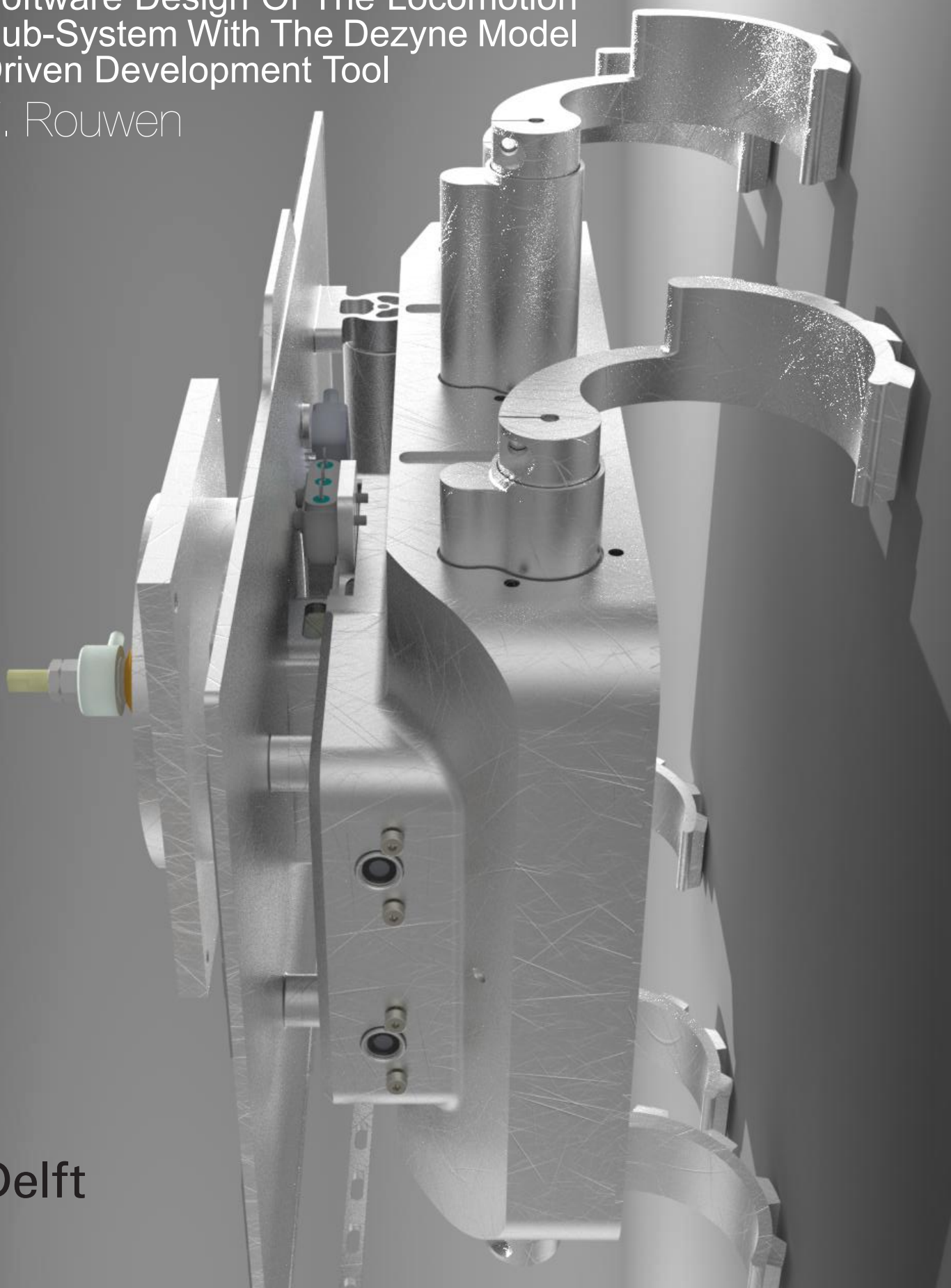
Lunar Zebro

Software Design Of The Locomotion Sub-System With The Dezyne Model Driven Development Tool

F. Rouwen

Delft University of Technology



TU Delft

Lunar Zebro

Software Design Of The Locomotion Sub-System With The Dezyne Model Driven Development Tool

by

F. Rouwen

in partial fulfilment of the requirements for the degree of

Master of Science
in Embedded Systems

at the Delft University of Technology,
to be defended publicly on 27 August, 2018 at 15:00.

