1) but they could all be solved.

4.6.2 Multithreading

The first approach to the matter was to implement all functions in one thread. As mentioned above, when putting a function to sleep, either through *Sleep()* or *STAR_waitOnTransfer-OperationCompletion()*, the GUI freezes and the data simulator cannot be controlled or even closed.

Although this behavior is obvious, it created additional workload. Fortunately, Qt is equipped with the developer-friendly multithreading package *‹QThread›*. Together with *‹QMutex›* it guarantees thread safety. When using a variable *mutex*, shared variables are blocked from undefined behavior. If a thread has called *mutex.lock()* and has not called *mutex.unlock()* yet, other threads calling *mutex.lock()* have to wait until it has been unlocked again.

Upon closing the GUI the destructor *MainWindow::~MainWindow* is called. It ensures that the boolean *stop* is set and the main thread waits until the termination of the background. Otherwise, channels would not be closed and allocated memory would not be freed. The destructor of the Send GUI in Source Code 3 shows the use of *mutex*, the access to variables of the derived thread, and *wait()*.

```
1 MainWindow::~MainWindow() {
2   /* Upon closing the GUI */
3   /* Stop the receive thread */
4   bc_thread->mutex.lock();
5   bc_thread->stop = true;
6   bc_thread->mutex.unlock();
7   /* Wait until the thread stops */
8   bc_thread->wait();
9 }
```

Source Code 3: Destructor of the Send GUI

4.6.3 Broken Hardware

After creation of the Receive GUI and checking the transmitted packets for errors, discrepancies between sent and received messages were detected. Possible reasons for this were examined and invalidated. Among them, of course, checking the code for errors, restarting the OS, power cycling the Brick, and checking every cable and every plug.

While investigating the bug with StarDundee GUIs in binary mode, it was noticed that it occurred very regularly. In every fourth byte, starting with the third, the second bit was set to "false", no matter the circumstances. When sending messages with all bits set to "true", the fault could be seen easily through the received bits set to "false" (cf. Fig. 14).

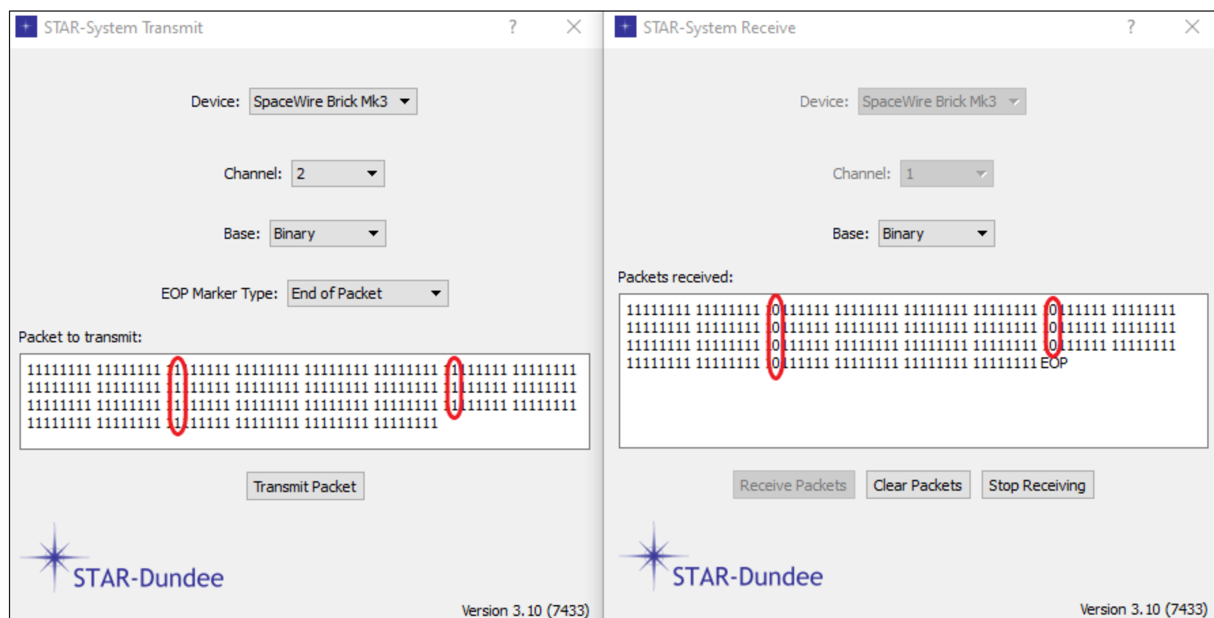


Figure 14: StarDundee's Transmit (l) and Receive (r) GUIs showing the defect of the Brick Mk3

