Rover Deployment Software System

The Brains Behind the Rover's pod

Bachelor Graduation Thesis

H. Vanhuynegem
D.Y. Aris

Delft University of Technology

TU Delft

LUNAR ZEBRO • subsulteo aude ad

Rover Deployment Software System

The Brains Behind the Rover's pod

by

H. Vanhuynegem
D.Y. Aris

Student Name	Student Number
Henri Vanhuynegem	5252687
Diederik Aris	5345790

Instructor:	Dr. ir. C.J.M. Verhoeven
Project Duration:	April, 2024 - June, 2024
Faculty:	EEMCS, Delft

Cover:	Lunar Zebro Render (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

This document represents the culmination of our Bachelor Graduation Thesis project, undertaken at the Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS) at Delft University of Technology. The primary objective of this project was to develop a comprehensive software system to manage the deployment of the Lunar Zebro rover on the lunar surface. Throughout this project, we faced numerous technical challenges, from understanding the complexities of communication protocols to designing a system capable of operating autonomously through a space mission.

Our work was greatly supported by the guidance and expertise of Prof. Dr. Ir. C.J.M. Verhoeven and Prof. Dr. Ir. G.N. Gaydadjiev, whose insights were invaluable in navigating this project's technical intricacies. We also extend our gratitude to the Lunar Zebro team for their collaborative spirit and the opportunity to contribute to this pioneering mission.

This thesis is structured to provide a detailed account of the design process, technical challenges, and testing methodologies employed to ensure the success of the RDSS. We hope that the findings and methodologies presented here will not only contribute to the success of the Lunar Zebro mission but also provide a foundation for future developments in the field of space deployment systems.

H. Vanhuynegem
D.Y. Aris
Delft, June 2024

Abstract

The Rover Deployment Software System (RDSS) is a critical component designed to ensure the successful deployment of the Lunar Zebro rover onto the lunar surface. This thesis presents the design, implementation, and testing of the RDSS, which consists of three primary subsystems: a communication system between the lander and the RDSS, an electronic control system, and an integration with an existing rover communication system. Moreover, the existing rover communication system will not be covered in this thesis due to the implementation being done by the Lunar Zebro team in the future. The RDSS is tasked with managing the deployment sequence, providing power during transit, and facilitating communication between the rover and the lander. Key challenges addressed include handling the harsh lunar environment, ensuring reliable communication, and adhering to strict weight constraints. Extensive testing, including unit, integration, system, and performance tests, validated the system's robustness and reliability. The insights and methodologies developed are intended to support the Lunar Zebro mission and inform future projects involving space deployment systems.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
RDSS	Rover Deployment Software System
RDS	Rover Deployment System
RDCS	Rover Deployment Control System
EECS	Electronic Components Control System
LZ	Lunar Zebro
CS	Control System
MCU	Microcontroller Unit
SLIP	Serial Line Internet Protocol
NEA	Non Explosive Actuator
GND	Ground
Vcc	Common Collector Voltage
SEC	Standard Electrical Cable
ADC	Analog to Digital Conversion
DC	Direct Current
AC	Alternating Current

1

Introduction

Lunar Zebro is ”World’s smallest and lightest rover yet, built by TU Delft students” [10], a rover that is being designed for space missions. For the current iteration, the focus of the Lunar Zebro team is sending the rover to the moon as a piggyback payload, which is attached to a larger lander. After the lander makes contact with the lunar surface, it will send a signal indicating that the Rover Deployment System (RDS) should release the rover onto the lunar surface. Thus, the primary goal of the RDS is to release the rover onto the lunar surface upon receiving a deployment signal. The rover should not be released at any other moment.

The mechanical part of the RDS which will physically deploy the rover has mostly been designed by the Lunar Zebro team. During the development, the need arose for an electronic control system that controls the mechanical system, allows the rover to communicate with the lander, and that provides power from the lander to the rover. Therefore, the RDS is a crucial system that aids in the rover’s survival.

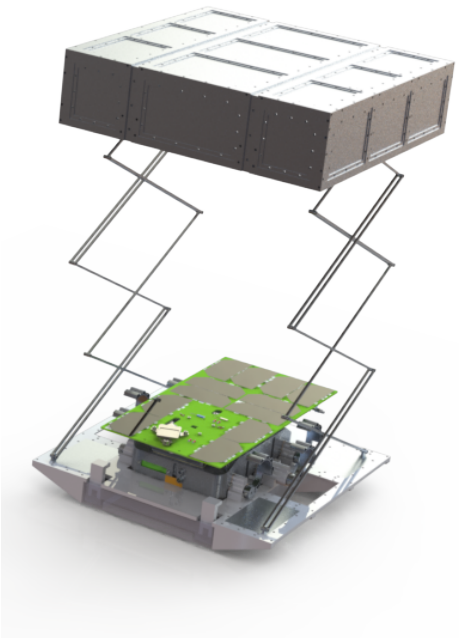

Figure 1.1: Rover Deployment System

State-of-the-Art Analysis

Space exploration started in 1957 and has since grown into a 630 billion dollar industry [24]. Large agencies such as NASA, ESA, and many more are developing various projects to extend human knowledge about extraterrestrial life. A nanosatellite is the most similar product available on the market compared to the Lunar Zebro rover since it weighs between 1-10 kg and operates in the same harsh environments[15]. These nanosatellites are deployed into space to orbit in the Low Earth Orbit zone, which lies between 200 to 2,000 kilometres above Earth. The deployment system used for these nanosatellites is the product most similar to the Rover deployment system, as it functions likewise in similar environments. An example would be the ISIPOD CubeSat Deployer, created by ISISPACE [7]. However, no information about how the control system is designed can be found online. Many scientific papers are available about the development of mechanical deployment techniques, such as the paper by Wang [23]. Many space agencies maintain a lot of proprietary information, which makes it difficult to find information about the electrical control systems used in such deployers. For this reason, the Rover Deployment System was designed. This system has 3 objectives:

1. Deploy the Lunar Zebro rover on the surface of the moon.
2. Provide power from the lander to the rover during transit when needed.
3. Function as an intermediary for communication between the rover and the lander.

The Lunar Zebro Rover Deployment System is thus a first-of-its-kind system designed specifically to transport a nano rover to the moon or other extraterrestrial planets.

Many limitations are posed on the design of the RDS. The harsh environments, with temperatures ranging from -173.15 to 126.85 degrees Celsius and solar flares occurring [11]. Every gram costs approximately 1000 dollars to be transported to the lunar surface. Weight is thus of significant importance when designing the RDS. The Lunar Zebro team aims to produce a lightweight/cost-effective rover that could be used for swarming in the future. Thus, it is important to consider the cost of the components used.

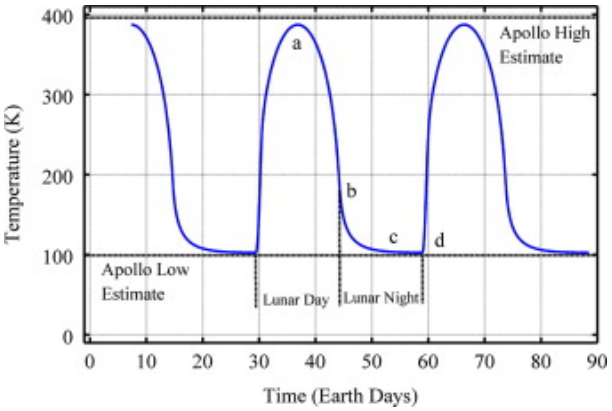

Figure 1.2: Lunar surface temperature throughout the lunar cycle.

This study is meaningful for the immediate objective of supporting the Lunar Zebro mission and its broader implications. By developing a high-level design that will be decomposed into various subsystems and collaboratively refining design choices with the Lunar Zebro team, this research aims to create a high-quality, functional, and adjustable system. The insights and methodologies developed here could inform future projects and contribute to the broader field of electronic systems for space deployment contraptions. This research will help the Lunar Zebro team build an electronic system for their RDS and potentially aid future teams in developing comparable systems for their deployment mechanisms.

Thesis synopsis

This thesis focuses on building the software system required to deploy the LZ rover on the Lunar surface successfully. The Rover Deployment Software System (RDSS) consists of three main subsystems, two of which will be discussed in this thesis: a communication system for the RDS and the lander, and an electronic component control system. Additionally, a communication system for the RDS and the rover is needed. However, since the Lunar Zebro team has already designed their functioning communication system, they will implement it into the RDSS. The interaction between multiple subsystems of the RDS can be seen below in Figure 1.3. The highlighted section contains an overview of the interaction between the RDSS and other subsystems. The design of the voltage rails and sensing and actuating subsystems lie outside the scope of this thesis, as these elements are designed by different sub-groups [8] [19]. The code of the RDSS can be found on GitHub using the following URL: https://github.com/Hvanhuynegem/BAP-Rover-Deployment-Software-System [5].

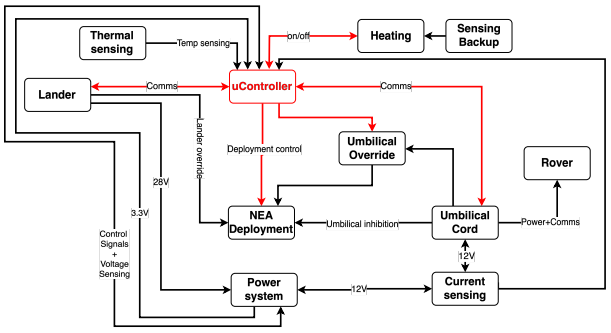

Figure 1.3: System overview

2

Programme of requirements

2.1. Introduction

Good requirements must be set to build any reliable and qualitative system. The RDS system has multiple requirements which need to be satisfied. These requirements are essential for a successful Lunar Zebro mission. The RDS requirements can be split into functional and non-functional requirements. These, together, are the requirements that should be satisfied by the designed system for the system to be qualified as fully functioning.

The RDS department at Lunar Zebro is a relatively new department. Currently, it officially only consists of a mechanical department, without a control unit. The current RDS's mechanical department has designed a pod that will be attached to the rocket and a mechanism that lowers the Rover onto the Lunar surface. The addition of the RDCS will provide multiple capabilities. RDCS will supply power, do checkups on the Rover (while in transit), facilitate communication between Lander and Rover, and will be in charge of triggering the deployment. The RDCS will handle everything in regards to deployment up until the latch of the pod opens up and the lowering mechanism lowers the Rover onto the lunar surface.

The RDCS will be formed by three subsystems that, when working in unison, ensure the system functions properly. The subsystems are the software system, the Power system, and the actuation and sensing system.

2.2. Requirements for the entire system

Functional requirements

[1.1] The system should be able to actuate 4 Non-Explosive Actuators (NEA)

[1.2] An umbilical cord must be used to connect the electronic RDS to the Rover

[1.3] The system should make all unconsumed power available to the rover

[1.4] The rover must remain fixed to the RDS pod unless it deploys

[1.5] There should be a thermal control system on the RDS

[1.6] The system should release the rover by actuating four NEAs

[1.7] Fail-safe backups should be implemented to prevent single points of failure

[1.8] The system must relay data from the Rover to the Lander and vice versa

Non-functional requirements

[2.1] The system should be able to actuate two types of NEA, the NEA® Model 9040 Miniature Hold Down & Release Mechanism (HDRM) and NEA® Model 1120-05 Pin Puller

[2.2] The system should be able to operate in an environment with temperatures between -120 and $+120°C$

[2.3] The system should be able to withstand vibrations experienced during launch, transit, and landing

[2.4] It can be assumed that the system operates in a Faraday cage. Therefore, no radiation will influence the system.

[2.5] The electronic RDS has a mass budget of 200 grams

[2.6] The electronic RDS must be smaller than 20cmx20cmx10cm

[2.7] The system should be able to operate on a 3W, 28V DC supply rail

[2.8] The system must achieve 99.9% reliability to release the rover and the pod-latch

2.3. Requirements for the Rover Deployment Software System

Functional requirements

Functional requirements specify what the system should do. They define the functionalities or services that the system must provide.

[A.1] The RDSS should be an autonomous system that is able to recover from a power failure.

[A.2] The system should act on every single incoming message from the lander.

[A.3] The system should provide a reliable way of deploying the LZ rover onto the Lunar surface.

[A.4] The RDSS should provide an algorithm for temperature control to survive the harsh environment of space.

[A.5] The RDSS should be a control loop system that is able to recover from every error, therefore operating autonomously.

[A.6] The system should never deploy the rover unless it gets a deployment signal from the lander.

[A.7] The RDSS should function as a data relay system between the Rover and Lander.

Non-functional requirements

Non-functional requirements specify how the system performs a function. They define the quality attributes, performance metrics, and constraints of the system.

[B.1] The software must be able to operate in extreme space conditions.

[B.2] The RDSS should prioritize reliability over performance.

[B.3] The Misra C 2012 guidelines must be followed for software design.

[B.4] The code must be compiled and tested using Code Composer Studio from Texas Instruments.

[B.5] Code should be tested using Google Unit Tests.

[B.6] Each algorithm needs to have 100% branch/line coverage on its unit tests.

[B.7] Software should be programmed in a modular manner.

[B.8] The UART output/input pins 2.5 and 2.6 should be used for the RDS-lander communication protocol.

[B.9] The UART output/input pins 2.0 and 2.1 should be used for the RDS-rover communication protocol.

[B.10] MSP430FR5969SP MCU must be used.

[B.11] The msp-exp430fr5969 development board must be used for testing.

[B.12] The software must be written in either C or C++.

3

System Design

3.1. Integrated System

The whole integrated system must be capable of fulfilling all the requirements. Because it involves a large system, it is naturally subdivided into smaller sections. This section aims to provide an overview of how all the different sections and functions are bundled together to create this large RDSS. It won't go into much detail about the created functions and the right implementations but rather sets out the design decisions made to create a coherent system. First, the system architecture of the whole integrated system will be explained. Then, it goes into more detail about the different transit modes.

3.1.1. System Architecture

For the RDS, the MCU is expected to perform three different kinds of tasks:

- Send and receive different kinds of messages to and from the lander
- Check the status of the different electrical components of the RDS
- Actuate different components of the RDS

The first task is described in much more detail in section 3.2 and the last two tasks in section 3.3. However, these sections only provide an explanation of their respective tasks. Therefore, this chapter provides an explanation of where these two subsystems belong in the full system.

The RDSS starts with an initialization of all the pins used in the MCU. This involves defining if they are used as input or output and selecting the right pin mode. Furthermore, the sub-main clock, the ADC module, and the different timers used by the MCU are initialized. With this initialization, it is important to note that it is done in such a manner that all the electrical components are in the default mode. This would mean that an active low pin to turn on a temperature sensor is initialized as high, for instance. The whole mission is subdivided into different transit modes, ranging from the moment the rover is attached to the lander to the moment the rover is released on the moon. These transit modes coincide with large external changes during the mission. The RDSS must behave differently during the different transit modes. The transit modes can be seen in Figure 3.1.

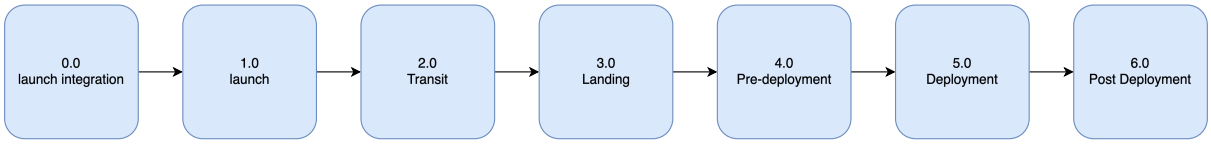

Figure 3.1: Different stages of the mission

These different transit modes are implemented in the code via a switch statement. The lander communicates the correct mode to the RDSS. Besides these transit modes, there is one extra mode. This mode is called the general startup. Every time the MCU turns on again (right before or during the mission), it defaults to the general startup mode. It stays in this mode until it receives the correct mode from the lander. The implementation of the different transit modes is elaborated further in the next subsection.

3.1.2. Transit modes

A separate functional flow diagram is made for each transit mode, including the general startup. These diagrams can be found in appendix B.1. The colours used in these diagrams are defined as follows: The white blocks represent sequential actions. The orange blocks represent actions that are part of a loop. This loop can be broken at any time by an interrupt caused by the lander's new transit mode message. The purple blocks represent actions that are not implemented in this project. These actions are related to a data connection with the rover. However, due to time constraints, this implementation has not been made. It is possible to implement these actions in the future.

General Startup

Figure B.1 shows the functional flow diagram of the general startup mode. Important to keep in mind, is that the MCU switches to this state whenever it starts, This could also happen during the mission after a system outage. The connection with the lander is crucial, as the lander is the only thing that can send the MCU the transit mode and especially the green light to deploy. However, since the general startup can happen during the mission when nothing can be manually changed, the system will move on after it fails to connect with the lander three times. Even if the initialisation sequence fails, it moves on to the RDS electronics status check in the hope that it may still receive a transit mode message from the lander, although the lander failed to send an acknowledgement back. It keeps checking the status of all the electrical components until a message with the correct transit mode is received from the lander. The results of these checks are always sent back to the lander.

Launch mode

Launch mode is pretty similar to general startup mode. Figure B.2 shows the functional flow diagram for the launch mode. The only difference is in the possibility to manually change things. Since this transit mode happens before launch, it is assumed that it is still feasible to plug cables back in. Therefore, it will keep trying to connect with the lander. The cables could be checked manually if the connection isn't made. The same holds for the umbilical cord connection with the rover. If the umbilical cord is not connected, an error message will be sent to the lander until it is restored manually. Although it is not implemented in this project, the same holds for the communication connection with the rover. After these checks, the RDSS will check the status of all the electrical components until a message with the correct transit mode is received from the lander.

Travel mode

In travel mode, the system has to operate independently and is not allowed to stall. Figure B.3 shows the functional flow diagram for the travel mode. All the actions during this stage can be interrupted by the lander's message of a new transit mode. It is important to note that after the temperature sensors are checked, the RDSS will control the heater of the RDS. This is to ensure an optimal operating temperature even under harsh space conditions. A timer has also been added to this mode. The actual travel from the Earth to the moon will take three months, as the lander will first orbit the moon before it lands. For this reason, during this travel, you don't want the RDSS to continuously give a status update of the check-up of all the electronic components. A message with a status check every hour could be sufficient. The length of this timer can be changed later.

Pre-deployment Mode

The pre-deployment mode happens when the lander has already landed on the moon. This is a period of time in which the rover is waiting to be deployed. It is unclear when the LZ rover will deploy [1]. It is not smart or necessary to start actuating components in this stage of the mission. Therefore, the functional flow diagram of the pre-deployment modeB.4 is identical to that of the travel mode. The only difference in implementation is the timer. It is desirable to send a status update of the electrical components of RDS more frequently, as the deployment can start at any time.

Deployment Mode

The deployment mode is the largest transit mode and the most important. Diagram B.5 shows the full deployment process, and design choices will be explained.

When the lander sends a message containing the deployment signal, the RDS should immediately be redirected to the deployment transit mode. To begin, as much current as possible should be diverted to the supercapacitors to be charged as fast as possible. Therefore, the heaters are first switched off, and the supercapacitors are checked to ensure that they are not shorted. Each individual NEA is

checked again to report back to the ground station that the NEAs are still ready to be actuated. Next, the rover should be notified about the deployment signal to prepare for deployment. As mentioned before, this thesis will not cover communication with the rover. Subsequently, power to the rover is switched off before disconnecting the umbilical cord. Once the umbilical cord is disconnected, a check can be performed. This is done three times in case of failure, and then the deployment sequence continues, even if the umbilical does not want to disconnect.