Student number:	4076605	
Project duration:	26 June, 2017 – 27 August, 2018	
Thesis committee:	Dr. ir. C. J. M. Verhoeven,	Associate Professor, TU Delft
	Dr. ir. A. J. van Genderen,	Master Coordinator, TU Delft
	Dr. ir. J. J. A. Keiren,	Assistant Professor, TU Delft Assistant Professor, Open Universiteit
	Dr. ir. G. L. E. Monna,	CEO of Hyperion Technologies
Supervisor:	Dr. ir. S. Engelen,	CTO of Hyperion Technologies

This thesis is confidential and cannot be made public until September, 2020.

TU Delft Delft University of Technology

Abstract

The TU Delft ZEs-Benige RObot (Zebro) project is presented with the opportunity to bring the Zebro concept to the surface of our moon. To maximise the probability of success, the Locomotion Sub-System (LSS) software of Lunar Zebro is developed using a novel model-driven design tool called Dezyne. Dezyne uses a proprietary language to describe systems, perform model checking and execute source code generation. Dezyne is proposed for this thesis as it can improve the quality of the LSS design. This is required because the project is under strict time- and manpower constraints. The goals for this thesis are:

i. Deliver a reliable software design for the Locomotion Sub-System (LSS) of Lunar Zebro.
ii. Deliver a working implementation of the Locomotion Sub-System (LSS) software design on Lunar Zebro's hardware.
iii. Investigate the advantages and disadvantages of the use of the model-driven software design tool *Dezyne*.

The current workhorse of the Zebro Project, is a Zebro called DeciZebro. Lunar Zebro builds on the legacy of the design of DeciZebro. The LSS of Lunar Zebro consists of an Locomotion Controller (LC) and six leg modules. The requirements of the LSS software of Lunar Zebro are derived from the requirements of Lunar Zebro as a whole. These requirements are categorised according an European Space Agency (ESA) standard.

With the help of Dezyne, the system is designed, verified, simulated and integrated into native code. It is found that Dezyne is not suitable for designing the software of the leg modules. Therefore, the LC consists primarily of generated code, while the leg modules solely contain code developed by hand.

The goals of this thesis are partly reached. An LSS software implementation is delivered with the use of Dezyne. However, the design is lacking a framework in which errors that occur during operation can be resolved by means of extensive state machining. This is due to time constraints. Additionally, it is unclear if this specific LSS software implementation is more reliable than an implementation that is made without the use of Dezyne. The third goal, listing the advantages and disadvantages of Dezyne, is fulfilled.

Keywords— *Lunar Zebro, Model-Driven Design Approach, Formal Verification, Dezyne, Code Generation, Embedded Systems*

Contents

1 Introduction

The TU Delft ZEs-Benige RObot (Zebro) project is presented with the opportunity to bring the Zebro concept to the surface of our moon. A dedicated project team, the *Lunar Zebro Team*, will design and manufacture two extremely light Zebro rovers called Lunar Zebros within a time frame of less than a year. The Lunar Zebro team fully consists of students and is supervised by two associate professors. Due to the stringent time and manpower constraints on the project, Dutch industrial partners support the team and its mission.

This thesis describes the design and implementation of the LSS software of Lunar Zebro.

Outline This chapter introduces the mission that Lunar Zebro will embark on. The thesis subject is detailed and briefly motivated. Lastly, the thesis structure is outlined.

1.1. Moon Mission

Lunar Zebro is designed to piggyback on Indian Space Research Organisation (ISRO) its Chandrayaan 2 mission [41]. Chandrayaan 2 will travel to the south pole of the moon [43]. When the spacecraft enters an orbit around the moon, it will separate in two parts; an orbiter and a lander. The orbiter will stay in orbit around the moon and will function as a relay station for moon-to-earth communication. The lander will make a soft landing on the moon's surface and deploy the ISRO moon rover as well as Lunar Zebro.