This extraordinary regularity indicated a systemic error. Because the reason for the error could not be found on the software side, StarDundee's customer service was contacted. Their kind support sent the Brick Mk4 as a replacement. Nevertheless, it proved to be difficult to send the faulty Brick Mk3 for repair to Scotland since the United Kingdom has left the European Union. After several time consuming attempts of shipping it, it finally worked out. The device is still in Dundee for repair.

4.6.4 Protocol change to RMAP

Shortly before completion of the data simulator with the protocol of the CCSDS, it was changed to RMAP. This has made changes to the header necessary, which could not be realized in the remaining time. When comparing the two protocols (Chapter 3) the headers are very different. A sample project, in which the general behavior of the API in connection with RMAP was tested, is included.

5 Conclusion

5.1 Summary

The aim of this work has been the creation of a data simulator for the DDA. The overall functionality has been achieved. Two GUIs were developed, one sending data as broadcast or telecommand, the other receiving telemetry. Additionally, both of them can process raw data without a header.

At the beginning of the work, it was considered realistic that all specifications of the packages would be known before submission of the thesis. As time went on, it became clear that a lot of work had to be done with assumptions. In consultation with the IRS, it was determined that telecommands and broadcast should be controlled primarily via free-text fields. Also, data can be sent from *.txt* files. Telemetry, by contrast, is sufficiently defined. In addition to the headers, the Source Data Field is also specified. The received packets are parsed accordingly and saved into *.csv* files. Therefore, data can be processed into a spreadsheet.

The change of protocol to RMAP was announced only during the writing of the report. Therefore, the data simulator stayed in its original form and a draft of the RMAP version was additionally submitted. This guarantees better comprehensibility and at least one functional version.

5.2 Evaluation

The work can be classified as an entry into implementation. Important literature has been evaluated and a coding frame has been issued. All basic functions work impeccably. The IRS can now start testing their experiment. When proceeding with the development of the simulator, the needed software, hardware, and environments can be inherited. However, the project fell short of expectations. During the development of the source code, the packet definitions were not extended. On the contrary, they were discarded altogether by changing the protocol.

Nevertheless, the GUIs are much better adapted to the DDA when compared to the basic StarDundee programs, regardless of protocol. They reduce the user's workload by creating a workspace suitable for the needed functionality. The changes in protocol are relatively easy to implement because the existing source files provide examples. The final result can consequently be considered a success.

5.3 Outlook

The simulator is supposed to become a powerful tool of the DESTINY$^+$ mission. The IRS wants to use it not only as test equipment for the flight software but also for tests of the whole instruments. It has to be enhanced to be fit for use. The following improvements are recommended.

RMAP The GUIs should be adjusted to match the requirements of JAXA concerning the headers. There will be more than the previously assumed three message types. Once these subcategories can be displayed, their definitions can be converted into code. Note that some messages in RMAP have a Command and a Reply. Therefore, the amount of messages will increase and additionally, received and transmitted packets will have to be linked to each other. Additional functions could check for incomplete transfers by evaluating the returning information.

Specify Telecommands It should be possible to enter telecommands and their parameters in drop-down menus and spin boxes on the GUI. The free-text field would be obsolete. For example, the user should be able to select the command “Turn DDA” and add the angle, e.g. “10 Degrees”. The data simulator should be able to create the packet automatically with this information. This has not been carried out because the packet definitions have not been specified yet.

Specify Broadcast Similarly, all broadcast should be defined and carried out automatically. The user can input mode changes, e.g. “Safe Mode On” or “Off”. Other information, like time or temperature, should be incorporated. This would replace the free-text makeshift. JAXA is responsible for determining the content of the broadcast, which it has not finished yet.