The actual deployment of the rover can commence once all the previous steps are performed. The second part of Figure B.5 is shown below in Figure 3.2. This image shows the functionality of the actual deployment. Firstly, it is checked whether the NEA can still be deployed. In both cases, the supercapacitors will be charged, and the NEA will be activated regardless of whether the system measures whether it has already been deployed. The reason for this is that if a false value is read and the NEA has not been deployed, then the system will always try three times, increasing the odds of success. Once the supercapacitors are charged and the NEA has been activated, the system performs a second check on the status of the NEA; if it is actuated, the system continues to the next. It performs this algorithm for all four NEAs, and after all four have been actuated, or at least tried to be actuated, then the system has to wait for the rover to send a message back to base since it is unable to communicate or know whether the rover has actually landed on the surface of the moon.

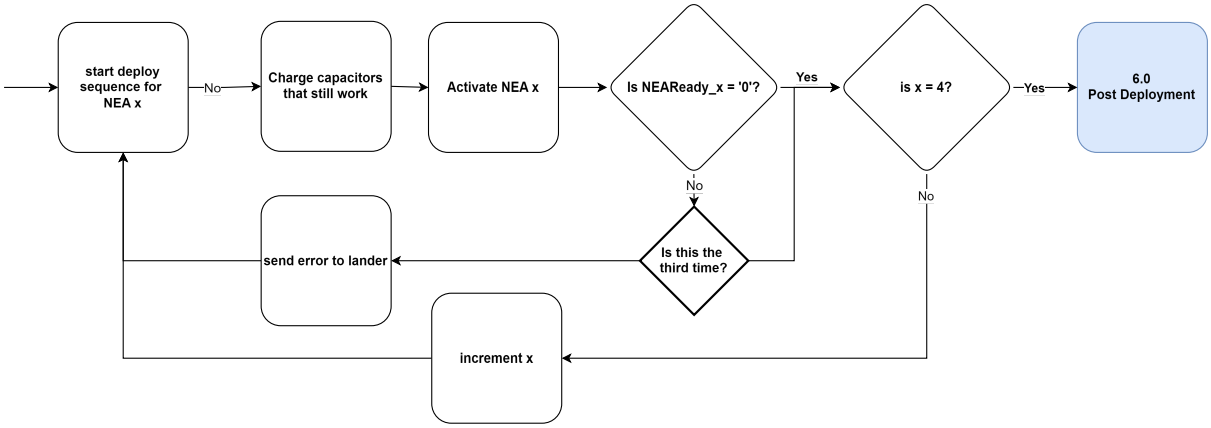

Figure 3.2: Deployment mode NEA activation

Together, these states will create a functioning RDS system that is able to aid the rover during transport and deployment. Comprehending these different transit states will help understand the design of the two subsystems explained in chapter 3.2 and 3.3.

3.2. Lander - RDS Communication System

This section presents the system design for the lander and RDS communication system through a step-by-step structure decomposition. First and foremost, the system architecture will be presented, providing a good overview of how the system works. Secondly, the different communication protocols that will be used shall be explained to provide a clear understanding of the message structure. Additionally, the software design shall be explained using functional flow diagrams so that there is no need to understand C++. Furthermore, the message handler will be elucidated. Additionally, a test plan will be presented in chapter 4 to guarantee that the different functions are working properly.

3.2.1. System Architecture

The system's architecture can be divided into hardware and software. The hardware architecture has been specified by the various lander manuals provided by the Lunar Zebro team [1] [3] [12] [17]. These specified that an RS-422 full-duplex connection is required for the hardware. Furthermore, the software architecture shall be designed using the Serial Line Internet Protocol (SLIP) and a self-made communication protocol.

The hardware architecture can be viewed below in Figure 3.3. A simplified diagram of the hardware interaction between the lander and RDS is shown. The RDS requires an output and an input pin for a UART connection to an RS-422 transceiver. This can be achieved by either using *pin 2.0* and *2.1* or *pin 2.5* and *2.6* on the *MSP430FR5969*. The subgroup Sensing and Actuation found an RS-422 and Rs-485 transceiver that will be used. More information can be found in their thesis [8]. Therefore, the lander-RDS communication system will focus on designing the software and leaving the hardware to others.

Figure 3.3: System architecture of hardware

Below in Figure 3.4, an overview of the higher-level software architecture can be observed. The graph can be read as two separate processes. The first process is the transmission process. It serializes the message to be sent using the self-made communication protocol. Furthermore, it encodes the message using SLIP. Finally, it adds character-per-character into the transmit buffer of the UART pin 2.5. The second process is the receiving process, which receives character per character into the receiving buffer of the MCU and stores it into a local RX_buffer array, which can then be processed when the MCU has time. The same process applies to transmitting. However, it is reversed, and the message is handled using a handler after deserialising and decoding. These two processes will be decomposed into multiple flow diagrams to show the step-by-step functionality throughout the upcoming sections.

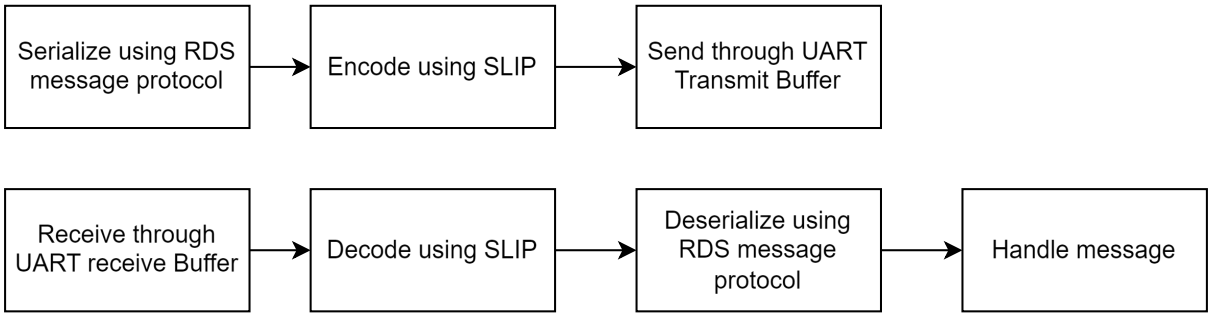

Figure 3.4: System architecture of software

3.2.2. Communication Protocols

The software architecture uses three different communication protocols, which will be explained in detail: Universal asynchronous receiver-transmitter, serial line internet protocol, and a self-made protocol that allows for better handling of incoming messages.

Universal asynchronous receiver-transmitter

The Universal asynchronous receiver-transmitter (UART) protocol is a well-known and often-used protocol for serial communication. The protocol requires a baud rate and transmits one or multiple bytes at this specified baud rate [18].

Furthermore, there are multiple different options for choosing a clock that can supply a correct baud rate for the UART module USCI_A1; there is the auxiliary clock, the sub-main clock and a baud rate clock. At first, it seemed logical to use the Auxiliary clock since it is much more stable and able to operate for years without any sign of degradation due to using a low-frequency crystal. However, the UART parameters were chosen based on the LZ team's implementation of their internal communication system. They use a sub-main clock frequency of 16MHz for all their processes, and thus, their communication system has been made to work around this frequency. This needs to be taken into consideration when building the RDSS since their communication protocol will also be added in the future. Therefore, the lander-RDS communication system is designed to operate using the sub-main clock frequency of 16MHz and a baud rate of 115200 baud/second. Furthermore, due to the conditions in space, oversampling has been activated so that incoming signals are sampled multiple times per bit period. For the MSP430FR5969, there is only one option: oversampling the signal 16 times. Additionally, the UART pins for the lander-RDS communication system are chosen to be pin 2.5 and pin 2.6 since the LZ team uses pins 2.0 and 2.1 for their communication system. Therefore, if the pin assignment is consistent, implementing their own protocol on the RDS system will be easier. After initialising these parameters using the proposed values, the UART RX and TX buffers can be used to communicate 8-bit strings.

Serial Line Internet Protocol

The Serial Line Internet Protocol or the User Datagram Protocol are both accepted as communication protocols by the Astrobotics lander [1]. When comparing both, the overhead of SLIP proved to be the more simplistic version [22] [16]. Therefore, fewer possibilities of failure exist, and more time can be spent on performing tests. The UDP protocol is also problematic since the UDP packets can get lost in the network [4]. However, UDP is more scalable and allows for the possibility of adding more addresses, while this is not possible with the SLIP protocol. Due to SLIP not having an address structure in its overhead, it must be used as a one-to-one serial communication protocol, resulting in a more reliable solution. Finally, the Lunar Zebro team approved the implementation of SLIP.

A theoretical decomposition of the protocol will be done to gain a better understanding of the structure of SLIP. According to C. M. Kozierok, four constants are used to encode/decode a message into the SLIP format [9]. The constants used in the SLIP encoding and decoding process are defined as follows:

- END (0xC0)
- ESC (0xDB)
- ESC_END (0xDC)
- ESC_ESC (0xDD)

SLIP works by sending the ”END” byte as the last character of the message. However, Kozierok also mentions that it is better also to start a message using the ”END” character, such that the receiving buffer is flushed. Moreover, whenever the character 0xC0 (”END”) is transmitted, it must be replaced by 0xDB and 0xDC. The same applies to the character 0xDB (”ESC”). However, it must be replaced by 0xDB and 0xDD. In Figure B.6, the SLIP encoding algorithm is shown to present the workings of the encoding process of a message. The pseudocode of the SLIP encoding and decoding algorithm can be found in appendix A, as algorithm 1, and algorithm 2, respectively.

The slip encoding algorithm converts an array of 8-bit characters into the SLIP format by applying the abovementioned techniques. Furthermore, it considers the possibility of an overflow occurring. This is achieved by tracking the index of the encoded message, and if the next encoded character being added results in a buffer overflow, then **false** is returned. This allows the overflow to be handled accordingly by the wrapper function. If the encoding process is successful, then **true** is returned. The same principle

is applied to the decoding algorithm 2. However, this will not be explained in detail to avoid repetition of the structure.

Self-made communication protocol

A message structure was created to handle incoming messages from the lander so different types of messages can quickly be handled and responded to. There is very little information about which kinds of message structures the different Landers use. This allows for some freedom in optimizing the design and limits the freedom to choose a very complex and specific protocol. Below in table 3.1, the basic structure is shown.

Field	Description	Size
Start Byte	Indicates the start of a message	1 byte
Message Type	Indicates the type of message. Possible values: (INIT), (ACK), (NACK), (REQUEST), (DATA), (RESPONSE), (DEPLOY), (TRANSIT_MODE), (ERROR).	1 byte
Length	Indicates the length of the payload.	1 byte
Payload	The data being transmitted.	
Checksum	A calculated value to ensure data integrity during transmission.	1 byte
End Byte	Indicates the end of a message.	1 byte

Table 3.1: SLIP Message Format

The message structure consists of a start byte, a message type which allows for a very quick handler, a length byte, the payload array, the checksum to check for errors, and an end byte. Using the message type byte, a case statement can be created to handle each individual type of message. Furthermore, the payload size is modifiable, with a maximum size of 249 bytes. To achieve this, serialisation and deserialisation are used to convert a payload array to a message structure. This allows for faster communication because there is no constant message size. The payload array is stored in the message structure as a pointer. Additionally, the length is stored so that only the message can be retrieved from the static array. There was also the possibility of using vectors. However, the Miscra C 2012 guidelines forbid dynamic memory allocation [14]. Thus, this option was removed. The serialisation algorithm will now be covered to provide enough information to recreate the system. Both the serialisation and deserialization algorithms can be found in appendix A, as algorithm 3, and algorithm 4, respectively.

The serialisation algorithm converts a message structure into an array that can be sent via the UART protocol. Figure 3.5 shows the serialisation and deserialisation processes. The method copies each member of the message structure into the correct location of the provided array. An if-statement also ensures that if the length byte is larger than the maximum payload size, then it updates the length to MAX_PAYLOAD_SIZE. Such that no memory issues can occur or the transmission buffer overflows.

The deserialisation algorithm converts an array into a message structure. First, it checks whether the array is of adequate length to be converted. Furthermore, it extracts each member from the array and stores it in the correct location of the message structure. The same checkup of the payload length is done to ensure that the message can be handled using other methods.

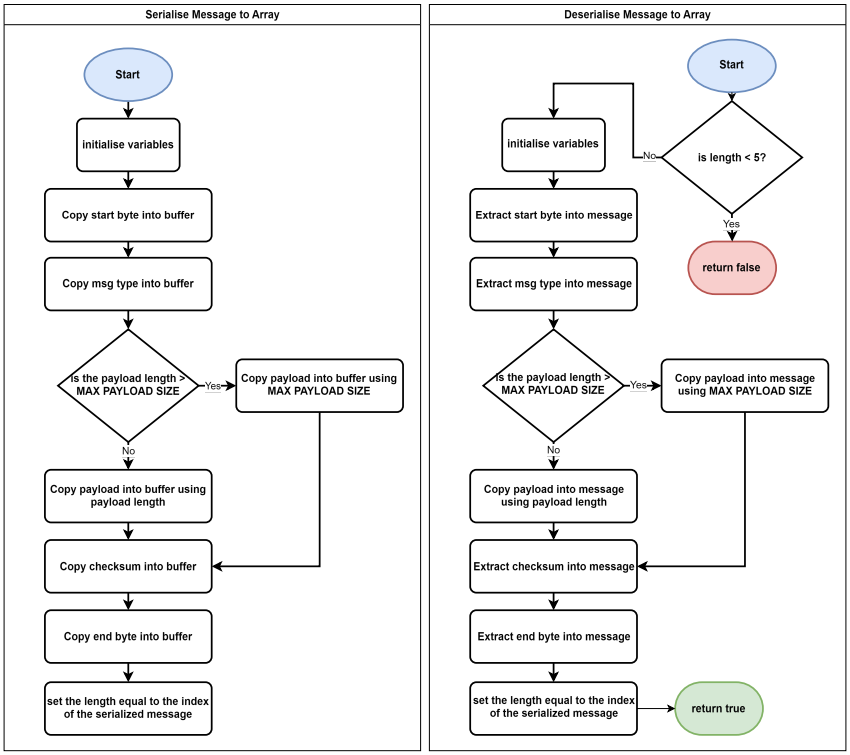

Figure 3.5: Functional flow diagram of serialisation and deserialisation algorithm

3.2.3. Software Design

The software design of the communication system can be split into transmission and receiving. These two features are crucial for a functioning communication system and shall be explained separately since they are independent of each other.

Transmission

As mentioned before, the transmission process can only be run if the UART TX pin is correctly configured and a clock is selected in combination with a baud rate. The pin used for transmission is PIN 2.5 from the module USCI_A1 [21]. The baud rate is 115200 baud/s. The clock used for the UART module is the sub-main clock with a frequency equal to 16MHz, and oversampling is switched to ensure reliable transmission. Below in Figure 3.6, the transmission process is shown using a functional block diagram.

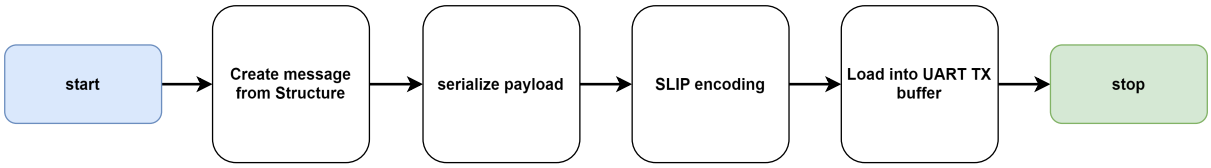

Figure 3.6: Transmission process

A message is created using the message structure to transmit an array of bytes. Furthermore, the message is serialised and thus converted to an array. Lastly, the message is encoded using SLIP. This message is loaded character per character into the TX buffer of the module USCI_A1 and transmitted. This process combines all the previously explained processes into one function called $send\_message()$.

```
1 /* Function to send a message */
2 void send_message(uint8_t msg_type, const uint8_t *payload, uint8_t length);
```

Listing 3.1: Send message function signature

Receiving

The receiving process is split up into two parts. It is partly handled by interrupts and partly handled when the MCU has time to read the RX_buffer. Figure 3.7 shows a higher-level functional flow diagram of the states where the receiving process can operate. Whenever the MCU is not receiving a message, it operates in the **IDLE** state. When an interrupt is triggered, the MCU checks whether the first character received equals the SLIP encoding character ”END (0xC0)”. If this holds, then the receiving state is changed to **RECEIVING**. Subsequently, the character is stored in the RX_buffer, and the timer is started to check for timeouts. While the MCU runs in the **RECEIVING** state, there is the possibility of having a full buffer, a timeout and an error. A timeout is achieved if the interrupt of the timer goes off. The timer is currently set at 1 ms. This is approximately 10 times the time it takes to send one byte. Since one byte is sent at a rate of $\frac{1}{115200} * 10$ bits per byte $= 86.8\mu s$. two extra bits are needed to send a byte due to the start and stop byte. Whenever the MCU goes into the **TIMEOUT** state, it sends a NACK message and resets the RX_buffer indices to 0 and the receiving state to **IDLE**. Furthermore, it waits for a retransmission. The **BUFFER_FULL** state is entered if the message being sent is larger than the size of the UART_BUFFER_SIZE. Next, it sends an error message that tells the lander that the message is too large. Finally, it also returns to the **IDLE** state and resets the RX_buffer indices. In case of an error that is not recognised, the MCU defaults to sending an empty error message. For a more in-depth decomposition of every single step, the functional flow diagram of the interrupt handler can be found in appendix B.2.2. The way the MCU handles the received message will be explained in subsection 3.2.4

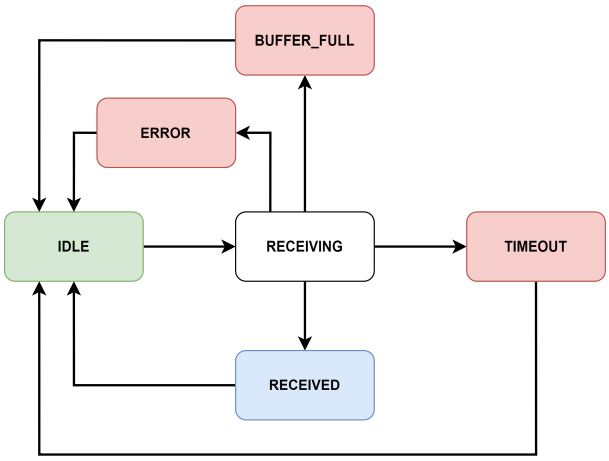

Figure 3.7: Receiving states

3.2.4. Data Handling and Processing

Message handling begins with checking the integrity of the received message. The first step is to verify that the first and last bytes of the message match the designated start and end bytes, respectively. This verification ensures that the message is neither corrupted nor incomplete. If the bytes do not match, a type ERROR message is sent with a payload containing the sentence ”INVALID_MESSAGE”.

Next, the checksum of the incoming message is verified. The handler calculates the checksum of the received message and compares it with the checksum provided within the message. If the checksums do not match, indicating possible data corruption during transmission, the handler initiates the transmission of an ERROR message with a payload containing the sentence ”INVALID_CHECKSUM”.

Upon passing these integrity checks, the handler determines the type of message received. The protocol defines several message types, each requiring specific handling:

- MSG_TYPE_INIT: Sends an ACK message
- MSG_TYPE_ACK: Does nothing
- MSG_TYPE_REQUEST: TBD handling of requests
- MSG_TYPE_RESPONSE: TBD handling of responses
- MSG_TYPE_DEPLOY: Goes to the deploy transit mode
- MSG_TYPE_TRANSIT_MODE: goes to the sent transit mode

The REQUEST and RESPONSE message types are added for future adaptability. However, this has not been implemented since there is no information about what the lander sends. A more in-depth flow diagram of the handler can be found in appendix B.2.2, Figure B.8.

3.3. Electronic Components Control System

This section will present the design of the electronic component control system. The control system consists of checks and eventual actions after these checks. This section starts with the system architecture of this part of the software. Since there are multiple electronic components, with their own functionality, the control consists of a sequence of individual checks with eventual actions after every check. These individual checks and actions will be subdivided into three different sections: Connections, temperature sensors and NEA activation. In each subsection, the importance of the check will be elaborated further, the possible actions depending on the result of the check will be explained, and a flow diagram will be presented to illustrate the functionality. Additionally, a test plan will be presented in chapter 4 to guarantee that the different functions are working properly.