Lunar Zebro shall function for the duration of one lunar day, equivalent to 14 earth days. Lunar Zebro is not required to survive the lunar night. Lunar nights are extremely cold and heating the rover during the lunar night is deemed infeasible by the project lead, due to mass restrictions. Lunar Zebro has three mission goals.

The first and most important goal for Lunar Zebro is to **communicate with earth** using either an emergency beacon or a full communication channel, both in the amateur radio band.

Secondly, Lunar Zebro shall traverse a distance of **200 meters** during the lunar day to show it can walk a significant distance for a small rover. This number is picked randomly by the project lead, but to see why it is significant, one can look to the Chinese lunar Rover Yutu. This rover could not drive 200 meters [1], indicating how challenging traversing the lunar terrain is.

Lastly, Lunar Zebro shall be able to take pictures and **send these to earth**. The imagery is also used for rudimentary crater and obstacle avoidance. With this, Lunar Zebro shall walk semi-autonomously.

When the lunar night starts, Lunar Zebro's mission is technically over. Practically, and because of the uniqueness of this opportunity, it is decided to ensure Lunar Zebro will have a fully charged battery and a special hibernation mode to maximise chances of surviving the cold lunar night. After the lunar night Lunar Zebro will wait for its internal temperature to rise before starting the boot-up sequence again. If this strategy leads to Lunar Zebro surviving the lunar night, it will be viewed as a major success and the mission will continue as on the first lunar day.

1.2. Thesis

To achieve the highest quality result with limited available time and resources, experimental procedures are employed. This thesis uses an experimental model-driven software design approach to accelerate the design process and increase productivity.

Thesis Goals The thesis goals are listed and explained below.

i. Deliver a reliable software design for the Locomotion Sub-System (LSS) of Lunar Zebro.
ii. Deliver a working implementation of the Locomotion Sub-System (LSS) software design on Lunar Zebro's hardware.
iii. Investigate the advantages and disadvantages of the use of the model-driven software design tool *Dezyne*.

The first and second goal need to be reached to have a successful mission. This will be done by using a model-driven design approach. Usually, a design approach of multiple design iterations and extensive system testing is used, but this approach does not fit in the time frame of the project. The model-driven design approach is helped by formal verification methods and code generation. Formal verification will add to the reliability of the design, while code generation will play a major role in delivering a reliable implementation of the design.

Dezyne is a model-driven design tool that has built in formal verification methods and code generation possibilities. Dezyne is used to enlarge the probability of meeting the thesis goals. However, Dezyne is not a widely used software development tool and therefore this thesis investigates the advantages and disadvantages of the use of this tool Dezyne. A potential advantage of Dezyne, which needs to be verified, is whether Dezyne will deliver reliable and safe LSS software within the project constraints of time and manpower. The reasons for choosing this tool are elaborated upon in chapter 2.

Project Motivation There have only been three countries, Russia, America and China, to ever operate a rover on the surface of the moon. Adding The Netherlands to that list with this project is all the motivation anyone needs. However, the performance of the rover greatly influences how this project will be remembered. Just as Eugene Francis Kranz, former flight director of the National Aeronautics and Space Administration (NASA), we will lay out all the options and failure is not one of them.

Failures often occur during the operation phase of a project. In the case of Lunar Zebro, these failures cannot be fixed in this phase. This thesis investigates one option to diminish the probability of failures occurring. Using a model-driven software design approach in combination with formal verification can potentially bring faulty execution paths to light long before the deployment of the rover.

Additionally, there is a more down-to-earth reason to explore the use of model-driven software design. When producing large numbers of devices, it also becomes increasingly more difficult, or even impossible, to patch software faults when the devices are operational. Investigating software design methods that increase reliability is in society its best interest.

Lastly, showing code can be reliably generated from formally verified models can benefit development of space-grade software. Looking at the European Cooperation for Space Standardisation (ECSS) software standards [25] and the NASA Software Safety Guidebook [4], they both acknowledge the potential of model checking but give no guidance on this subject. NASA only mentions a list of tools, containing SPIN and SMV, that can be used for software verification via model checking. This shows there is a gap to fill. However, this gap should be filled in an engineering sense. Instead of academically improving model checking methods, the tools that use model checking should be evaluated. It is the goal of this thesis to do just that.