Data Model for Telemetry Once the contents of telemetry packets are defined more precisely, the simulator could be equipped with a detailed sorting algorithm. A distinction could be made between housekeeping and science data, incomplete or questionable data could be separated from trustworthy, etc.

6 References

6.1 Articles and Conferences

[Ara17] T. Arai et al. "DESTINY$^+$". In: *17th Meeting of the NASA Small Bodies Assessment Group* (2017). URL: https://www.lpi.usra.edu/sbag/meetings/jun2017/presentations/Araia.pdf.

[Mas18] M. Masanori, R. Srama, et al. "DESTINY$^+$ Dust Analyzer". In: *49th Lunar and Planetary Science Conference* (2018). URL: https://www.hou.usra.edu/meetings/lpsc2018/pdf/2050.pdf.

[Som20] M. Sommer, R. Srama, et al. "Destiny$^+$ Dust Analyzer – Campaign & timeline preparation for interplanetary & interstellar dust observation during the 4-year transfer phase from Earth to Phaethon". In: *Europlanet Science Congress* (2020). URL: https://doi.org/10.5194/epsc2020-342.

[Toy17] H. Toyota et al. "DESTINY$^+$: Deep Space Exploration Technology Demonstrator and Explorer to Asteroid 3200 Phaethon". In: *Low-Cost Planetary Missions Conference* (2017). URL: https://web.archive.org/web/20170914034331/http://www.lcpm12.org/wp-content/uploads/2017/08/1415-1435-Toyota.pdf.

6.2 Books and Guides

[CCS10] CCSDS. *Time Code Formats, Recommended Standard (301.0-B-4)*. Vol. 4. CCSDS, 2010. URL: https://public.ccsds.org/Pubs/301x0b4e1.pdf.

[CCS20] CCSDS. *Space Packet Protocol, Recommended Standard (133.0-B-2)*. Vol. 2. CCSDS, 2020. URL: https://public.ccsds.org/Pubs/133x0b2e1.pdf.

[ECS10] ECSS. *SpaceWire - Remote memory access protocol (ECSS-E-ST-50-52C)*. ECSS-E-ST-50-52C. ESA, 2010. URL: https://ecss.nl/standard/ecss-e-st-50-52c-spacewire-remote-memory-access-protocol-5-february-2010/.

[IEE17] IEEE. *Portable Operating System Interface Base Specifications (IEEE Std 1003.1)*. Vol. 7. The Open Group, 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/.

[Par12] S. Parkes. *SpaceWire User's Guide*. STAR-Dundee Limited, 2012. URL: https://www.star-dundee.com/wp-content/star_uploads/general/SpaceWire-Users-Guide.pdf.

6.3 Other References

[CCSMA] CCSDS. *CCSDS Member Agencies*. 2021. URL: https://public.ccsds.org/participation/member_agencies.aspx.

[Des21] DESTINY+ Project Team. "C&DH Overview". Internal Document. 2021.

[Ecl21] Eclipse Foundation. *Eclipse*. Version 4.20. 2021. URL: https://www.eclipse.org/.

[IRS19] Institute of Space Systems. "DESTINY+ Dust Analyzer". Internal Presentation. 2019.

[MicVS] Microsoft Corporation. *Visual Studio 2019*. Version 16.9. 2021. URL: https://visualstudio.microsoft.com.

[MicWi] Microsoft Corporation. *Windows 10*. Version 10.0. 2015. URL: https://www.microsoft.com/windows/.

[Qt21] The Qt Company. *Qt*. Version 6.0. 2021. URL: https://www.qt.io/product/qt6.

[QtWin] The Qt Company. *Qt for Windows - Deployment*. 2021. URL: https://doc.qt.io/qt-6/windows-deployment.html.

[StaMk] STAR-Dundee Ltd. *SpaceWire Brick Mk3*. Factsheet. URL: https://www.star-dundee.com/wp-content/star_uploads/product_resources/datasheets/SpaceWire-Brick-Mk3_0.pdf.