3.3.1. System Architecture

It is desirable to do a full checkup of all the electronics of the RDS during different stages of the mission. The checkup will be a sequential process. This checkup has the following requirements:

- The process cannot stall if a sensor or connection is broken
- The result of every check is sent to the lander
- If an error occurs, an error message needs to be sent to the lander, and the process must continue

The RDSS is expected to check the following components. The connection with the umbilical cord to the rover, The power flow from RDS to the rover, the temperature sensors, the supercapacitors and the NEAs. Depending on the results, each check can lead to possible actions. This is illustrated in diagram 3.8.

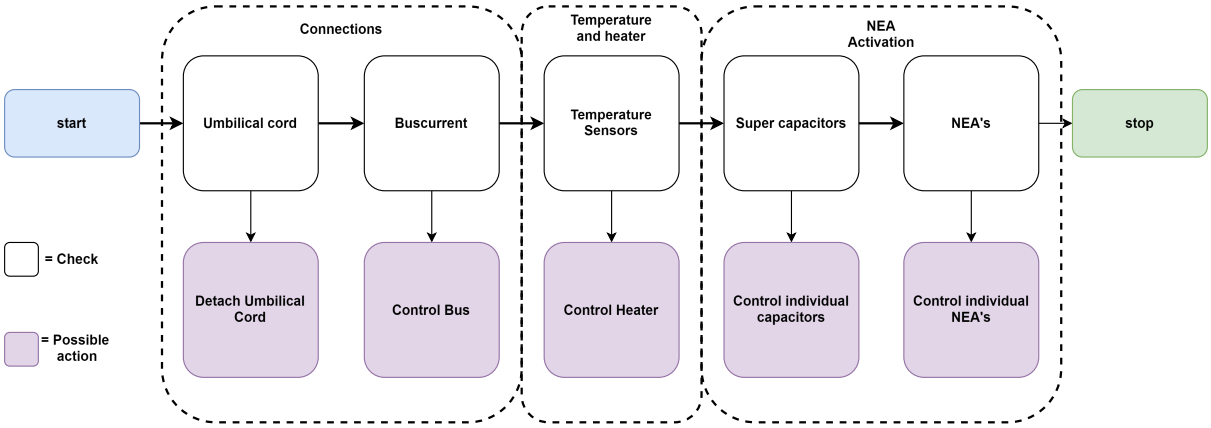

Figure 3.8: System architecture of the electronics components control system

This diagram also shows the subdivision of the different checks and possible actions into three separate sections.

The course of each check and the possible action may differ depending on the stage of the mission. You could imagine that, more intense and frequent checks are required right before the deployment of the rover compared to during transit. All these different forms of checks and possible actions are elaborated further in the following three subsections.

3.3.2. Connections

The connections check consists of two individual checks which are intertwined. The first one is to check the umbilical cord connection. The umbilical cord is a cable connection between RDS and the rover, which can be detached, preferably right before the deployment of the rover. The umbilical cord houses different smaller cables with functionalities like data and power transfer. During the mission up until right before deployment it is of great importance for the umbilical cord to be connected with the rover. All the involved pins for the connections section of the ECCS are summarized in the following table3.2.

Name	Pin	Direction	Active / Mode
Read12VFromRover	Pin 2.2	Input	General I/O mode
DetachUmb	Pin 4.6	Output	Active high
BusCurrentSense	Pin 4.3	Input	ADC mode
BusFlag	Pin 4.2	Output	Active low

Table 3.2: Connections pins

The pin ”Read12VFromRover” is actually used to sense the status of the umbilical cord connected to the rover. The pin is named ”Read12VFromRover” by the other subgroup of the project sensing and actuation[8] due to a different functionality in the beginning. To prevent confusion and to retain consistency, in this thesis the same pin names will be used as in that of the sensing and actuation subgroup [8]. The Read12VFromRover is low when the umbilical cord is disconnected, and high when the umbilical cord is still connected. A low of a pin is seen as a 0 in the corresponding input register of the MCU. Subsequently, a high pin is seen as a 1. For the Read12VFromRover pin, this means that BIT 2 of input register 2 is a 1 if the umbilical cord is still connected. This status of the corresponding bit is converted to a Boolean. See code A.1. This makes it possible to put the status of the pin in an if-else statement and program a sequential action based on the status of the pin.

After a message with the status of the umbilical cord connection is send, a possible action is to detach the umbilical cord. This is done with the DetachUmb pin. The problem is that the actuator and sensing subgroup didn’t have time to implement a whole analogue system that ensures a correct umbilical cord detachment. Therefore, until LZ decides to use this signal, DetachUmb won’t actuate anything. However, since it is likely LZ will use this signal in the future, it has already been implemented and documented in the code. By setting the pin high the umbilical cord can be detached.

The second check senses whether current and power flow from RDS to the rover’s power bus. This signal is measured right after the 28-to-12 V stepdown converters. A voltage divider further reduces this 12V signal to reach a maximum voltage of 3.2V. This is well below the maximum voltage that can be applied to any pin of Vcc + 0.3V (with an absolute maximum of 4.1V) [21]. It is desirable to sense the bus as an accurate voltage instead of a simple Boolean. It is possible to calculate the current flow with this measured voltage using a formula that is not provided yet by the sensing and actuating subgroup [8].

ADC Voltage Conversion

Analog to digital conversion is necessary to read the voltage on a pin. ADC is also necessary for other checks described in section 3.3.4. So, a modular function for this process will be made to save time and reuse code. Luckily, the MSP430FR5969SP features a dedicated ADC function for almost all its pins. To make use of the ADC function, it is important to understand the different initializations necessary:

- The whole ADC module of the MCU: The dedicated ADC sampling timer is chosen. The sampling takes 16 clock cycles, the conversion result is set to a 12-bit value, interrupts for overflow are enabled, and the ADC module is activated.
- The relevant pin: The pin is set as an input pin, and its ADC mode is selected by setting both selection bits to high.
- The ADC for the relevant pin: The pin’s conversion result is coupled to a memory register, and the ADC conversion complete interrupt is set high.

So the first initialization is done at the startup of the whole RDSS, the other two are pin specific. The last one is done within the ADC voltage function. The whole ADC voltage function is shown in diagram 3.9. The conversion works with polling. Whenever the interrupt flag for the memory register is high, the while loop will be exited, and the conversion result will be saved. Because polling is used, a timeout must be implemented to prevent stalling if the conversion fails. An interrupt handler is used for this timeout. For this project, it is sufficient to handle only the interrupt that occurs if the conversion time overflows. This happens when the conversion fails, resulting in an exit from the while loop. Due to the implementation of the interrupt handler, it is possible for the current project to be developed further using other interrupts. Currently, more interrupts do not seem suitable. This would only require extra enables and extra cases in the handler. If the conversion is successful, the voltage will be calculated with the following formula:

$$Voltage = \left(\frac{measuredADCValue}{MaxADCValue}\right) \cdot MaxVoltage \tag{3.1}$$

$$measuredADCValue = \text{value ranging from } 0 - 4095 \text{ (12-bit resolution)}$$

$$MaxADCValue = 2^{12} - 1 = 4095 \text{ (12-bit resolution)}$$

$$MaxVoltage \approx 3.6V\ (V_{cc} + 0.3V)$$

The function will return this voltage. If the conversion fails, the interrupt will be triggered and the function will return an error voltage of 99V.

In addition to a message containing the measurement of the voltage on the BusCurrentSense pin, it is possible to activate or deactivate the rover's charging with the BusFlag pin.

To conclude the connection section, the electronics control consists of two checks with possible actions. Now, for the actual implementation of the above-mentioned checks and actions, a differentiation can be made between the deployment stage and every other stage of the mission.

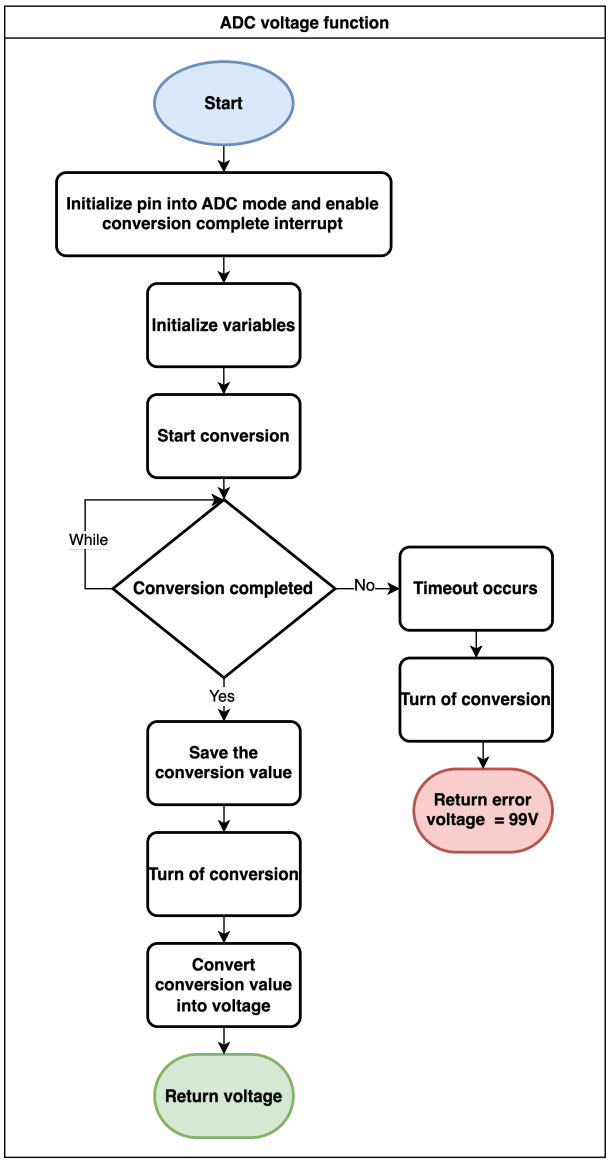

Figure 3.9: Flow diagram of the ADC voltage function

No Deployment

In every stage apart from the deployment, the control of the connections can be seen as a checkup, which reports to the lander and, therefore, to Earth. First, the umbilical cord connection is verified. If it is connected properly, a validation message will be sent. If the pin is low, either because the umbilical cord is not connected or the pin is defective, an error message will be sent. Before launch, this could be useful information and may result in a manual check of the umbilical cord connection. However, during the mission, not much can be done about this error message. The process must go on. Only the "detachUmb" signal would be put low again to ensure that this is not the problem.

The process continues with the voltage sensing on the bus. It will always send a message containing this voltage back to the lander, and if a timeout occurs, it will send an error message. Whatever the result of this measurement is, the charging is always activated. Only during deployment will a situation occur where you don't want to charge the rover. After this, the process continues to the temperature sensor and heater section.

Deployment

Most of the connection control during deployment is already covered in section 3.1.2. As mentioned, during the deployment also the possible actions of the connections control are utilized. First, the power to the rover is turned off by setting the Busflag pin to high. It could be checked if the rover actually stopped charging by sensing the voltage on the BusCurrentSense pin. However, since the umbilical cord can detach anyway, it is chosen not to do this and just to continue. This extra data could, however, be used in future implementations. Then, the umbilical cord to the rover will be disconnected with the "detachUmb" signal. If this had success, the "Read12VFromRover" pin would be low, and the process

would move on to the next section. If the disconnection didn't work and the "Read12VFromRover" pin would still be high, the detachment would be tried two more times. If there is still no success, the process will move on to the next section anyway.

3.3.3. Temperature sensors and Heater

For the temperature sensor and heater section of the control system, the involved pins are summarized in the following table 3.3.

Name	Pin	Direction	Active / Mode
TempOn1	Pin 3.5	Output	Active low
TempOn2	Pin 1.3	Output	Active low
Temp1Out	Pin 3.4	Input	Capture mode
Temp2Out	Pin 1.4	Input	Capture mode
MCUHeaterOn	Pin 3.6	Output	Active High
MCUHeaterOff	Pin 1.6	Output	Active High
ReadHeaterActive	Pin 3.7	Input	Active Low

Table 3.3: Temperature Sensors and Heater Pins

First, it is important to understand the temperature sensors. Two temperature sensors are used for the RDS. The function written to measure the temperature with one of these sensors is modular, so it is possible to implement more temperature sensors for the RDS if needed. The temperature sensors work via oscillation: the higher the temperature, the lower the frequency. To read the value of the temperature sensor, one has to measure the frequency. Due to the design of the temperature sensors by the actuator and sensing subgroup [8], the operating range is between 400 and 700 Hz to represent a temperature of -55 up to 80 degrees Celsius.

Frequency measurement

Frequency measurement is less straightforward than ADC conversion due to constraints by the chosen MCU: it lacks a dedicated frequency measurement module. Instead, the MCU is capable of storing the value of a timer with the rising edge on a pin. This is called the capture mode. To use the rising edge of the correct pin, caution is necessary when initialising the pins and connected timer:

- Relevant input pins (e.g. Temp1Out): The pin needs to be set in the timer function by setting the first selection bit to a 1 and the second to a 0.
- Relevant timer: The input clock is set to the sub main clock, it is in continuous mode, the input clock is divided by 4 and the timer is cleared.
- The timer capture/compare control of the pin: Capture on the rising edge, capture mode, and connection with the pin.

The relevant timer depends on the pins. In this project, the relevant timer is chosen to be the same for both temperature input pins: Timer B0. Furthermore, it is necessary to initialize the pins used with the right timer capture control register. This is pin-specific and can be determined with the use of the msp430fr5969 data sheet [21].

A modular function is made to determine the temperature of the different temperature sensors. This can be seen in 3.2 Now it is possible to implement this function even if pins with different registers are used. Only the registers that match the pin have to be known from the data.

```
1  /*
2   * Reads out the temperature sensor measurements.
3   *
4   * Parameters:
5   *   volatile unsigned int* TxxCCTLx : pointer to Timer control register
6   *   volatile unsigned int* TxxCCRx : pointer to Timer capture/compare register
7   *
8   * Returns:
9   *   float temperature: the measured temperature
10  */
11 float readout_temperature_sensor_n(volatile unsigned int* TxxCCTLx, volatile unsigned int*
       TxxCCRx);
```

Listing 3.2: read temperature sensor N function signature

The whole frequency measurement process is displayed with a functional flow diagram in Figure B.9 in appendix B. The actual measurement uses polling. It waits for a capture event. It stores the value of the capture register if the capture flag is high (at the rising edge of the measured pin). The capture flag is turned off. It waits for a capture event a second time. It stores the value of the capture register in another variable if the capture flag is high again. This way, the amount of clock cycles of timer B0 between two rising edges of the pin is measured. The problem is that timer B0 is only 16 bits large. This means that it can only count up to a value of $2^{16} = 65536$. It is desirable to measure a minimal frequency of 150 Hz. For simplicity, you don't want to keep track of the amount of times timer B0 overflows between two captures. So, the number of clock cycles measured at the lowest frequency must be lower than 65536. This means that timer B0 must run on a clock frequency lower than $150Hz \cdot 65536 = 9830400Hz$. The problem is that the whole RDSS works with a sub-main clock of 16 MHz. So, to ensure that the measurement works, the sub-main clock is divided by four as the input of timer B0. So timer B0 runs on 4 MHz.

Since polling is used twice for this measurement, a timeout is necessary for the system to continue even if the measurement fails or takes too long. For this timeout, a new timer (timer A2) is initialized. This timer has a maximum of 60000 clock cycles and works with the same pre-divided 4 MHz clock. This means that the interrupt handler will be triggered, and the process will exit the while loop if the measured frequency is below $4000000Hz/600000 \approx 66.67Hz$ or the measurement failed.

Because timer B0 is in continuous mode, the timer is able to overflow. If this happens, the counter will start counting again at 0. This means that if an overflow happens between the first capture and the second capture of the measurement, it would be possible for the second measurement to be smaller than the first. Larger is impossible since the timeout timer is chosen to be smaller than one full cycle (0 - 65536) of timer B0, and the smallest frequency of interest (150 Hz) results in a difference between the capture events of at most $4000000Hz/150Hz = 26666.67$ clock cycles. The following simplified diagram illustrates this example 3.10.

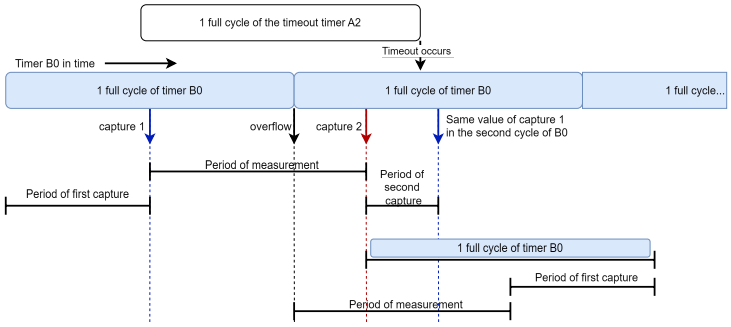

Figure 3.10: Finding the period of the measurement when overflow of timer B0 occurs

Now, it is always possible to calculate the period of one measurement. If the first capture value is smaller than the second capture value, the whole capture event happened in one full cycle of timer B0, and the period is calculated with the second capture value - the first capture value. If the first capture value is larger than the second capture value, the period is calculated as follows: The maximum value of timer B0 (65536) is added to the second capture value. Then, the first capture value is subtracted from this addition. This results in the period between the two capture events, even when timer B0 overflows. Once again, diagram 3.10 clarifies this calculation.

To make the measurement more accurate, the period is measured nine times. The nine results are saved in an array. The median will be taken as the most accurate period of the measurement. Whenever a timeout occurs, the frequency is set to one. This means that it should not be an issue if one or two periods were recorded wrongfully.

With the period between the two capture events (two consecutive rising edges on the pin), it is easy to calculate the signal's frequency on the pin. This is done with the following formula:

$$Frequency = \frac{Clk}{period} \tag{3.2}$$

$$Clk = 4MHz$$

If the calculated frequency falls within the predefined range of 150 Hz - 800 Hz, the temperature will be calculated in degrees of Celsius with the following formula:

This formula is based on the hardware implementation of the temperature sensors by the subgroup sensors and actuation [8].

$$x = \frac{1.0}{0.4055 \cdot 2.0 \cdot C \cdot Frequency} \tag{3.3}$$

$$Temperature = \frac{x - 1000.0}{3.85} \tag{3.4}$$
$$C = Capacitance = 2.2\mu F$$

If the frequency is outside the range, the function will produce an error temperature of -99. If a timeout occurs during at least half of the measurements, the result will equal a frequency of 4000000 Hz.

To summarize, to check the temperature of both temperature sensors, the following few steps are performed:

- Both temperature sensors are put on
- A short delay is implemented for the temperature sensors to reboot
- The period is measured for the first sensor
- This period is converted to a frequency and again to a temperature
- The period is measured for the first sensor
- This period is converted to a frequency and again to a temperature
- The temperature of both sensors is communicated to the lander
- Both temperature sensors are turned off again

Heater Control

The temperature on different parts of the RDS is needed to control the heater present on the RDS PCB. This heater is used to keep the temperature above the minimum temperature of -55°C (minimum operating temperature of the MCU [21]) at all costs. It is important to keep in mind that the sensing subgroup also made an analogue control system for the heater [8]. However, this system functions as a backup for extreme temperatures in which the MCU might fail. With the precise control options of the MCU, it is possible to use the heater in a much smaller range. To control the heater, a range is taken between 20°C and 40°C by the RDSS. For the two control pins, a truth table can be made:

MCUHeaterOn	MCUHeaterOff	Mode
0	0	Backup
0	1	Heater is off
1	0	Heater is on
1	1	Heater is on

Table 3.4: Truth table heater control pins

Backup mode means that the control of the heater is left to the analogue system. The heater control is displayed with the flow diagram in Figure B.10 in appendix B B.3.2. If only one of the temperature sensors works, only the readout of this sensor will be taken into consideration when determining if limits are exceeded. This is done via a switch statement with three cases:

- Both temperature sensors are working
- Only sensor 1 is working
- Only sensor 2 is working

If more temperature sensors are needed in future projects, the heater control algorithm can be expanded by adding more cases.