Thesis Outline Chapter 2 gives the reader background on Zebro, Lunar Zebro and Dezyne. Chapter 3 describes the set of requirements for the LSS software. Chapter 4 elaborates on the Dezyne terminology and shows the system design of the LSS software and its verification. Chapter 5 discusses code generation and integration on the target platform. Chapter 6 shows the result of the design study, reflects on the set of requirements and lists the advantages and disadvantages of the use of Dezyne. Chapters 7 and 8 discuss the validity of the results from this project and conclude by summarising.

2

Background

ZEs-Benige RObot (Zebro) is an acronym directly translating to six-legged-robot. The robot's distinguishing feature is c-shaped legs, which enables it to overcome obstacles of its own height much easier than traditional wheels [40]. These legs, together with the motors, motor controllers and Locomotion Controller (LC) are called the Locomotion Sub-System (LSS) of Zebro. To give insight into how this definition of LSS came to be, this chapter elaborates upon the history and current status of the Zebro Project. This insight is important as Lunar Zebro builds on the lessons learned from the Zebro Project as opposed to doing a complete concept redesign.

Outline This chapter starts with a brief genealogy of Zebro, followed by a detailed description of the most recent iteration of Zebro, DeciZebro. Next, Lunar Zebro is described and how it derives from DeciZebro. Lastly, this chapter describes the difficulties of software development for space applications and why Dezyne is chosen to aid in the design of the LSS.

2.1. History

The development of Zebro at TU Delft can be traced back as early as 2009 to the faculty of Mechanical Engineering. Here, a four legged robot is developed by the systems and control group, based on the RHex concept [37]. Shortly after this, the same group develops a robot with six legs [29]. Both robots are shown in fig. 2.1.

The research from [36, 37] focuses on using Max-Plus algebra to implement walking algorithms on Zebros. The use of Max-Plus algebra results in optimal gait switching and allows for a host of interesting research topics like smooth turning of legged robots as shown by Suriana [50]. Research on the use of Max-Plus algebra continues to date.

In 2013, the faculty of Electrical Engineering, Mathematics and Computer Sciences is involved in the project [19]. The work focuses on creating a modular system to allow different groups of students to work together on Zebro more efficiently. Additionally, a modular design is theorised to increase reliability compared to the fixed design used by the systems and control group of Mechanical Engineering [29]. The separation and functionality of different modules in Zebro are inspired on the human body.

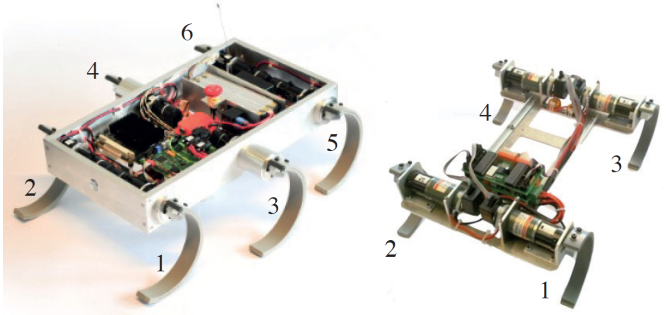


Figure 2.1: "Zebro robot on the left and RQuad on the right both developed at DCSC, TU Delft. The numbers represent the leg index numbering." [29]

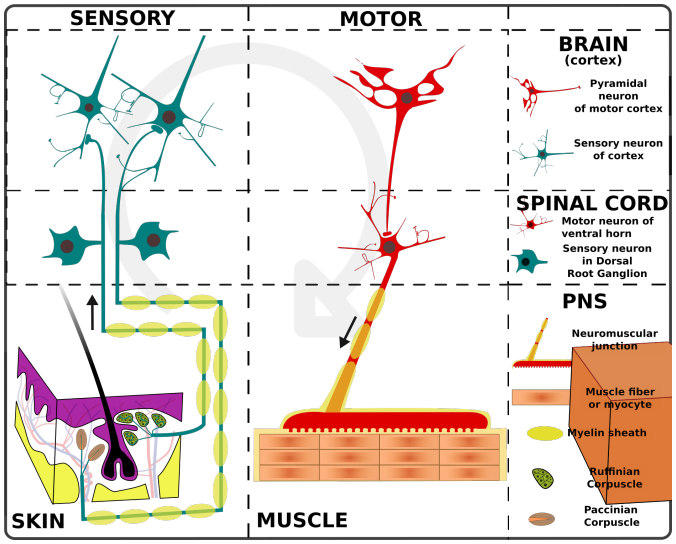


Figure 2.2: The three layers of the human nervous system. The lowest level contains nerves, or sensors, in our skin and muscles. The middle layer is a transport layer towards our brain that can initiate reflexes. The top layer, our brain, receives all information from lower levels, but can operate the human body on a higher level by commanding the lower levels.