[StaSy] STAR-Dundee Ltd. *STAR-System*. Factsheet. 2020. URL: https://www.star-dundee.com/wp-content/star_uploads/product_resources/datasheets/STAR-System.pdf.

[Wik21] Wikipedia. *List of filename extensions*. Wikimedia Foundation Inc. 2021. URL: https://de.wikipedia.org/wiki/Liste_von_Dateinamenserweiterungen.

A Zusammenfassung (German Summary)

A.1 Motivation

Ziel dieses Projekts ist die Erstellung von GUIs, die die Kommunikation zwischen der Raumfahrtmission DESTINY$^+$ und dem dazugehörigen Staubteleskop DESTINY$^+$ Dust Analyser (DDA) simulieren sollen. Dieser Bericht fasst den Entwicklungsprozess zusammen, unterteilt in die Anforderungen (Kapitel 3) und die Umsetzung (Kapitel 4).

Der Sinn dieses Datensimulators ergibt sich daraus, dass das Instrument im Auftrag des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR) am Institut für Raumfahrtsysteme (IRS) der Universität Stuttgart in Deutschland gefertigt wird, während die Mission ein Projekt der Japan Aerospace Exploration Agency (JAXA) ist. Die geographische Distanz zwischen den Modulen macht ein frühzeitiges Testen schwierig. Eine wirtschaftlichere Alternative ist dieser Simulator. Nicht nur der Datentransfer von und zum Staubteleskop wird damit getestet werden, auch der gesamte DDA kann bei Experimenten, z. B. am institutseigenen Staubbeschleuniger, über die GUIs angesteuert und überwacht werden.

A.2 Anforderungen

Der DDA kommuniziert über SpaceWire mit dem Satelliten. Dieses Bussystem wurde von der europäischen Raumfahrtbehörde (ESA) als Standard für die Raumfahrt entworfen und kann Rohdaten versenden. Um eine standartisierte Syntax zu garantieren, werden hauptsächlich zwei Nachrichtenprotokolle genutzt. Eines von CCSDS und eines von der ESA selbst, namens RMAP. Vorgabe für die vorliegende Arbeit war, jenes von CCSDS zu nutzen, so wie es die ESA empfiehlt. Gegen Ende der Bearbeitungszeit wurden die Spezifikation von JAXA konkretisiert. Die Kommunikation läuft nun über RMAP.

Unter der Annahme, die Pakete würden dem Protokoll des CCSDS folgen, wurde der Rahmen für drei verschiedene Pakete erstellt. **Telemetrie** soll vom DDA gesendet und somit vom Datensimulator empfangen und verarbeitet werden. Der Inhalt dieser Pakete war zu Beginn der Bearbeitung vom IRS ausführlich definiert. Vom Datensimulator zum Experiment werden **Telecommands** (direkte Anweisungen an den DDA) und **Broadcast** (allgemeine Informationen und Anweisungen für alle Experimente der Mission) versandt. Deren Spezifikationen waren noch nicht vollständig, deshalb sollte die GUI Freitextfelder zur manuellen Eingabe von Daten haben.

Die neuen Anforderungen zu RMAP verfügen über mehr verschiedene Pakete, jedoch bleibt die Anforderung an die im Rahmen dieser Arbeit entwickelte Software, Nachrichten zu senden, die Anweisungen oder Informationen transportieren, und Telemetrie zu empfangen. Bei einigen RMAP-Paketen werden CCSDS-Header zusätzlich intern verwandt. Dennoch sind die Protokolle grundverschieden und manche RMAP-Pakete verlangen Antworten des Kommunikationspartners, was dazu führt, dass die zuvor erstellten Spezifikation teilweise hinfällig sind.

A.3 Umsetzung

Die Umsetzung des Simulators erfolgte in Qt in der Entwicklungsumgebung Microsoft Visual Studio 2019 (VS) auf Microsoft Windows 10. Qt ist eine C++ Erweiterung für die Erstellung von GUIs. Zusätzlich wurde die Hardware von STAR-Dundee Ltd. aus Schottland in Form eines „SpaceWire Brick Mk3“ bzw. „Mk4“ und die dazugehörige Schnittstelle (API) genutzt. Die Bricks werden mit USB an den PC angeschlossen und wandeln die zu sendenden Bytes in ein SpaceWire-Signal um. Empfangene Nachrichten werden entschlüsselt.