3.3.4. NEA Activation

The last section of the electronics components control system concerns the NEA activation. As can be seen in Figure 3.8, the NEA activation consists of the check-up of two different types of electrical components with their respective possible actions: the supercapacitors and the NEAs. The non-explosive actuators need a large amount of power to release. The lander cannot provide this amount of power at once, so supercapacitors are needed for the NEA activation. For the NEA activation section, there is a large difference between all the other stages of the mission and the deployment stage. If no deployment signal is received, it is necessary that none of the NEAs are activated. It is a short checkup to ensure that none of the NEAs were actuated. During deployment, a whole check-up will be done on every single component to ensure the correct activation of every single NEA. Once again, all the involved pins for this section are summarised in the following table3.5:

Name	Pin	Direction	Active/Mode
ChargeCapFlag1	Pin 2.7	Output	Active high
ChargeCapFlag2	Pin 2.3	Output	Active high
ChargeCapFlag3	Pin 3.4	Output	Active high
CapDischarge	Pin J.4	Output	Active high
CapReady	Pin 2.4	Input	ADC mode
NEAReady1	Pin 3.1	Input	General I/O mode
NEAReady2	Pin 3.2	Input	General I/O mode
NEAReady3	Pin 3.3	Input	General I/O mode
NEAReady4	Pin 4.7	Input	General I/O mode
NEAFLAG1	Pin 1.0	Output	Active High
NEAFLAG2	Pin 1.1	Output	Active High
NEAFLAG3	Pin 1.2	Output	Active High
NEAFLAG4	Pin 3.0	Output	Active High

Table 3.5: All NEA activation pins

No Deployment

When the rover is not being deployed, no actions are required. After every single check, a message with the outcome is sent to the rover. The check-up starts with voltage sensing on the CapReady pin. This should be 0V during no deployment stages. If it were to be different from 0V, the only thing RDSS can do is set all the ChargeCapFlags and the CapDischarge to low once again. The check-up continues by checking the status of every single NEA. If the NEAReady pin is high, the NEA is still in place and not activated yet. So, all the NEAReady pins should be high during every stage except deployment. Once again, if this isn't the case, there is not much RDSS can do about it.

Deployment

The general deployment process is already covered in section 3.1.2. However, there is one process in the functional flow diagram of the deployment mode (B.5) that deserves a broader explanation. The "Check functionality of each supercapacitor" is done with a certain algorithm. This algorithm was necessary because the RDS has three supercapacitors, which can act together as one voltage source. The problem with capacitors is that they can create a short in your circuit if they break. Therefore, an individual check is necessary for every supercapacitor right before the actuation of every NEA.

A functional flow diagram of this check is shown in Figure B.11 in section B.3.3 of appendix B. It is important to keep in mind that in the real implementation, timers are involved, as supercapacitors take time to charge. The subgroup power flow designs the integrated circuit of the supercapacitors, and the design can be found in their thesis. [19]. Figure B.12, shows the architecture of the electronic capacitor checkup system. All three capacitors can be charged individually, yet they are all connected to the same bus. The current through this bus can be restricted with the CapDischarge signal. If this signal is set to high, current can flow to the NEAs. The Capready flag is read behind the switch on the bus, actuated by the CapDischarge signal. Therefore, sensing the current on the entire bus is only possible after the CapDischarge signal is set high. So, to read the voltage provided by every individual supercapacitor, the algorithm depicted in Figure B.11 is needed.

After this supercapacitor check, only the capacitors that are still working properly will be used to activate the NEAs (Only the ChargeCapFlag signals will be set high before the CapDischarged is activated).

4

Testing Methodologies

4.1. Introduction

Multiple methodologies are used to test whether the RDSS works as expected. Different hardware setups are needed to measure results, and various software configurations are required to ensure proper initialization and correct integration of these subsystems. The hardware setup for different subsystems will be introduced. Secondly, the software needed to test these will be explained. Subsequently, the performance tests of these subsystems will be presented.

Continuous Improvement

The testing process is iterative. Issues identified during testing are addressed, and the tests are updated accordingly. This ensures that the system remains reliable even as new features are added, or existing ones are modified.

Following this structured testing approach will build a robust and reliable RDSS. Each method will be rigorously tested to ensure it meets the required functionality and reliability standards.

4.2. Hardware Setup and Configuration

With the aim of obtaining results which satisfy the requirements of the RDSS, a hardware setup is required for various tests. These different setups will be shown below.

Lander-RDS communication

Both parties' incoming and outgoing signals must be observed to ensure correct communication between the RDS and the lander. Since the UART protocol is used for transmitting data, a logic analyzer is suitable for viewing the messages transmitted on a computer. For all the communication tests, an eight-channel 24 MHz logic analyser was used in conjunction with the software Logic2 from Saleae [20]. In Figure 4.1, it can be seen that a white wire is used to connect the logic analyzer to GND. Two purple wires are used to read out the TX pins of the lander and the RDS. A green and red wire sends data from the lander MCU to the RDS MCU.

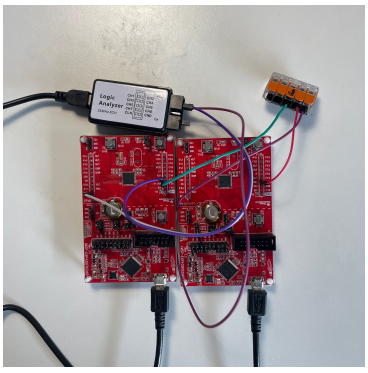

Figure 4.1: Logic analyzer connected to lander TX pin and RDS TX pin

Electronic Components Control System

Most of the attention and work for the ECCS has gone into designing and testing the two main functions: the ADC voltage conversion function and the frequency measurement function. The rest of the software concerns the bundling of the different check-up functions and setting pins high or low accordingly to actuate the right components. Although just as important this can only be tested through software testing. The workings of this will be explained in section 4.3. The two main functions of ECCS are tested with two different hardware setups. For the ADC testing, a test pin was connected to a DC voltage source 4.2. This way, different voltages can be measured. For the

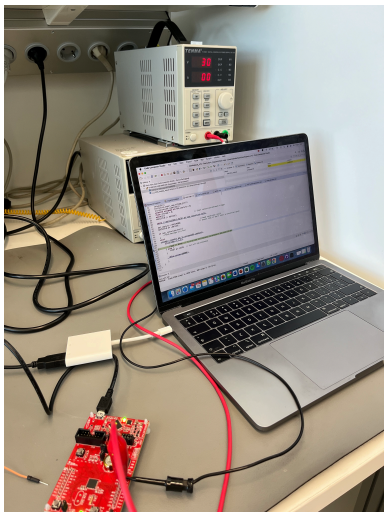

Figure 4.2: ADC testing with a DC voltage source

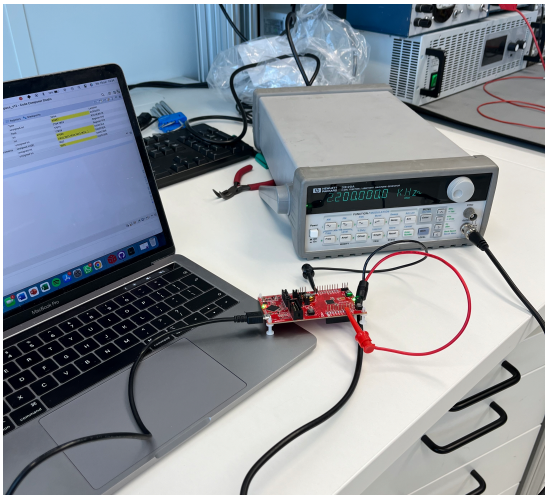

Figure 4.3: Frequency testing with a function generator

frequency measurement, a test pin was connected to a function generator 4.3. This function generator had the following settings to replicate the real temperature sensors:

- 3 V peak to peak voltage
- Square wave
- 1.5 V DC offset

This way, different frequencies can be measured.

4.3. Software Testing

In the interest of improving reliability, it is important to profoundly test software to mitigate errors. This can be done by testing each component separately using Google Unit Testing. These different components are also integrated into one system and tested to ensure correct interaction. Finally, system testing is applied to ensure the full system can operate as required.

4.3.1. Unit Testing

Unit testing is done using the Google Unit Testing library [6]. This has been proposed by the Lunar Zebro team and has proven to be effective in finding errors in their algorithms. A unit test consists of an assertion between an algorithm's expected value and the output value. Thus, some value is entered into the algorithm. Some value is expected from the algorithm, and this is compared with the actual output. This way, many cases can be tested to ensure the algorithm behaves as expected. Below is listing 4.1, an example of a Google Unit test is shown for the slip encoding algorithm.

```
TEST(slip_encoding_testsSuite, NormalDataTest){
    // input buffer with data
    uint8_t input_buffer[UART_BUFFER_SIZE] = "Hi";
    uint16_t input_length = 2;
    // output buffer after encoding
    uint8_t output_buffer[UART_BUFFER_SIZE];
    uint16_t output_length;
    bool result;

    // expected buffer
    uint8_t expected_buffer[UART_BUFFER_SIZE] = {0xc0, 0x48, 0x69, 0xc0};
    uint16_t expected_length = 4;
    bool expected_result = true;

    // encoding
    result = slip_encode(input_buffer, input_length, output_buffer, &output_length);

    // Assertions
    EXPECT_EQ(result, expected_result);
    EXPECT_EQ(output_length, 4);
```

```
21      for(int i = 0; i < expected_length; i++){
22          EXPECT_EQ(output_buffer[i], expected_buffer[i]);
23      }
24  }
```

Listing 4.1: Google Test Example for SLIP Encoding

The unit tests for the slip_encoding() method cover a wide range of scenarios to ensure its reliability and correctness. These scenarios include:

- Normal Operation: Ensuring that regular data bytes are decoded correctly.
- Edge Cases: Testing the handling of empty buffers, maximum buffer sizes, and sequential special characters.
- Special Characters: Verifying that the method correctly processes SLIP-specific escape sequences.
- Error Conditions: Checking the response to invalid inputs, such as buffer lengths smaller than expected or invalid start and end bytes.

4.3.2. Integration Testing

After individual algorithms are validated through unit tests, integration testing will be performed to ensure that the combined system functions correctly. This involves:

- System Integration: Combining the tested components into the complete communication system and verifying overall functionality.
- Interoperability Testing: Ensuring that different parts of the system communicate correctly with each other.
- End-to-End Testing: Simulating real-world scenarios to verify that the entire communication system works as intended from start to finish.

Integration testing consists of three different parts: the lander-RDS communication system will be tested as a whole. Furthermore, the NEA activation will be tested to ensure correct analogue voltage measurements on a pin. Finally, the temperature sensors will be tested to ensure a correct frequency measurement and proper conversion to a temperature.

4.3.3. System Testing

After successful integration testing, full system testing can be done to ensure a correct control loop design. This will be done in multiple different phases, where each transit mode is tested separately. Moreover, the transition from different transit modes also needs to be tested. Finally, a staged mission can be done where the different transit modes are being covered sequentially, and finally, the deployment needs to be tested very thoroughly by checking individual processes.

4.4. Performance Testing

Performance testing is used to determine the system's boundaries. It should help find data to better understand the system's limits. For example, the highest and lowest frequencies of the temperature sensors need to be determined through thorough testing. It might also help determine the minimum time needed for the system to process incoming data. Either load testing or stress testing will simulate these critical situations.

4.4.1. Load Testing

Load testing is used to Figure out how the system behaves under expected load conditions and to find the maximum operating capacity of a subsystem.

4.4.2. Stress Testing

Stress testing is used to determine how the system behaves under extreme load conditions and to find out where and how it fails.

5 Results

5.1. Introduction

The results will be used to find out whether the RDSS met all of its requirements. These requirements are needed to ensure that the RDS achieves its requirements. As mentioned before, the RDS has three main objectives: deploy the lunar Zebro rover on the surface of the moon, function as an intermediary for communication between the rover and the lander, and provide power from the lander to the rover during transit when needed. Firstly, the software testing results will be presented. Furthermore, the performance testing results will be shown. These results will be discussed in chapter 6.

5.2. Software Testing Results

A lot of software was written to create the RDSS. This software needs to be tested to ensure it is functional and reliable for the Lunar Zebro mission. The way this will be done has been explained in chapter 4. Now, the results of these various testing strategies will be shown. Unit testing results, integration testing results, and system testing results will be presented, respectively.

5.2.1. Unit Testing Results

As discussed in subsection 4.3.1, unit testing allows one algorithm to be tested thoroughly by injecting different inputs and comparing the outputs with the expected results.

lander-RDS communication

Four different algorithms were tested using Google Unit Testing for the lander-RDS communication: slip_encoding, slip_decoding, convert_message_to_array, and convert_array_to_message. A list of the tests can be found in appendix C.1 for each individual algorithm. The pseudocode of each algorithm can also be found in appendix A.1. Below in Figure 5.1, the tests' coverage is presented. The lander_communication.cpp file contains all four algorithms and has 100% line coverage and 100% branch coverage. The other values are unimportant since they cover functions that cannot be tested with unit testing due to using pins on the MCU. These can thus be disregarded.

During unit testing, several errors were identified and resolved. For instance, the slip_decoding algorithm initially did not return any value, leading to a failure in error handling. This issue was corrected by modifying the method to ensure it returns appropriate error codes, thereby enabling proper decoding. Furthermore, an error was spotted that would occur if a message was received with an ESC character but nothing following it. This is an error in the encoding software of the opposite party, or a bit flip could have occurred during transmission. This error now returns a false and can be dealt with. As well, there was no check for a minimum or maximum size of messages; this had to be implemented and has been tested to work. Slip_encoding also had this issue, which had to be solved and tested.

Both the convert_message_to_array and the convert_array_to_message method were changed multiple times throughout the development phase since the self-made communication protocol was first developed using a static buffer size for sending messages and subsequently converted to using a variable buffer size. Figure D.1 of such a transmission can be found in appendix D. Using a fixed payload size wasted a lot of time and carried useless data. This was thus converted to a variable payload size that

can be seen in Figure D.2. The first implementation contained vectors. However, these use dynamic memory allocation, which is not allowed according to the MISRA C 2012 guidelines [14].

Element	Line Coverage, %	Branch Coverage, %
TestingRepositoryBEP	70% files, 34% lines covered	35% branches covered
Google_tests	65% files, 33% lines covered	34% branches covered
lander_communication_lib	100% files, 100% lines covered	100% branches covered
lander_communication.cpp	100% lines covered	100% branches covered
lander_communication_protocol.cpp	100% lines covered	100% branches covered
electronics_components_control_system_lib	100% files, 100% lines covered	100% branches covered
temp_sensors.cpp	100% lines covered	100% branches covered
supercap_readout.cpp	100% lines covered	100% branches covered

Figure 5.1: Google Unit Tests Coverage

Electronics Components Control System

The electronics components control system consists of multiple different subsystems. These subsystems work with many assigned pins, hindering the code from being unit-tested. However, three algorithms were used which could be unit-tested. These include: convert_adc_to_voltage, calculate_frequency, and frequency_to_temperature. Figure 5.1 shows the line and branch coverage of the three algorithms by their respective files. Through unit testing, an error was spotted where a frequency of zero degrees could result in a 1 by 0 division. This was removed by inserting a frequency boundary check at the start of the method. It now returns a value of -99 degrees Celsius since this is a temperature that the sensor cannot measure, and it is a temperature at which the MCU cannot operate [21].

5.2.2. Integration Testing Results

After testing multiple algorithms, it is important to integrate these different functions together and test each subsystem independently from each others. This will be done during the integration testing results. After successful integration testing results, system testing can be started.

lander-RDS communication

The lander-RDS communication subsystem started with sending one character via the UART output pin 2.5 and was developed into a rather large communication system with multiple functionalities. Figures D.12, D.13, D.14, and D.15 show the evolution of the lander-RDS communication protocol. Image D.12 shows the first character being sent. Figure D.13 displays the first retransmitted message; here, the receiving interrupt handler was correctly configured and could store a message and retransmit it. Figure D.14 shows the first variable payload message sent. However, this was done using vectors and later had to be converted from vectors to static memory as explained in chapter 3.2.2. Figure D.15 shows an error detected in the receiving mechanism, resulting in only transmitting one acknowledgement every two messages. This was solved by implementing the receiving protocol explained in chapter 3.2.3.

Electronic Components Control System

The two main functions of the electronic components control system are the ADC voltage conversion function and the frequency measurement function. However, the LED light was sometimes used during testing to ensure the function worked. Most of the time, the debug mode of the Code Composer Studio was used [2]. With this debug mode, the values of certain internal variables could be seen. This enabled an exact frequency readout without depending on a light within a certain range.

In the end, the ECCS function was made (as shown in Figure 3.8). It was tested and proved to work. However, the main two functions for executing these checks require additional attention.

ADC voltage conversion

As mentioned in 4.2 the ADC function is tested by connecting a test pin to a DC voltage source. After multiple tests and different iterations, a function was created to read the voltage of a chosen pin. Figure 5.2 shows the ADC value that is read out by the software in debug mode while a certain voltage is applied to the tested pin.

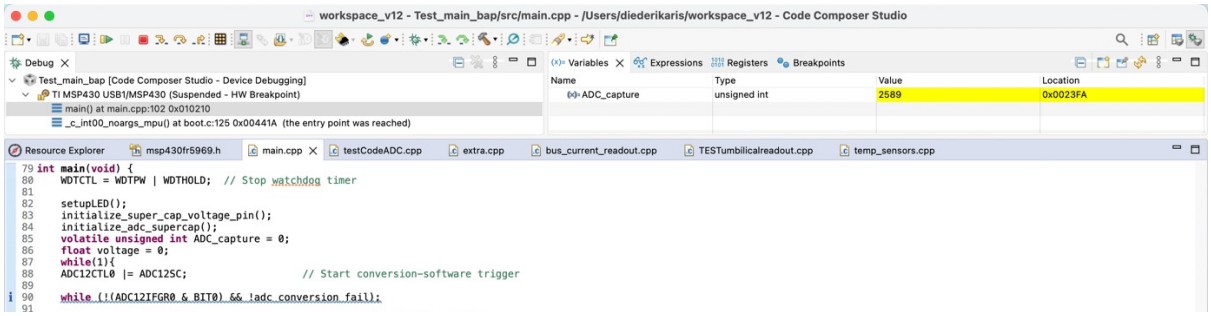

Figure 5.2: ADC conversion function results in debug mode

Given the chosen resolution of 12-bits and a maximum sensible voltage of 3.6V. It would follow that the function should be able to measure the voltage on a pin with a resolution of: $\frac{3.6V}{4095} \approx 8.9 \cdot 10^{-4}V$. However, in reality, the achieved resolution of the function is only $0.02V$. It also turned out that the maximum voltage, which coincides with an ADC value of 4095, is actually 3.64V. This was taken into account in the final ADC conversion function.

Frequency Measurement

The Frequency measurement function is tested by connecting a test pin to a function generator, as mentioned in 4.2. First, a system which worked on an interrupt of the capture flag was tried to be implemented. The handler would be activated every time a capture event happened (a rising edge of the measured pin). After two capture events the interrupt would be disabled. The problem with this implementation of measuring the period was in the handler. It took the MCU way too much time to handle the capture events and set the flag low. This resulted in a mismatch between the second capture event and the actual consecutive rising edge on the pin. This caused a too-large measured period and a too-small measured frequency.