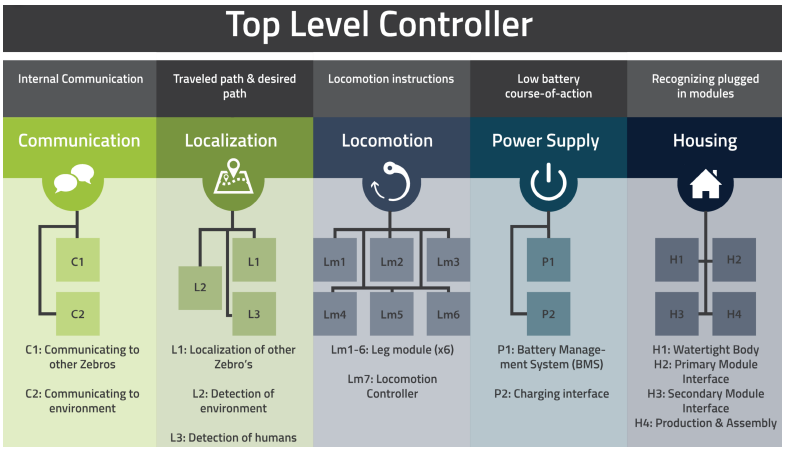
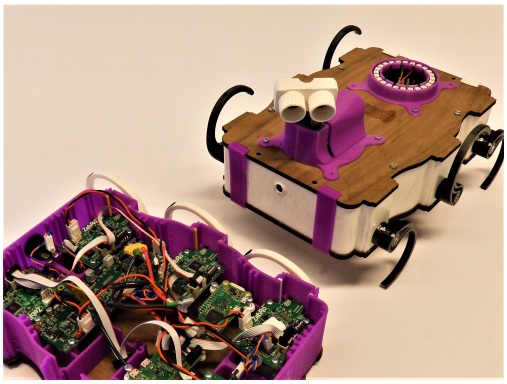


Figure 2.3: Modular pieces of Zebro as visualised in Otten [42, p. 26].

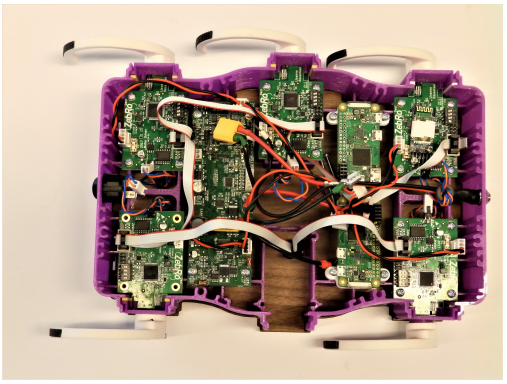
The human body has three distinct layers where processes take place to operate the human body. These are shown in fig. 2.2. Going bottom-up, the lowest layer is called the Peripheral Nervous System (PNS). This layer consists of nerves, or sensors, spread throughout the body that collect information about the surroundings and the functioning of the body. The middle layer is described as the spinal cord with atop the Cerebellum, or little brain. This layer transports all sensor input from the lower layer to the higher layer. A specific function of the Cerebellum is that it can aid in motor functions caused by reflexes in the lower layer. Lastly, all layers are controlled by the Cerebrum, or big brain. The Cerebrum receives input from, for example, our eyes and ears and uses this input to operate the body.

Zebro adopts the three layers as shown in fig. 2.3. The top level controller collects all information from sub-systems and it outputs, for example, a desired path or locomotion instructions toward the middle layer. The middle layer consists of devices overseeing one or multiple lower level devices. Examples of this are the LC and the power supply controller. The lower layer is formed by a host of devices shown at the bottom of fig. 2.3. Most notable for this project are the six leg modules. These lowest devices can trigger reflexes that can protect Zebro without sensor information actually reaching the top level controller.