Zunächst wurden gemäß der Anforderungen zwei GUIs entwickelt, eine zum Senden und eine zum Empfangen. Die empfangenen Nachrichten werden in eine *.csv*-Datei gespeichert, um sie anschließend in einem Tabellenkalkulationsprogramm auswerten zu können. Vor dem Senden kann der Inhalt der Nachricht aus einer *.txt*-Datei importiert werden, wenn das Textfeld zu umständlich erscheint. Zusätzlich können Parameter, wie Periode oder Header, eingestellt werden.

Die Änderungen zu RMAP konnten aus Zeitgründen leider nicht mehr vollständig in die GUIs aufgenommen werden. Stattdessen wurde jedoch ein Beispielprogramm zur Demonstration erstellt, welches RMAP-Nachrichten versenden und empfangen kann. Dabei wurde Wert auf die Kommentare und den modularen Aufbau gelegt, wodurch nachfolgenden Personen der Einstieg in dieses Projekt erleichtert werden soll.

A.4 Fazit

Während der viermonatigen Bearbeitungszeit konnte ein funktionierender Datensimulator geschaffen werden. Die initialen Anforderungen werden damit erfüllt. Die GUIs sind deutlich besser an den DDA angepasst, als die Software von STAR-Dundee. Leider kam die Konkretisierung der Spezifikationen zu spät, um sie im Rahmen der Arbeit vollständig umsetzen zu können. Das Projekt muss also fortgesetzt werden. Dabei kann das Send-GUI erweitert werden, sodass die verschiedenen Nachrichten und deren Parameter durch Auswahllisten und andere Eingaben eingestellt werden, anstatt das Freitextfeld nutzen zu müssen. Das Empfangs-GUI kann beispielsweise um einen Sortieralgorithmus ergänzt werden, der die empfangenen Daten verschiedener Messungen trennt.

B Files

The following files are submitted along with this thesis. Each subsection represents one VS project and they all access the libraries *lib* and headers *star* included in their folder for this purpose.

B.1 Send GUI

The project's name is "Send03", therefore the project files and the executable carry this name.

- Executables
  - *Send03.exe* (release version, incl. *Release* directory)
- VS project files
  - *Send03.sln*
  - *Send03.vcxproj*
- Form files
  - *mainwindow.ui*
- Header files
  - *broadcast_thread.h*
  - *mainwindow.h*
  - *readFromFile.h*
  - *utilities.h*
- Source files
  - *broadcast_thread.cpp*
  - *main.cpp*
  - *mainwindow.cpp*
  - *readFromFile.cpp*
  - *send_telecommand.cpp*
  - *send_vanilla.cpp*
  - *utilities.cpp*
- Text files
  - *broadcast.txt*
  - *telecommand.txt*

B.2 Receive GUI

The project's name is "Receive01", therefore the project files and the executable carry this name.

- Executable
  - *Receive01.exe* (release version, incl. *Release* directory)
- VS project files
  - *Receive01.sln*
  - *Receive01.vcxproj*
- Form files
  - mainwindow.ui
- Header files
  - *mainwindow.h*
  - *receivethread.h*
  - *utilities.h*
- Source files
  - *main.cpp*
  - *mainwindow.cpp*
  - *receivethread.cpp*
  - *utilities.cpp*
  - *writeToFile.cpp*
- Text files
  - *telemetry.csv*
  - *no_header.csv*

B.3 RMAP Example

The project's name is "RMAP01", therefore the project files and the executable carry this name.

- Executable
  - *RMAP01.exe* (release version, incl. *Release* directory)
- VS project files
  - *RMAP01.sln*
  - *RMAP01.vcxproj*
- Header files
  - *RMAPexample.h*
  - *utilities.h*
- Source files
  - *main.cpp*
  - *readFromFile.cpp*
  - *RMAPSend.cpp*
  - *RMAPReceive.cpp*
  - *utilities.cpp*
- Text file
  - *RMAP.txt*