For this reason, the implementation was made using polling as described in much more detail in section 3.3.3. After multiple tests, a final function was made, which can accurately determine a pin's frequency. Figure 5.3 shows the determined frequency and the multiple measured periods in debug mode when a certain frequency is applied to the test pin.

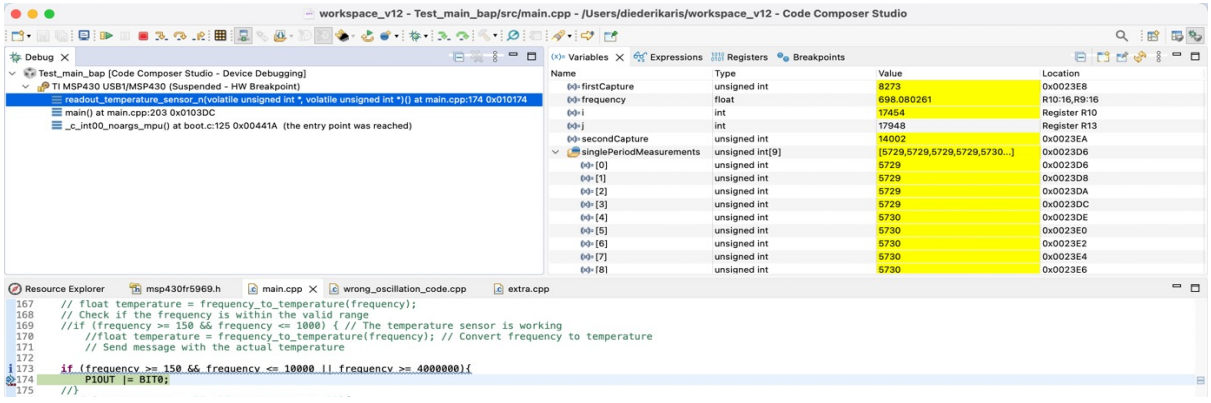

Figure 5.3: Frequency measurement function results in debug mode

During testing, one more noticeable change to the function was made. It was originally made to take the average of the 9 measurements instead of the median. However, it was discovered that occasional misreads of the period happened less than half of the time, yet they happened frequently enough and were large enough to weigh heavily on the average. Therefore, the median of the 9 measurements was chosen as the true period for calculating the frequency. This proved to be accurate a 100% of the time within the right range during testing.

After testing, there were a few interesting results: With the described initialization of timer B0 in 3.3.3, the range between which the measurement was accurate was between 70 Hz and 2.2 kHz. The lower bound can be explained by the fact that the function was designed so that it would not work if the measured

period was larger than the full register of timer B0. The lower bound can, however, be modified by changing the divider for the input clock of timer B0. If the frequency of timer B0 was changed in this way to 2 MHz, it would be possible to measure frequencies below 50 Hz. The upper bound deserves more research but can probably be explained by processing time. Exiting the while loop, saving the value of the register and resetting the capture flag takes time. If the frequency on the pin is too high, the consecutive rising edge could already have happened before the MCU had time for the next capture event. Resulting in a longer and false period. This effect could be observed if a frequency of just over 2.2 kHz was chosen. A frequency of 3 kHz on the pin gave a readout of 1.5 kHz because the MCU missed one rising edge between the first and second capture.

The last result was rather strange. The measurement always failed if the frequency on the pin was between 100Hz and 130Hz. More research is needed to explain this phenomena.

To counter these problems with the frequency measurement, an operational range was chosen between 150 Hz and 800 Hz. Converted to temperature with the formula 3.3, this means a range between -77.77°C and 710.78°C. This is already far beyond the operating range of the MCU. However, it is important to know the possible range for future work.

5.2.3. System Testing Results

To meet the system requirements, the full system needs to be tested, and this can be done by testing individual stages and a full sequence of multiple stages. This subsection will focus on each stage individually and then run multiple sequences after each other.

General startup stage

The general startup sequence is the stage that is always started whenever the MCU boots up. It then requests which stage it needs to go to and then continues running in the replied stage. Below in Figure 5.4, the startup process can be observed and works as intended. First, the transit stage is sent by the microcontroller upon booting, then it tries to initialise and sets a timer; once the timer is done, it checks for an acknowledgement; there is one received, and thus it requests the transit mode through a TRANSIT_MODE request message. It continues without checking for an acknowledgement and checks if the umbilical cord is connected; due to us testing it on the development board, we did not connect a power source to the umbilical cord pin, and thus, as expected, it returned a message saying that the umbilical cord is not connected. It then continues and performs the Electronic component checkup. Furthermore, it received an interrupt from the acknowledgement and thus changed the state from general startup to transit. However, for this test, the interrupt was not enabled to check if the full electronic components checkup would run correctly. A zoomed-in picture of the first part can be found in appendix D, Figure D.5. Likewise, Figure D.6 displays the second part of the image, and the last part can be found in Figure D.7, D.8, D.9, D.10, and D.11.

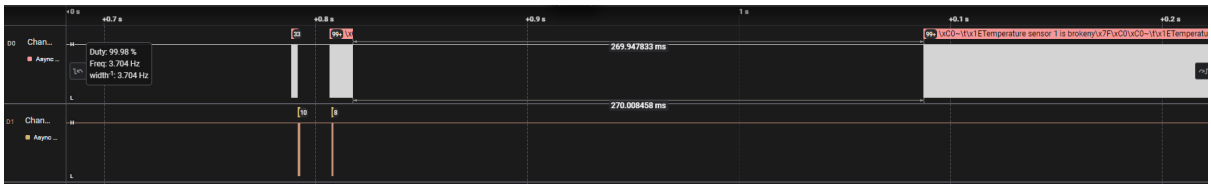

Figure 5.4: General startup sequence test

5.3. Performance Testing Results

5.3.1. Load Testing Results

lander-RDS communication

As mentioned in chapter 4, load testing will be done to see how the system behaves under normal load conditions. The following tests were done to test this: sending the largest possible message and checking the overhead bytes. Finding the accuracy of acknowledgements send on incoming initialisation messages. Test different message types and make sure the messages are handled as expected.

As shown in Figure 5.5, it can be seen that a message is transmitted with 256 characters. The image depicts this as ”256 results” since it found 256 characters. The first four bytes are the overhead from the message and are correct. The first byte corresponds to the END encoding bit from the SLIP protocol

as mentioned in chapter 3.2.2. The second byte corresponds to the correct start byte of the self-made communication protocol and the third and fourth byte corresponds to the correct message type and length of the payload. Which shows that a message can be sent using the maximum payload size.

Figure 5.5: Maximum message size of 256 bytes with correct overhead

To test the reliability of message responses sent, a test was made to send a message with a payload containing its message number. Subsequently, if the other microcontroller is able to respond, it sends an acknowledgement with the acknowledgement number. This shows at the end of 256 transmissions how many acknowledgements were made. This resulted in a 100% pass rate.

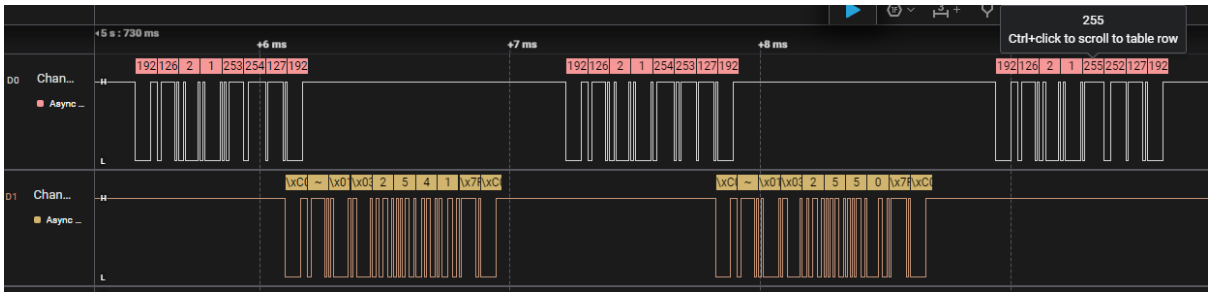

Figure 5.6: 256 acknowledgements on 256 incoming initialisation messages

5.3.2. Stress Testing Results

Stress testing is used to find the limits of the system. Different methods were used to determine whether the system performs well and what different boundaries are, such as frequencies for recording a correct temperature and the time the CPU needs to process incoming messages.

Lander-RDS communication

Multiple tests were performed to find the boundaries of the lander-RDS communication system, ranging from finding the speed limit to system recovery and determining what happens when the system is flooded with messages.

Below in image 5.7, an interrupt overflow is shown. This happened when the transmission rate was set to the maximum. A time of 141.8 μ seconds was observed between each message, and this can be seen in Figure D.3 in appendix D. The system is able to process four acknowledgements consistently. However, after multiple resets, four acknowledgements were always maintained. This could be due to interrupt nesting. Since so many interrupts are triggered, the system can only operate until too many interrupts are called and the MCU gets stuck.

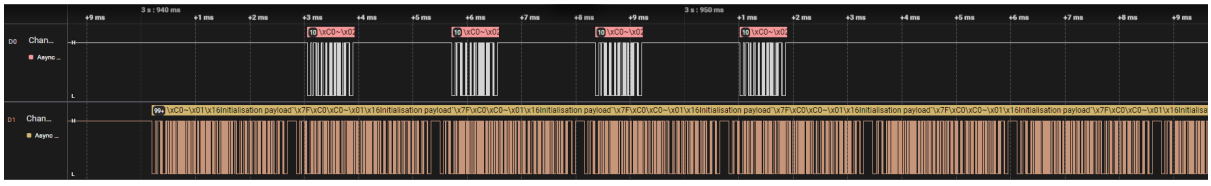

Figure 5.7: Interrupt overflow occurring when sending messages without any delay

Figure 5.8 shows the minimum time needed for correctly handling messages. Once the time between messages exceeded 238.125 μ seconds, the MCU could always respond. This is an important measurement to share with the lander organisation to ensure reliable communication. A 20% buffer could be added to prevent the boundary from being crossed.

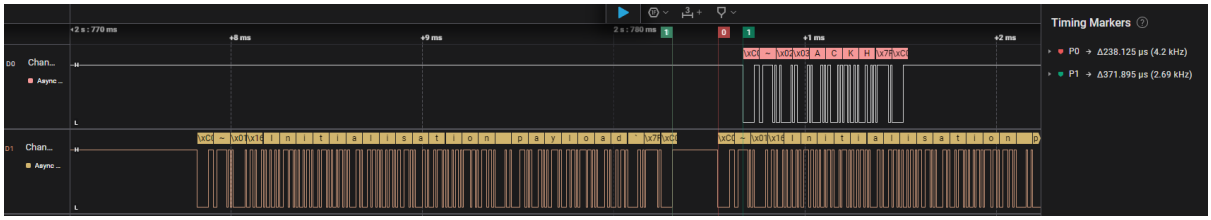

Figure 5.8: The minimum time needed between messages to ensure correct handling

During testing, an exciting result occurred. The software was able to consistently recover from receiving a message without a start byte until it the receiving buffer was fully filled, then it started processing information normally again. Figure 5.9 shows this, after approximately 50 milliseconds, the system responded only with acknowledgements and removed all the invalid messages from the receiving buffer. In appendix D, Figure D.4 shows a zoomed-in picture, where it is clear that the software finds one invalid message and one good where it acknowledges it. it continues doing this until all invalid messages are removed from the receiving buffer, and it does not have to notify about its existence anymore.

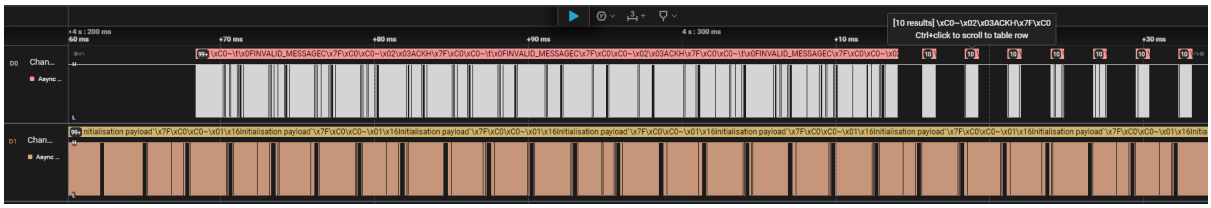

Figure 5.9: System recovery when starting to receive messages halfway

6

Discussion of the results

The discussion will cover the presented results from chapter 5 and reflect on how they can be used to improve the system further. It will also give insight into how future development of similar systems can be done more efficiently and result in better designs. It is important to reflect on the design choices and determine whether better solutions exist. This chapter will first discuss the results. Secondly, it will look at whether the requirements have been met and what could possibly be improved to satisfy the requirements that have not been met. Using all this information, future work can be determined.

6.1. Analysis of Findings

The results of the electronic components checkup system, the lander-RDS communication and the full system will be discussed. Problems will be addressed, and possible solutions will be provided. Also, good implementations will be highlighted and shown to be reliable.

6.1.1. Integrated System

Only the general startup mode has been implemented until now, and there have not been a lot of tests regarding its functionality. One test was done where the startup sequence was run, and as mentioned in chapter 5.2.3, it proved to work. However, none of the sensors were connected; thus, everything resulted in an error message. But the code performed as required. Further testing will be done, and more transit modes will be implemented later this week.

6.1.2. ECCS

The different functions of the ECCS worked, and it was possible to make a function for the full check-up. This function was able to perform all the different checks sequentially and send a status update to the lander after every check. However, due to time constraints, the whole ECCS is not yet tested for possible errors.

The problem with these checks is that they don't differentiate between the different types of errors that may occur. This is enough for the RDS in its current form, where the analogue system around the MCU is designed in such a way that the MCU can't be the single point of failure when actuating the electronic components. However, for future projects it could be interesting to know the type of error. As of now, the temperature readout function only provides an error temperature of -99, if the frequency measurement fails. However the reason can be a timeout or a measured frequency out of range.

The same holds for the ADC voltage conversion function. If the conversion takes too long, it will put out an error voltage of 99V. This is only one of the flags that can be handled by the dedicated interrupt handler for the ADC Module. It is also possible to enable an interrupt that senses an overflow of the ADC conversion register. This would happen if the voltage on a pin is higher than 3.64V. An extra case in the handler can be written to deal with this kind of error.

6.1.3. Lander-RDS communication

In chapter 5, multiple different tests were executed to ensure that the lander-RDS system worked as needed. Some of these tests showed the reliability qualities of the communication system. However,

not enough tests have been done to state that the RDS system is ready for a space mission.

Chapter 5.3.2 showed some stress tests, and the minimum time between received messages was found. This ended up being 238.125 μ seconds. This is extremely important to note since the lander software needs to take this into consideration. The Astrobotics lander uses a 32-bit high-performance dual-core LEON 3 FT microprocessor, which is rated for a clock frequency up to 125MHz [1] [13]. This means the lander could send messages faster than 238.125 μ seconds.

When looking at the reliability of the lander communication system, it performed quite well. It could reliably acknowledge 100 % of the incoming messages if the time between transmissions was larger than 240 μ seconds. However, this has not been tested thoroughly enough. More tests should be done for a longer period of time, and maybe longer and shorter messages should be sent, or messages every hour. This will result in a much better understanding of whether the system will be reliable enough for a space mission.

Another issue found while testing the lander communication was the receiving strategy. The MCU now handles interrupts and saves the incoming messages in a buffer. This buffer can hold a maximum of 256 bytes. While the electronic component control system is doing a temperature measurement, too many messages can be received and thus not handled, resulting in unanswered messages. Moreover, the receiving buffer is currently processed using a function that checks whether the uart state is equal to the received and the end and start index are not equal. However, if multiple messages are received. then this could lead to problems. Thus, more testing is needed to verify if the system can handle this and further improve the receiving strategy.

The messages are currently processed by running the process_RX_buffer() method after each electronic checkup process. This, however, is not a good implementation since it could result in further development not applying this strategy correctly. A possible solution for this problem could be sequential threading since the MCU only has one core. Additionally, cooperative multi-tasking could also serve as a solution. This way, a case statement is used during a certain transit mode to run each sub function. Every time a case is completed, a flag is raised, and the RX buffer is checked. After all the cases are completed. It resets this process and reiterates.

Currently, there is no implementation for all the message types. This is because there is little information about the Lander communication protocol. They have been created because they seem suitable for future development.

More planning and designing had to be done to improve the efficiency of this design and build a more robust system. There was a basic layout of how the system would be built, and edge cases were thought of. However, there were many more mistakes that were not foreseen and could have been found before starting the coding process.

Finally, the lander-RDS communication system still needs considerable improvement in terms of reliability. Many tests are still needed, and more information about the lander is needed.

To finalise, the system design consists of too many edge cases to be tested within 9 weeks. This should have been observed faster. However, a lot has been learned about system design and understanding the time needed to implement and test a simple system. A small system needs to be created and tested using software and subsequently tested using a real MCU for all of its edge cases. This can be very time-consuming and should not be underestimated.

6.2. Comparison with Requirements

To compare the requirements with the actual capabilities of the system, a table was constructed to show which requirements were met and which were not. Table 6.1, shows the functional and non-functional requirements of the RDSS. Each requirement that was not met will be discussed, and the ones that were and are quite important shall also be briefly discussed.

Functional requirements		Non-functional Requirements	
The RDSS should be an autonomous system that is able to recover from a power failure	✓	The software must be able to operate in extreme space conditions	✗
The system should act on every single incoming message from the lander	✓	The RDSS should prioritize reliability over performance	✓
The system should provide a reliable way of deploying the LZ rover onto the Lunar surface	✓	The Misra C 2012 guidelines must be followed for software design	✓
The RDSS should provide an algorithm for temperature control to survive the harsh environment of space	✓	The code must be compiled and tested using Code Composer Studio from Texas Instruments	✓
The RDSS should be a control loop system that is able to recover from every error, therefore operating autonomously	✗	Code should be tested using Google Unit Tests	✓
The system should never deploy the rover unless it gets a deployment signal from the lander	✗	Each algorithm needs to have 100% branch/line coverage on its unit tests	✓
The RDSS should function as a data relay system between the Rover and Lander	✗	Software should be programmed in a modular manner	✓
		The UART output/input pins 2.5 and 2.6 should be used for the RDS-lander communication protocol	✓
		The UART output/input pins 2.0 and 2.1 should be used for the RDS-rover communication protocol	✓
		MSP430FR5969SP MCU must be used	✓
		The msp-exp430fr5969 development board must be used for testing	✓
		The software must be written in either C or C++	✓

Table 6.1: Comparison of Full System Requirements and RDSS System Requirements

The RDSS should be a control loop system that is able to recover from every error is a requirement that has not been met. The reason for marking this one as not met is due to the fact that not enough tests have been done. It cannot be assured that the system is able to recover from every error. In theory, this should be the case. However, when it comes to an embedded system, many more factors play a role in determining whether the system is able to recover. The same holds for the requirement, ”The system should never deploy the rover unless it gets a deployment signal from the lander”. This can also not be assured since not enough testing has been done.

The last functional requirement that has not been met is that the RDS should be able to relay data from the rover to the lander and vice versa. This is, in theory, worked out. However, the implementation was much more complex than predicted and more work than expected. An agreement with the LZ team has been made that it would be wiser for their embedded system engineers to implement this since they already have the code ready, and they also need to update their code to add the RDS as a new subsystem (slave) to their bus.

A requirement that has been met is the reliable way of deploying the LZ rover onto the lunar surface. The theoretical description of the procedure has no loopholes and uses every single feature that the MCU can use to try and deploy the rover. The implementation could be tested better, however, many other factors apart from the software implementation play a role in its reliability.

The RDSS acts on every single incoming message using an interrupt handler; it then handles the message when it has time. This, in theory, fulfils this requirement. However, the RDSS’s receiving strategy could still be improved.

Almost all non-functional requirements have been met except ”the software must be able to operate in extreme space conditions”. This has not been tested and can thus not be assured. Various environmental tests could be performed. However, due to time constraints, this was not possible during the time given for making this project.