The first project to implement one of these modular devices is a leg module project [18]. A leg module is a smart motor driver that not only controls an electric motor, but also monitors its safe functioning. It can, for example, protect against over-temperature or over-current and react with a pre-programmed remedy.



(a) DeciZebro standing with ultrasonic 'eyes' and an indicative LED ring.



(b) The internals of DeciZebro, showing five identical leg modules, a BMS on the left and two Raspberry Pis on the right.

Figure 2.4: The current design of DeciZebro as developed by the TU Delft Zebro project.

Nowadays, the modular design is stretched all across Zebro.

2.2. DeciZebro

After multiple iterations of Zebro, DeciZebro, displayed in fig. 2.4, now is the workhorse of the Zebro Project. It is used as a teaching and research platform within the faculty of Electrical Engineering, Mathematics and Computer Sciences (EEMCS).

Hardware DeciZebro contains all three system layers. The top level controller, a Raspberry Pi, oversees the sub-systems which are housing, power, locomotion and localisation. A communication sub-system is currently not present. Middle layer hardware is only present for the locomotion sub-system in the form of another Raspberry Pi containing the LC. The lower layer systems are a Battery Management System (BMS), leg modules and a communication module. All of which are designed and developed by the team.

This thesis focuses on the Locomotion Sub-System (LSS), which consists of six leg modules and the Locomotion Controller (LC). Each leg module is made up of a leg, a motor, a hall-effect sensor, a current sensor, a temperature sensor and a motor driver. The motor drivers take care of the low level driving of, in DeciZebro's case, brushed DC motors. Additionally, the motor driver functions as a motion controller that is able to rotate the leg to any arbitrary radial angle (from now on called *position*). This rotation can be done with any speed or direction, regardless of the current position of the motor. The arbitrary positions that the motor driver shall rotate to are provided by the LC. Communication between the LC and the six leg modules happens by means of an I2C bus on which the LC is the master talking to six slaves.

Software Because of the modular hardware setup of DeciZebro's system, the software is inherently modular. The leg modules and LC of DeciZebro both have their own piece of software with specific functionality.

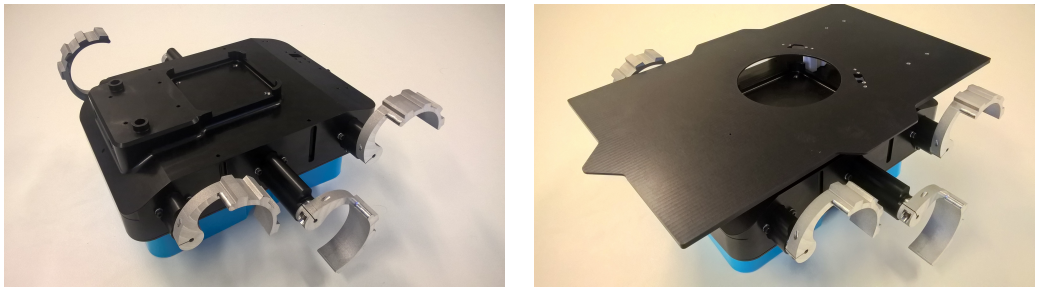
The current software architecture for DeciZebro's leg module is described in [17, p. 37-38]. This leg module receives commands from the LC. Commands can either be read or write. Read commands are answered with data from the leg module and write commands are executed.

The LC has middle layer software responsible for coordinating the movement of all leg modules in order for DeciZebro to walk efficiently. The current implementation of the LC for DeciZebro is in development and its design is not yet frozen. The motion for all leg modules is calculated using Max-Plus algebra in a similar manner as detailed in [29]. The research in this thesis benefits the DeciZebro project by supplying an LC design.

The LC is supervised by the top level controller fig. 2.3. At this time the implementation of the top level controller for DeciZebro is very limited. It only takes input from an ultrasonic distance sensor to determine whether or not an evasive movement should be made.

2.3. Lunar Zebro LSS

Even though Lunar Zebro's LSS is based on DeciZebro, they have their differences. The most important differences will be discussed in this section.