6.3. Future Work

The RDSS works, but it has not met all of its requirements and still needs improvements in certain subsystems. First, the Lunar Zebro team still needs to implement the rover communication subsystem. Second, as mentioned earlier in the lander communication discussion, the receiving strategy could be improved by either using single-core threading or cooperative multitasking. This would significantly improve the reliability of the communication system.

Furthermore, many more tests must be done to determine whether the RDSS is reliable enough for the LZ mission. These tests consist of integration, system, and performance testing. It is also important to

perform environmental tests on the MCU and find out how the radiation affects the transmission of bytes through the differential RS-485 and RS-422 hardware protocols.

The electronic component control system still needs to be tested more thoroughly to find out whether the maximum sensible frequency of the temperature sensors (2 kHz) can be increased. This increase in range makes it possible for a lighter capacitor to be used on the PCB. This could reduce the cost of sending the rover to space. A few grams could decrease the cost by a few thousand dollars. Additionally, the analogue-to-digital conversion function should also be tested more since it sometimes has readings of voltages fluctuating around 1.0V on the development board without being connected. Lastly, better error handling could be implemented, to differentiate between the possible errors.

7

Conclusion

The Rover Deployment Software System (RDSS) developed in this project serves as a pivotal component for the Lunar Zebro mission. Through detailed design, rigorous testing, and extensive validation, the RDSS has been developed to meet the strict requirements of space deployment. The system successfully integrates three primary subsystems: lander-RDS communication, electronic components control system, and support for the rover communication system.

Key achievements of the RDSS include its communication capabilities between the lander and the rover, validated through unit and integration tests under various conditions, ensuring reliable deployment and operations on the lunar surface. The electronic components control system effectively manages power supply and environmental monitoring via precise ADC voltage conversions and frequency measurements, guaranteeing accurate execution of the deployment sequence in harsh space conditions. Designed for autonomous operation, the RDSS can recover from power failures and handle most unexpected errors without human intervention, a crucial feature for the rover's remote lunar mission. Additionally, load and stress testing confirmed that the RDSS can manage expected operational loads and system boundaries, instilling confidence in its reliability. However, this could still be improved by conducting more tests.

Several areas require further improvement and testing: the RDSS has not yet undergone comprehensive testing under extreme space conditions due to time constraints, necessitating future work focused on environmental tests to ensure reliability in the lunar environment. Additionally, the communication system's receiving strategy could be enhanced through single-core threading or cooperative multitasking for improved reliability. The complete implementation of the rover communication subsystem, pending and to be carried out by the Lunar Zebro team, is also critical for relaying data between the rover and the lander and will finalize the RDSS functionality.

This study has been crucial for the immediate goal of supporting the Lunar Zebro mission and holds broader implications for future space missions. By developing a high-level design decomposed into various subsystems and refining design choices in collaboration with the Lunar Zebro team, we have created a high-quality, functional, and adaptable system. The insights and methodologies developed in this research are expected to benefit future projects and significantly contribute to the broader field of embedded systems for space deployment devices. This work not only assists the Lunar Zebro team in building an electronic system for their Rover Deployment System (RDS) but also provides a valuable foundation for future teams developing similar deployment mechanisms.

In conclusion, the RDSS has made significant strides in ensuring the successful deployment of the Lunar Zebro rover. While some aspects require additional development and testing, the current system provides a solid foundation for future enhancements and mission success

To find the code of this project, use the following url: https://github.com/Hvanhuynegem/BAP-Rover-Deployment-Software-System [5].

References

[1] *ASTROBOTIC LUNAR LANDERS Payload User's Guide*. Version 5.0. Astrobotic. Aug. 2021. URL: https://www.astrobotic.com/wp-content/uploads/2022/01/PUGLanders_011222.pdf.

[2] *CCSTUDIO IDE, configuration, compiler or debugger | TI.com — ti.com*. https://www.ti.com/tool/CCSTUDIO. [Accessed 21-06-2024].

[3] Firefly Aerospace, Inc. *Alpha Payload User's Guide*. Version 4.0. Accessed: 24/04/2024. 2023. URL: https://fireflyspace.com/wp-content/uploads/2023/08/Alpha-PUG-4.0.pdf.

[4] Forum Standaardisatie. *UDP*. Accessed: 2024-06-13. 2024. URL: https://www.forumstandaardisatie.nl/open-standaarden/udp#_Detailinformatie_.

[5] *GitHub - Hvanhuynegem/BAP-Rover-Deployment-Software-System — github.com*. https://github.com/Hvanhuynegem/BAP-Rover-Deployment-Software-System. [Accessed 21-06-2024].

[6] Google. *GoogleTest: Google Testing and Mocking Framework*. Accessed: 2024-06-14. 2024. URL: https://google.github.io/googletest/.

[7] ISISpace. *ISIPOD CubeSat Deployer*. Accessed: 2024-06-13. 2024. URL: https://www.isispace.nl/product/isipod-cubesat-deployer/.

[8] N. Kant and T. J. Velzel. *Sensing and Actuation: Lunar Zebro Rover Deployment System*. Thesis. Technical University Delft, 2024.

[9] Charles M. Kozierok. *Serial Line Internet Protocol (SLIP)*. Accessed: 2024-05-31. 2005. URL: http://www.tcpipguide.com/free/t_SerialLineInternetProtocolSLIP-2.htm.

[10] Lunar Zebro | TU Delft. *Lunar Zebro website*. [Accessed 12-05-2024]. URL: https://zebro.space/.

[11] Ramesh B. Malla and Kevin M. Brown. "Determination of temperature variation on lunar surface and subsurface for habitat analysis and design". In: *Acta Astronautica* 107 (2015), pp. 196–207. ISSN: 0094-5765. DOI: https://doi.org/10.1016/j.actaastro.2014.10.038. URL: https://www.sciencedirect.com/science/article/pii/S0094576514004160.

[12] Lockheed Martin. *McCandless Lunar Lander User Guide*. Accessed: 24/04/2024. 2023. URL: https://cdn2.hubspot.net/hubfs/517792/Space/McCandless_Lander_User_Guide_Release1.pdf.

[13] *Microprocessors — esa.int*. https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Microprocessors. [Accessed 20-06-2024].

[14] MISRA. *MISRA C:2012 Guidelines for the use of the C language in critical systems*. Third Edition. Accessed: 24/04/2024. Watling Street, Nuneaton, Warwickshire CV10 0TU, UK, Mar. 2013. URL: https://www.diva-portal.org/smash/record.jsf?dswid=3586&pid=diva2%3A1290254.

[15] NASA. *What Are SmallSats and CubeSats?* Accessed: 2024-06-13. 2024. URL: https://www.nasa.gov/what-are-smallsats-and-cubesats/.

[16] *Nonstandard for transmission of IP datagrams over serial lines: SLIP*. RFC 1055. June 1988. DOI: 10.17487/RFC1055. URL: https://www.rfc-editor.org/info/rfc1055.

[17] *Payload user's guide*. Version 2.0. ispace. Jan. 2020.

[18] Eric Peña and Mary Grace Legaspi. "Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter". In: *Visit Analog* 54.4 (2020), pp. 1–5.

[19] Sanjeev J H Ramhit and Brecht Goethals. *Power System: Lunar Zebro Rover Deployment System*. Tech. rep. TU Delft, 2024.

[20] Saleae. *Downloads*. Accessed: 2024-06-14. 2024. URL: https://www.saleae.com/pages/downloads.

[21] Texas Instruments. *MSP430FR5969-SP Mixed-Signal Microcontroller*. Accessed: 23/04/2024. 2023. URL: https://www.ti.com/lit/ds/symlink/msp430fr5969-sp.pdf.

[22] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[23] B. Wang et al. “Space deployable mechanics: A review of structures and smart driving”. In: *Materials & Design* 237 (2023), p. 112557. DOI: 10.1016/j.matdes.2023.112557. URL: https://doi.org/10.1016/j.matdes.2023.112557.

[24] World Economic Forum. *Space: The $1.8 Trillion Opportunity for Global Economic Growth*. Accessed: 2024-06-13. 2023. URL: https://www.weforum.org/publications/space-the-1-8-trillion-opportunity-for-global-economic-growth/.

A

Source Code

A.1. Lander - RDS communication

Algorithm 1 SLIP Encoding

```
1: Input: buffer, length, UART_BUFFER_SIZE
2: Output: output_buffer, received_length
3: Constants: END = 0xC0, ESC = 0xDB, ESC_END = 0xDC, ESC_ESC = 0xDD
4: index ← 0
5: output_buffer[index] ← END
6: index ← index + 1
7: for i ← 0 to length - 1 do
8:     if buffer[i] = END then
9:         if index + 2 ≥ UART_BUFFER_SIZE then
10:            return false
11:        end if
12:        output_buffer[index] ← ESC
13:        index ← index + 1
14:        output_buffer[index] ← ESC_END
15:        index ← index + 1
16:    else if buffer[i] = ESC then
17:        if index + 2 ≥ UART_BUFFER_SIZE then
18:            return false
19:        end if
20:        output_buffer[index] ← ESC
21:        index ← index + 1
22:        output_buffer[index] ← ESC_ESC
23:        index ← index + 1
24:    else
25:        if index ≥ UART_BUFFER_SIZE - 1 then
26:            return false
27:        end if
28:        output_buffer[index] ← buffer[i]
29:        index ← index + 1
30:    end if
31: end for
32: if index ≥ UART_BUFFER_SIZE then
33:    return false
34: end if
35: output_buffer[index] ← END
36: index ← index + 1
37: received_length ← index
38: return true
```

Algorithm 2 SLIP Decoding

```
1: Input: input_buffer, input_length, UART_BUFFER_SIZE
2: Output: output_buffer, output_length
3: Constants: END = 0xC0, ESC = 0xDB, ESC_END = 0xDC, ESC_ESC = 0xDD
4: Check: if input_length < 2 or input_buffer[0] ≠ END or input_buffer[input_length - 1] ≠ END return false
5: Check: if input_length > UART_BUFFER_SIZE return false
6: is_escaped ← false
7: output_length ← 0
8: for i ← 1 to input_length - 2 do
9:     c ← input_buffer[i]
10:    if is_escaped then
11:        if c = ESC_END then
12:            c ← END
13:        else if c = ESC_ESC then
14:            c ← ESC
15:        end if
16:        is_escaped ← false
17:    else if c = ESC then
18:        is_escaped ← true
19:        continue
20:    end if
21:    output_buffer[output_length] ← c
22:    output_length ← output_length + 1
23: end for
24: return true
```

Algorithm 3 Convert Message to Array (Serialisation)

```
1: Input: msg, MAX_PAYLOAD_SIZE
2: Output: buffer, length
3: index ← 0
4: Copy start_byte to buffer[index]
5: index ← index + 1
6: Copy msg_type to buffer[index]
7: index ← index + 1
8: if msg.length > MAX_PAYLOAD_SIZE then
9:     Set buffer[index] ← MAX_PAYLOAD_SIZE
10:    index ← index + 1
11:    Copy msg.payload (up to MAX_PAYLOAD_SIZE) to buffer starting from buffer[index]
12:    index ← index + MAX_PAYLOAD_SIZE
13: else
14:    Set buffer[index] ← msg.length
15:    index ← index + 1
16:    Copy msg.payload (up to msg.length) to buffer starting from buffer[index]
17:    index ← index + msg.length
18: end if
19: Copy checksum to buffer[index]
20: index ← index + 1
21: Copy end_byte to buffer[index]
22: index ← index + 1
23: Set length ← index
```

Algorithm 4 Convert Array to Message (Deserialisation)

Input: buffer, length, MAX_PAYLOAD_SIZE
Output: msg
if length $<$ 5 **then**
    return false
end if
index $\leftarrow 0$
Extract start_byte from buffer[index]
msg.start_byte $\leftarrow$ buffer[index]
index $\leftarrow$ index + 1
Extract msg_type from buffer[index]
msg.msg_type $\leftarrow$ buffer[index]
index $\leftarrow$ index + 1
Extract temp_length from buffer[index]
temp_length $\leftarrow$ buffer[index]
index $\leftarrow$ index + 1
if temp_length $>$ MAX_PAYLOAD_SIZE **then**
    temp_length $\leftarrow$ buffer[index]
    index $\leftarrow$ index + 1
    msg.length $\leftarrow$ MAX_PAYLOAD_SIZE
    Extract payload from buffer starting from buffer[index] (up to MAX_PAYLOAD_SIZE)
    Copy buffer[index] to msg.payload (up to MAX_PAYLOAD_SIZE)
    index $\leftarrow$ index + temp_length
else
    msg.length $\leftarrow$ temp_length
    Extract payload from buffer starting from buffer[index] (up to msg.length)
    Copy buffer[index] to msg.payload (up to msg.length)
    index $\leftarrow$ index + msg.length
end if
Extract checksum from buffer[index]
msg.checksum $\leftarrow$ buffer[index]
index $\leftarrow$ index + 1
Extract end_byte from buffer[index]
msg.end_byte $\leftarrow$ buffer[index]
index $\leftarrow$ index + 1
return true

```
// Function to read the status of P2.2 (umbilical cord rover)
bool umbilicalcord_rover_connected(void) {
    return (P2IN & BIT2) != 0;              // Return the status of P2.2
}
```

Listing A.1: check whether umbilical cord is disconnected

B

Functional Flow Diagrams

B.1. Transit modes

B.1.1. General Startup

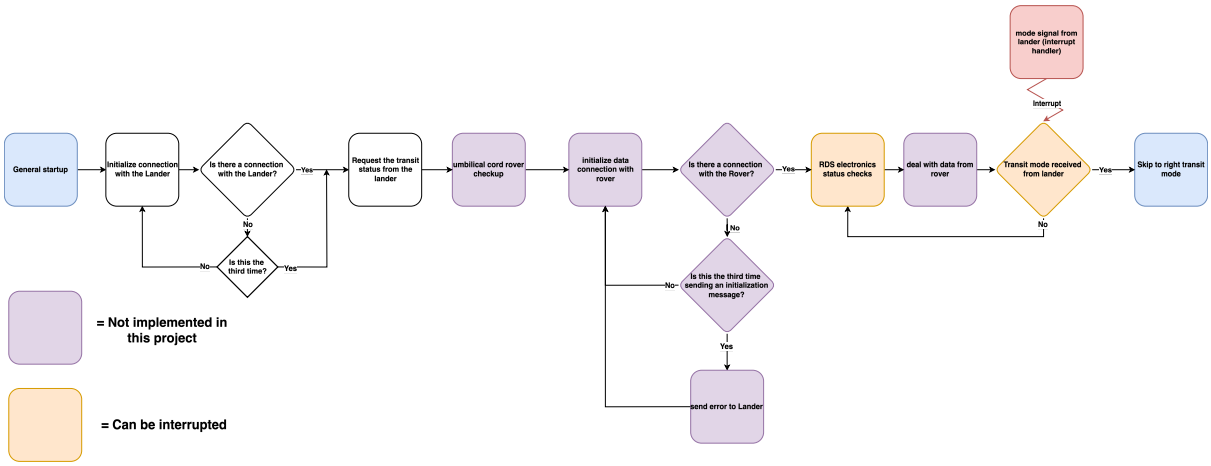

Figure B.1: Functional flow diagram of the general startup

B.1.2. Launch Mode

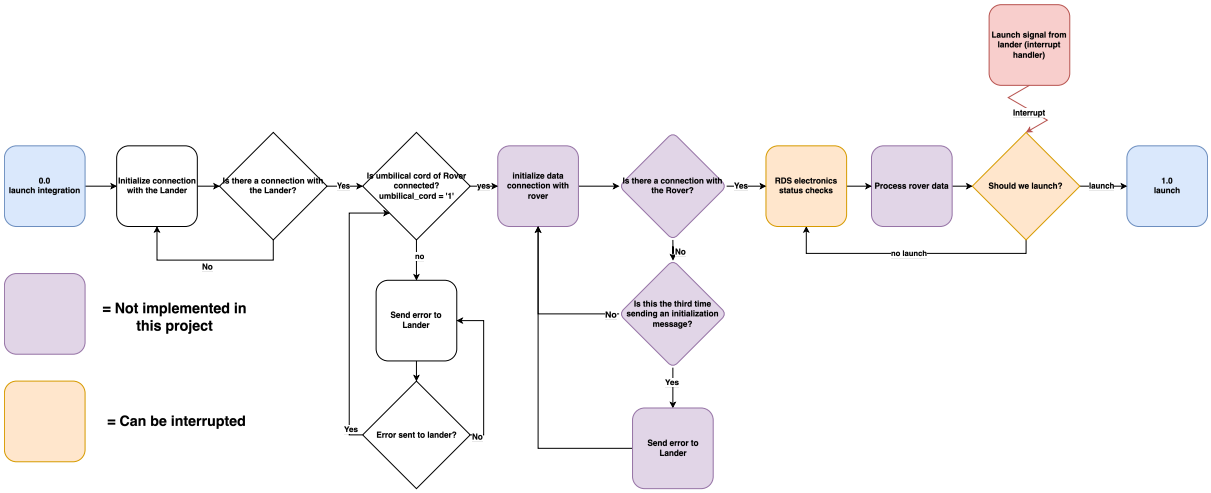

Figure B.2: Functional flow diagram of the launch mode

B.1.3. Travel Mode

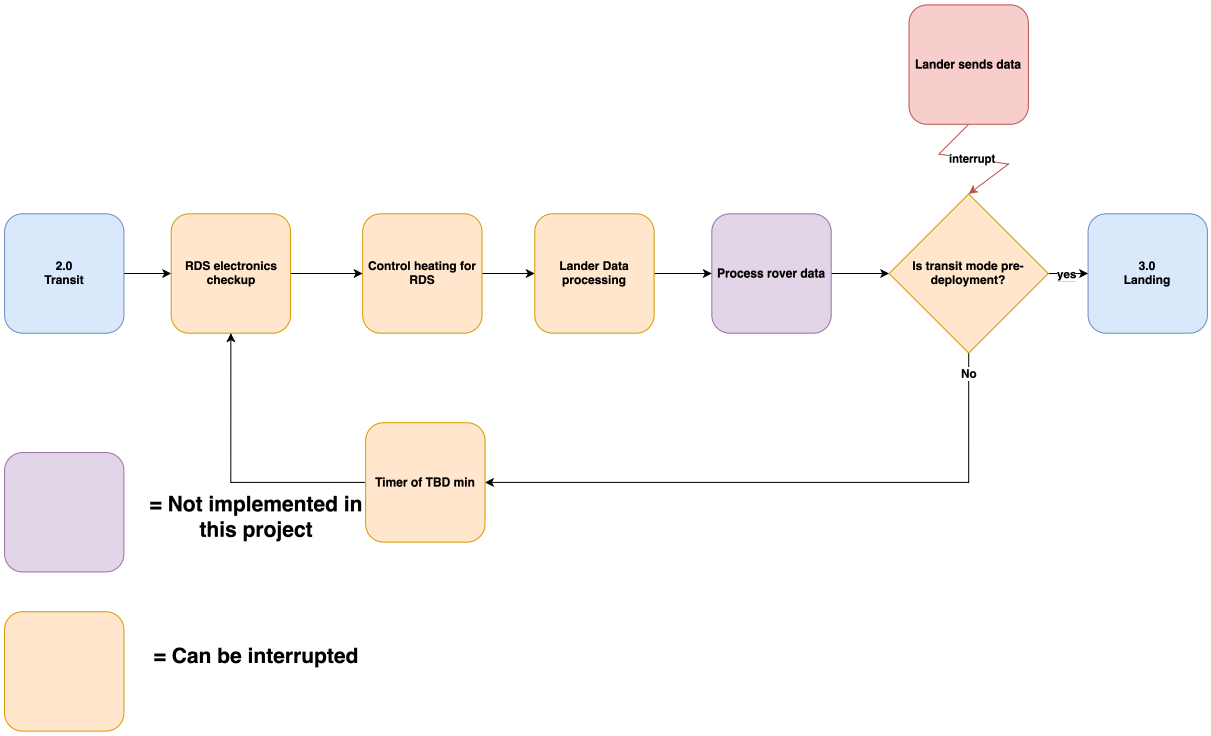

Figure B.3: Functional flow diagram of the travel mode

B.1.4. Pre-deployment Mode

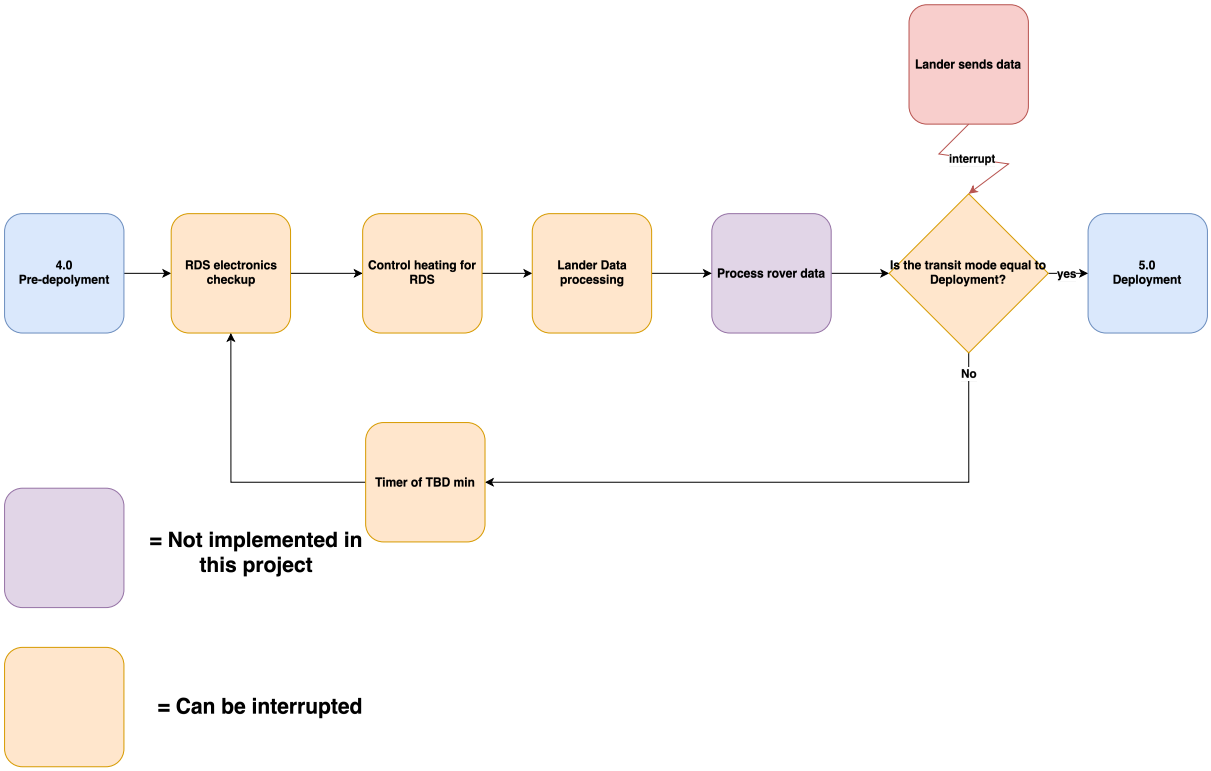

Figure B.4: Functional flow diagram of the pre-deployment mode

B.1.5. Deployment Mode

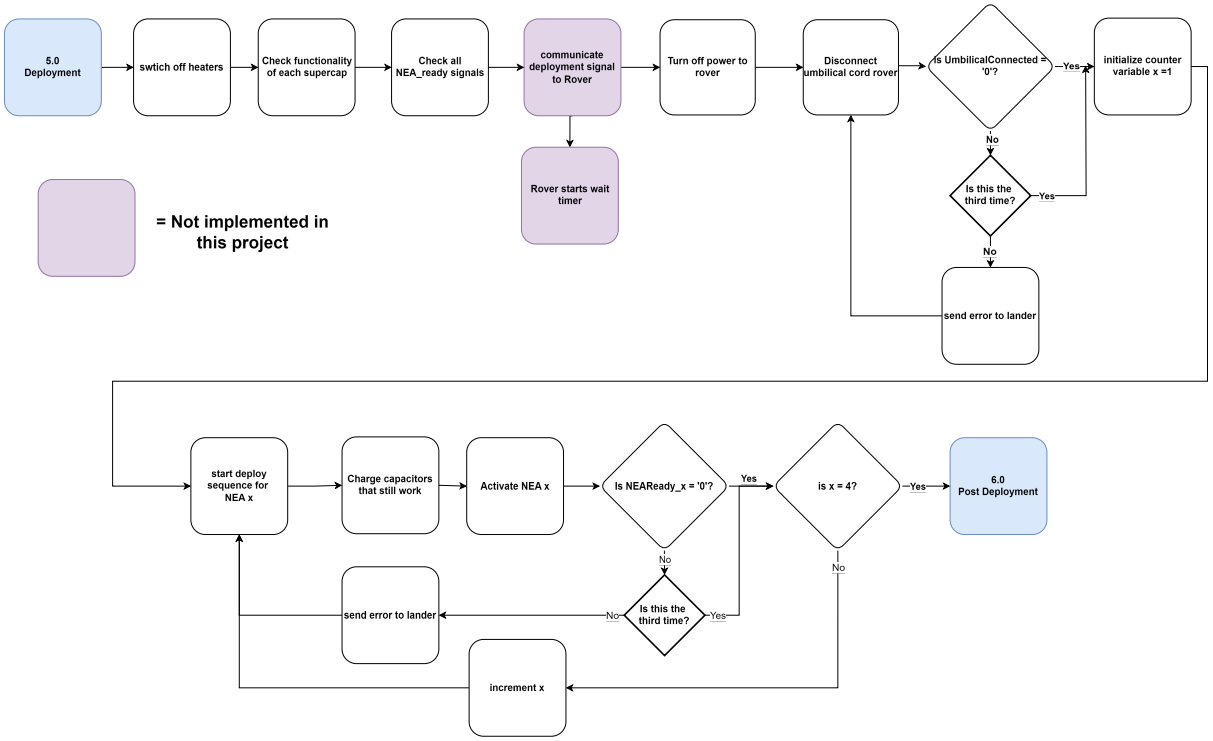

Figure B.5: Functional flow diagram of the deployment mode

B.2. Lander - RDS communication

B.2.1. Slip encoding

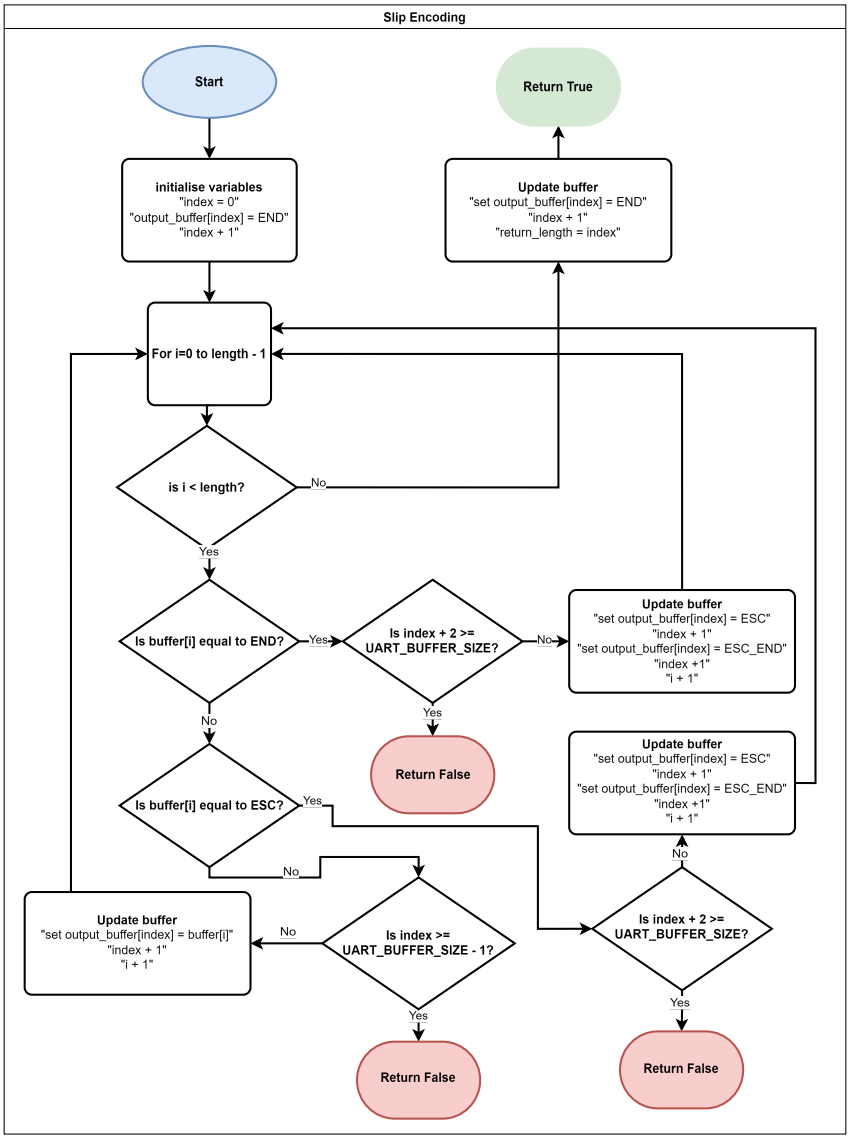

Figure B.6: Functional flow diagram of slip encoding algorithm

B.2.2. Receiving

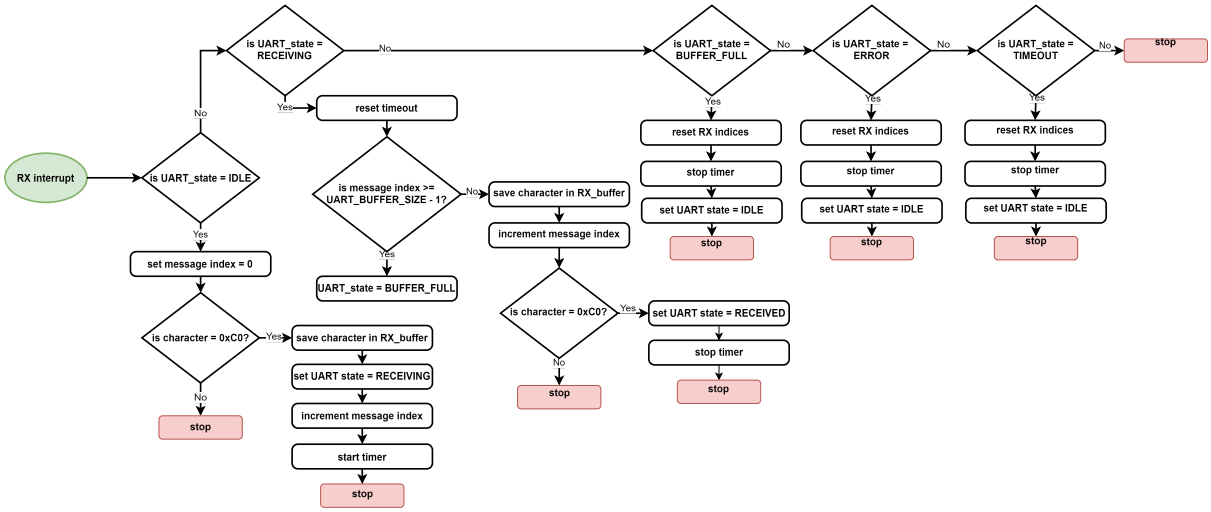

Figure B.7: Receiving process

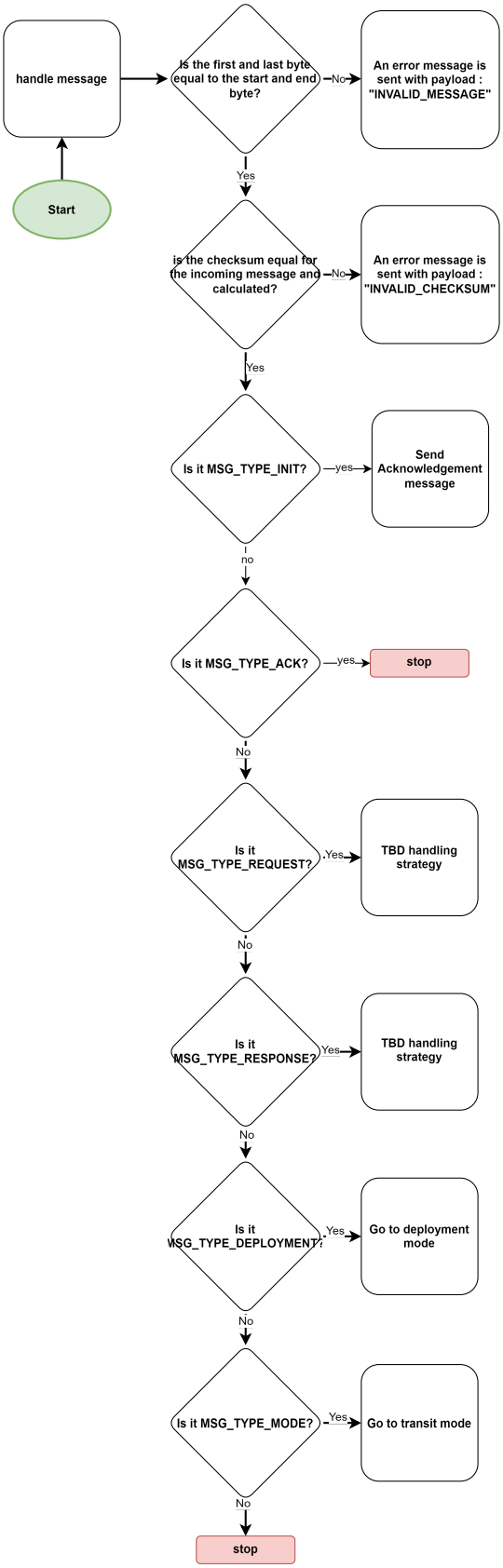

Figure B.8: Receiving handler

B.3. Electronics Components Control System

B.3.1. Frequency measurement

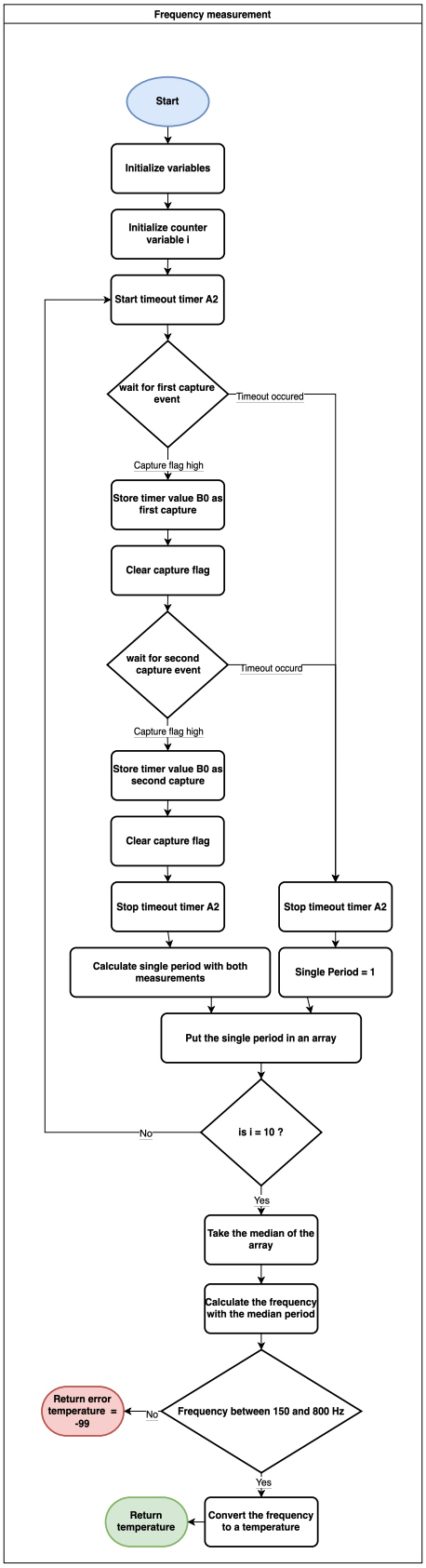

Figure B.9: Frequency measurement process

B.3.2. Heater Control

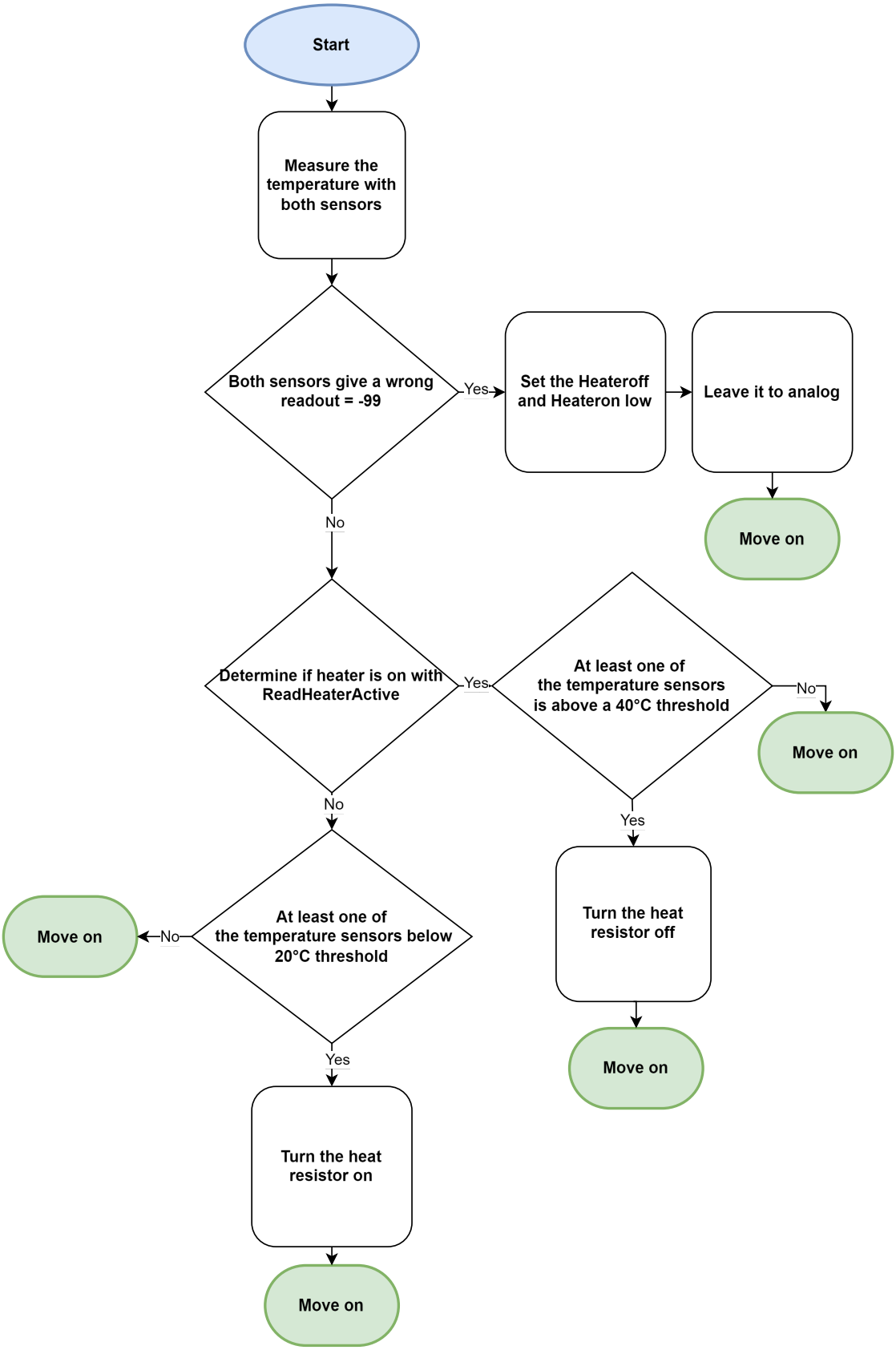

Figure B.10: Flow diagram of the heater control

B.3.3. NEA Activation

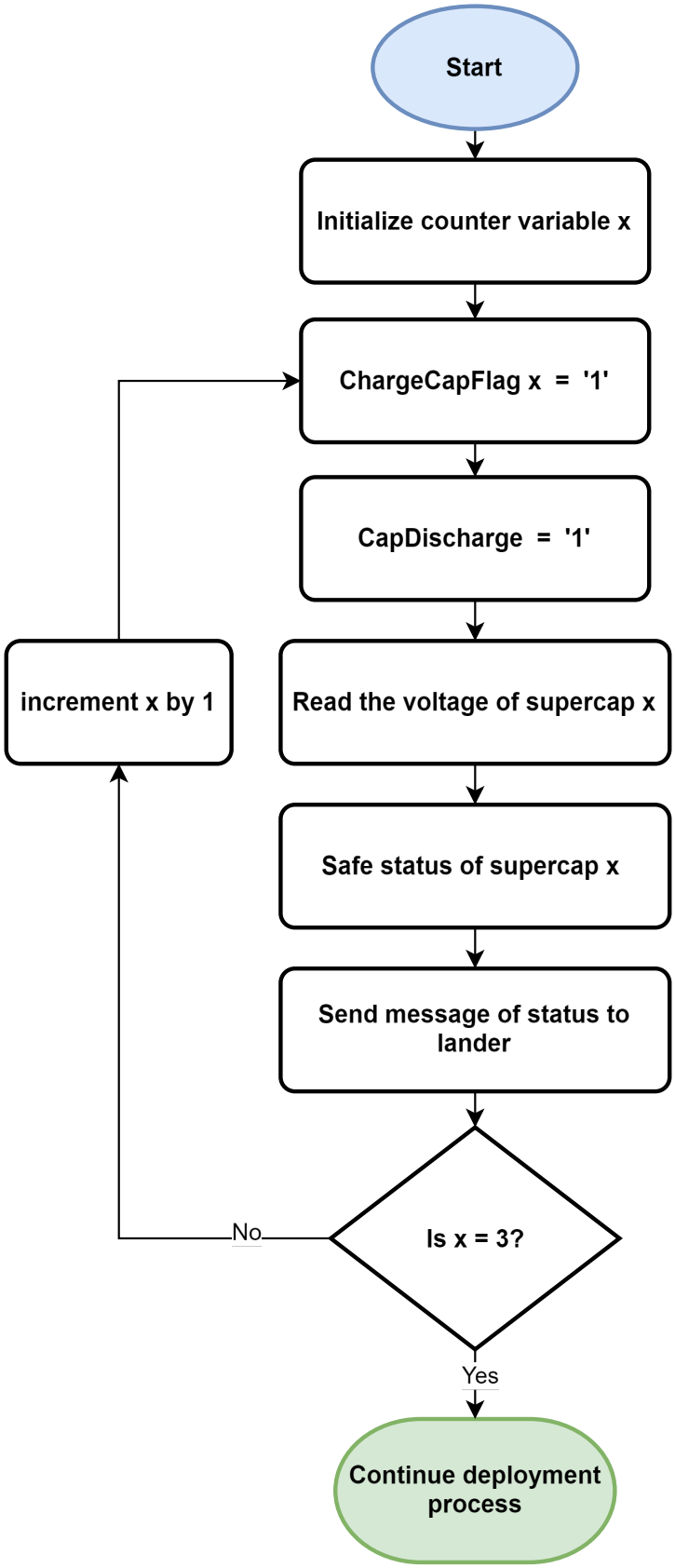

Figure B.11: Flow diagram of the supercap check

B.3.4. Capacitor Check Architecture

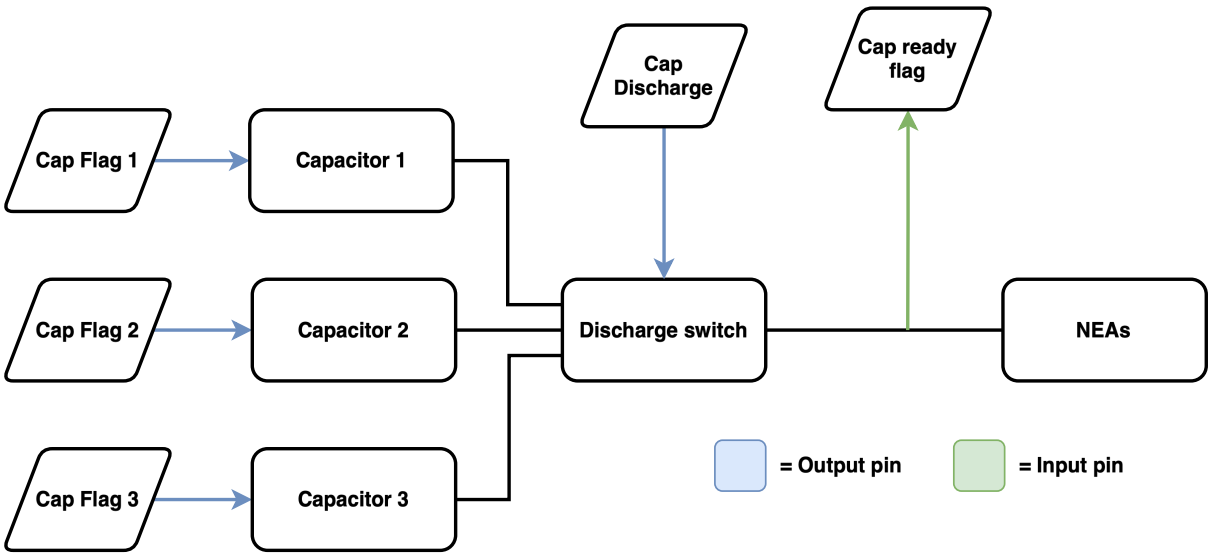

Figure B.12: architecture of capacitor check

C

List of Google Unit Tests

C.1. Lander-RDS communication

slip_encoding()

1. **Normal data test:** Ensure regular data bytes are passed through unchanged.
2. **Start byte test:** Verify the output starts with the END byte.
3. **End byte test:** Verify the output ends with the END byte.
4. **Empty buffer test:** Handle encoding when the input buffer is empty.
5. **Buffer larger than UART buffer test:** Handle encoding when the input buffer exceeds the UART buffer size.
6. **Buffer smaller than UART buffer but after encoding, it is larger:** Handle encoding when input buffer expands and exceeds UART buffer size after encoding.
7. **END replacement test:** Replace END bytes in the input buffer with ESC + ESC_END.
8. **ESC replacement test:** Replace ESC bytes in the input buffer with ESC + ESC_ESC.
9. **Sequential END bytes test:** Handle sequential END bytes correctly.
10. **Sequential ESC bytes test:** Handle sequential ESC bytes correctly.
11. **Mixed END and ESC bytes test:** Handle a mixture of END and ESC bytes correctly.
12. **Maximum payload size test:** Verify encoding with the maximum allowable input buffer size.
13. **Buffer overflow test:** Check for buffer overflow when encoding.
14. **ESC overflow test:** Check for not enough space for the ESC and ESC_ESC sequence.
15. **Maximum encodable data test:** Handle encoding expansion up to UART buffer size.
16. **Mixed special characters test:** Check encoding of a mixture of special characters.

slip_decoding()

1. **Normal data test:** Ensure regular data bytes are passed through unchanged after decoding.
2. **Start byte removed test:** Verify the payload has the start byte removed.
3. **End byte removed test:** Verify the payload has the end byte removed.
4. **Empty buffer test:** Handle decoding when the input buffer is empty.
5. **Empty payload test:** Handle decoding when the payload is empty.
6. **Input length larger than UART buffer test:** Handle decoding when the input length exceeds the UART buffer size.
7. **ESC + ESC_END replacement test:** Replace ESC + ESC_END bytes in the input buffer with END.
8. **ESC + ESC_ESC replacement test:** Replace ESC + ESC_ESC bytes in the input buffer with ESC.

9. **Sequential END bytes test:** Handle sequential END bytes correctly.
10. **Sequential ESC bytes test:** Handle sequential ESC bytes correctly.
11. **Mixed ESC + ESC_END and ESC + ESC_ESC bytes test:** Handle a mixture of ESC + ESC_END and ESC + ESC_ESC bytes correctly.
12. **Maximum buffer size test:** Verify decoding with the maximum allowable input buffer size.
13. **Check all four different characters for decoding:** Verify decoding for all special character sequences.
14. **Length zero test:** Correctly return false if the input length is smaller than 2 or if the start or end byte is not equal to END.
15. **No first slip encoding character:** returns false if there is no slip encoding character in the first position of the array
16. **No last slip encoding character:** returns false if there is no slip encoding character in the last position of the array
17. **First ESC character and then no ESC_ESC OR ESC_END:** ensure this returns false since this is invalid encoding.

convert_message_to_array()

1. **Convert normal message to array test:** Verify a normal message is correctly converted to an array.
2. **Start byte test:** Ensure the start byte is correctly placed in the output array.
3. **Message type test:** Verify the message type is correctly placed in the output array.