(a) Without solar panel attached. (b) With solar panel attached.

Figure 2.5: An early engineering model of Lunar Zebro, situated on top of a lunch box.

Hardware Lunar Zebro's leg modules consist of the same elements found in DeciZebro with minor differences in the elements itself. One difference is the use of a brush-less motor instead of a brushed one. The choice of motor type along with its specifications and the tests thereof, is elaborated upon in appendix A. The research necessary for this choice in motor is done during this thesis, but is not formally part of it.

The choice for brush-less motors removes the need for a relative encoder on the outgoing shaft of the gearbox. A brush-less motor has inherent position feedback implemented inside the motor, either by means of hall-effect sensors or back-emf measurements. In the case of Lunar Zebro the motors are equipped with digital hall-effect sensors. A sensor to determine the absolute position of the leg is still required. This sensor is the same as in DeciZebro, namely another digital hall-effect sensor. A permanent magnet is placed inside the legs of Lunar Zebro to trigger these hall-effect sensors and as such determine the absolute position of the leg once every rotation.

Lunar Zebro does not have an LC that runs on separate hardware as with DeciZebro. Instead, the top level controller and the LC run on the same hardware, the On Board Computer (OBC).

Communication between the LC and leg modules remains a bus structure where the LC is the only master on the bus. In Lunar Zebro's case, the bus is an RS485 full duplex bus.

Lastly, the adaptation of DeciZebro's leg design to suit this mission is discussed in detail in appendix B. Again, this design is done during this thesis but is not formally part of it. In fig. 2.5 an early engineering model of Lunar Zebro's leg design and body is shown.

Software It is clear from DeciZebro that two separate software architectures are needed to make Zebro walk. First, the leg module needs embedded software to control the motor, read sensor data and accept commands from an entity of a higher hierarchy. Secondly, the LC requires abstract code to keep track of the system status and generate positions for the motors to rotate to. These two software architectures shall be completely reliable and their behaviour must be predictable, even though environmental circumstances induce unpredictable inputs on the LSS. Section 2.4 describes the origin of these unpredictable inputs.

2.4. Space Software Challenges

Software that operates in space faces two main challenges. The first challenge is to verify that software fulfils the set of requirements at all times. NASA acknowledges in their software safety guidebook [4] that verifying the functioning of the system in all situations, is usually not possible.

> "However, it is often not possible to achieve 100% test coverage due to the enormous number of permutations of states in a computer program execution, versus the time it would take to exercise all those possible states." [4]

Additionally, when software errors are discovered during a mission, it is often extremely hard to update or patch the software in space. The consequence of this is that software must be guaranteed to work perfectly before the mission is launched. Guaranteeing this with testing is impossible as the NASA guidebook [4] noted. This leads to the conclusion that a software verification technique is needed which can guarantee proper functioning of software in a less time consuming manner than running system tests.

For precisely this reason, formal verification methods are employed by Jet Propulsion Lab (JPL) during the development of a number of spacecraft, including Cassini, Deep Space One and the Curiosity Mars Rover [33]. The tool used in these project is SPIN. It is used extensively in the Curiosity project to identify race conditions in the multi-threaded software architecture.

The second challenge faced by software in space is the challenge of soft errors. Soft errors can be defined as random changes of data or signals without having a broken system. There are many causes for

(soft) errors in space software, but 20% of all anomalies on satellites can be attributed to the environment as stated in [7]. A large part of this environment is made up of radiation. In microelectronics, Single Event Effects (SEE) are the main problem resulting from radiation according to [7], as they can create soft errors or latching circuits. ”High-energy particles are the predominant cause for faults within OBCs.” is concluded by [26] from research by [7]. Besides the SEE, there are also cumulative effects and electrostatic discharges caused by radiation. The cumulative effects can range from increased current draw, to completely failing components.

A tool to model different types of radiation and describe how it can create SEE and increase the Total Ionising Dose (TID) to which microelectronics are exposed is proposed in [3]. On top of that, [3] ”…also developed a new model that describes the Albedo neutron environment near the surface of the moon”. Albedo neutrons are created when different types of radiation hit a surface, in this case the lunar surface. Unfortunately, no conclusion is drawn on the accuracy of the model and the implications thereof. The lack of conclusions in this paper underpins how little, theoretically, is known about the effects of radiation on the functioning of hardware and software, other than soft errors occurring.