4. **Message length test:** Ensure the message length is correctly placed in the output array.
5. **Empty payload test:** Handle conversion when the payload is empty.
6. **Payload longer than UART buffer test:** Handle conversion when the payload length exceeds the UART buffer size.
7. **Checksum test:** Verify the checksum is correctly calculated and placed in the output array.
8. **End byte test:** Ensure the end byte is correctly placed in the output array.
9. **Buffer check test:** Verify the entire buffer is correctly populated with expected values.
10. **Length check test:** Ensure the output length is correctly calculated and returned.

convert_array_to_message()

1. **Normal test:** Verify a normal array is correctly converted to a message.
2. **Length smaller than five:** Handle an array that is smaller than length 5.
3. **Start byte test:** Ensure the start byte is correctly placed in the message.
4. **Message type test:** Verify the message type is correctly placed in the message.
5. **Message length test:** Ensure the message length is correctly placed in the message.
6. **Message length longer than max payload size:** Ensure that the message length longer than payload size is handled.
7. **Message length smaller than zero:** Handle the case where the message length is smaller than zero.
8. **Empty payload test:** Handle conversion when the payload is empty.
9. **Checksum test:** Verify the checksum is correctly placed in the message.
10. **End byte test:** Ensure the end byte is correctly placed in the message.
11. **Checksum verification test:** Check whether the checksum is correct.

C.2. Electronics Components Control System

calculate_frequency()

1. **Zero period test:** This test checks whether the function returns zero if the period is zero.
2. **Positive period test:** This test checks whether the function correctly calculates the frequency for a given positive period.
3. **Small period test:** This test verifies the function with a non-zero, small period.

frequency_to_temperature()

1. **Zero frequency test:** This test ensures that a default error value is returned if the frequency is zero.
2. **Positive frequency test:** This test verifies the temperature conversion for a given positive frequency.
3. **Small frequency test:** This test verifies the function with a small frequency.
4. **Frequency in range test:** This test checks the temperature conversion for a frequency within the expected range.
5. **Frequency in range test 2:** This test checks the temperature conversion for another frequency within the expected range.
6. **Frequency in range test 3:** This test checks the temperature conversion for yet another frequency within the expected range.

convert_adc_to_voltage()

1. **Zero value test:** This test checks whether the function correctly converts an ADC value of 0 to 0.0 V.
2. **Max value test:** This test checks whether the function correctly converts the maximum ADC value (4095) to the maximum voltage (3.64 V).
3. **Mid value test:** This test verifies the conversion when the ADC value is half of the maximum value.
4. **Quarter value test:** This test verifies the conversion when the ADC value is a quarter of the maximum value.
5. **Three-quarters value test:** This test verifies the conversion when the ADC value is three-quarters of the maximum value.
6. **Arbitrary value test:** This test verifies the conversion for an arbitrary ADC value (2048).

D

List of Figures

D.1. Lander-RDS Communication

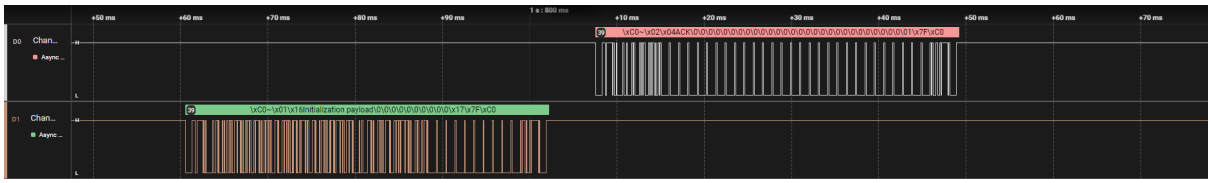

Figure D.1: Communication with a fixed payload size = 32 bytes, with baud rate = 9600 baud/s

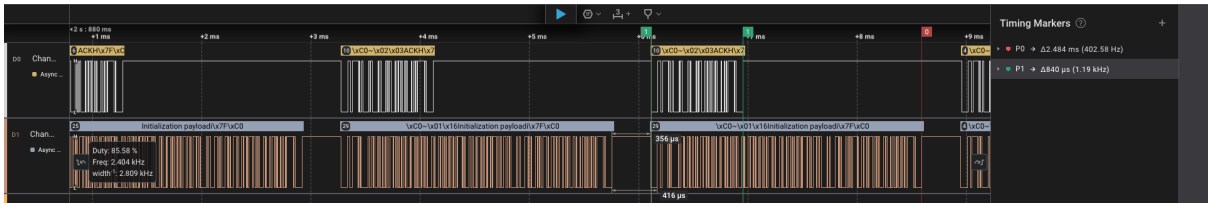

Figure D.2: Communication with a variable payload size with baud rate = 115200 baud/s

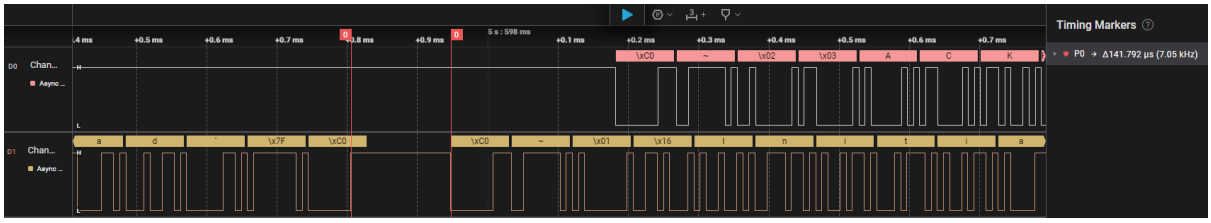

Figure D.3: Interrupt overflow occurring when sending messages without any delay zoomed in to see minimum time between messages

Figure D.4: System recovery when starting to receive messages halfway zoomed in

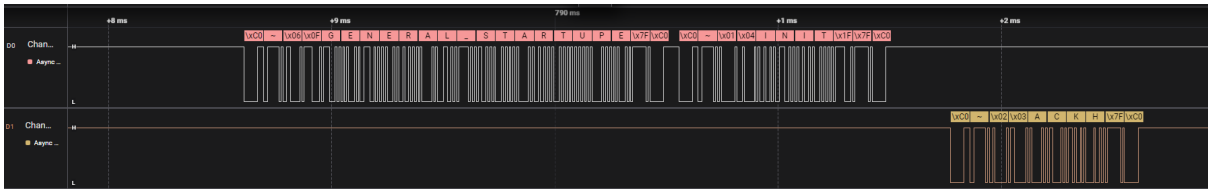

Figure D.5: General startup sequence test zoomed in on first part

Figure D.6: General startup sequence test zoomed in on second part

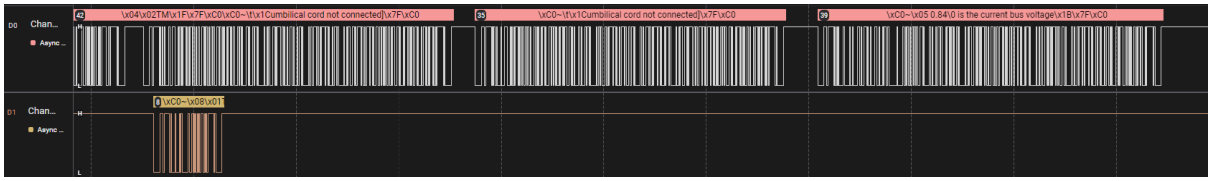

Figure D.7: General startup sequence test zoomed in on third part 3.1

Figure D.8: General startup sequence test zoomed in on third part 3.2

Figure D.9: General startup sequence test zoomed in on third part 3.3

Figure D.10: General startup sequence test zoomed in on third part 3.4

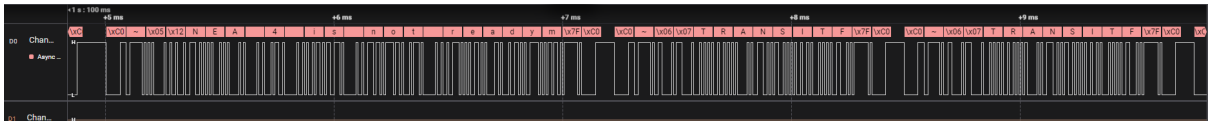

Figure D.11: General startup sequence test zoomed in on third part 3.5

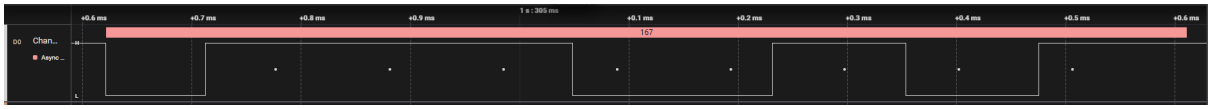

Figure D.12: First character ”167” being sent via UART output pin 2.5

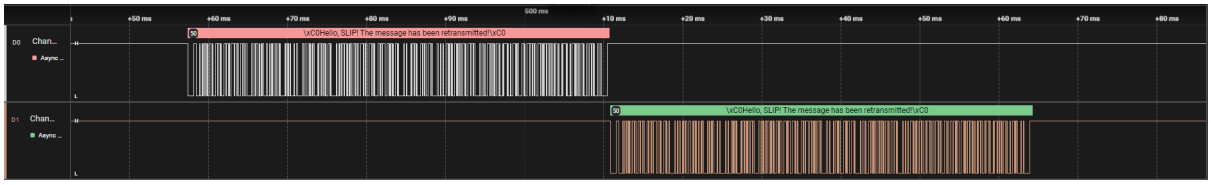

Figure D.13: First character retransmission of a message using the UART input pin 2.6

Figure D.14: First variable payload message using vectors

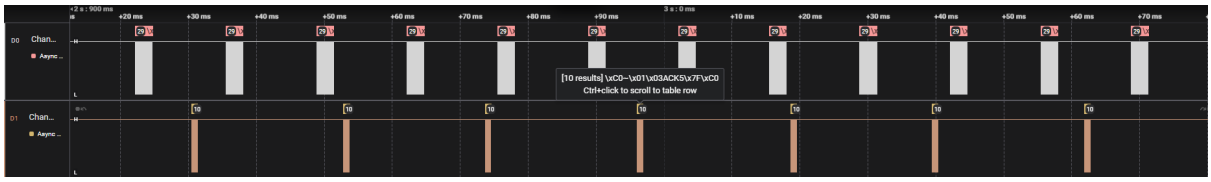

Figure D.15: error in acknowledgements for a baud rate of 115200 baud/s