Because of radiation, any data part of a non radiation-hardened system can become corrupt. Only data residing in radiation hardened logic can be assumed to be fault-free according to [26]. The consequence of this is that Analog to Digital Converter (ADC) readings, process counters, memories and many more values can become corrupt. This premise is backed by studies like [22]. This study measured over 20.000 bit flips, soft errors, in different types of memory over the course of two years. Notably, many of these flips occurred over the South Atlantic Anomaly where earth’s magnetic field is weaker as shown in [7].

Example of Mitigation of Radiation Effects There are several solutions to mitigate the effects of radiation in microelectronics. Hardware solutions often rely on voting on redundant circuit levels and do not make use of any software [26]. These hardware solutions are almost always single-vendor solutions or custom solutions for specific projects. It is beneficial to look into software techniques that can handle hardware faults in Commercial Off-The-Shelf (COTS) microelectronics. The goal is to save cost and increase performance without losing reliability compared to radiation hardened microelectronics.

A software solution described in [26] uses coarse-grained lock-stepping to enable self-testing via voting on redundant hardware as the first step toward fault tolerance. Step 1 of this technique is to find hardware errors. The knowledge of these errors is used in step 2 and 3. Step 2 utilises Field-Programmable Gate Array (FPGA) reconfiguration to re-purpose damaged logic cells and retain full functionality of the device. If many logic cells were to be permanently damaged, automated reconfiguration can not happen without loss of functionality. Additional status information can then be sent to earth for further investigation. If the processor would lose functionality in step 2, step 3 ensures a proper functioning of the processor. Step 3 ensures threads with a higher priority are automatically preferred over other threads to maintain satellite operation at the cost of low priority tasks.

Methods like the one described in [26] generate a hardware and software overhead that significantly increases development cost. Mission critical equipment in large projects that require 100% up-time for a mission duration of many years may benefit from a similar approach as described in [26]. However, it is unknown how much these methods actually increase the reliability of the system as the method of [26] is not applied in practice yet. For these reasons, Lunar Zebro is thought not to benefit from this approach. The project has a very small budget compared to that of ESA projects and time and manpower are very limited. Low cost and easy implementable mitigation strategies are thus required.

2.5. Proposed Software Design Technique

Even though money, time and manpower are limited, the same environmental requirements are imposed on Lunar Zebro as for any other lunar rover. To ensure Lunar Zebro is able to meet its requirements, several steps are taken. Instead of using redundant hardware or software, the use of a model-driven software design approach for Lunar Zebro’s LSS software is proposed. Although no substantial evidence exists, this method is expected to increase the probability of meeting the requirements and increasing the reliability of the system. Additionally, formal verification of software is proposed to spot critical design flaws early on in the design process.

Why a model-driven Approach? The first anticipated merit of using a model-driven design approach in combination with formal verification, is a guarantee of a functioning system. Although it might not be possible to verify specific functionality, it is possible to easily verify deadlock and livelock are not present in the system. This guarantees the system is, at the least, always functioning. This is extremely useful to spot design errors early on in the project.

The second anticipated merit of this approach is the limitation of the effects of soft errors caused by radiation. Corrupt data can trigger events causing unwanted state changes. Worst case, these state

changes force the system to go into an emergency state where errors, in this case the corrupt data, can be solved. This is at the very least a predictable scenario, which is considered advantages.

Lastly, there is the matter of verification and validation that is normally done in space projects. The differences between two companies using ESA's ECSS software standards and their struggles during verification and validation of their products is shown in [23]. One of the companies uses model-driven design approaches and code generating tools to create products of a higher quality and increase productivity. This makes proving compliance with ECSS much harder since the ECSS software standards have no provisions regarding the use of this kind of tools. Proving model-driven tools and code generation result in reliable software for space applications, can greatly benefit the development of these products. Especially when this software runs on COTS components and still increases reliability of the system compared to redundant hardware solutions. It could pave the way for allowing code generation and higher reusability of code. This increases productivity since projects can be built on top of already existing code that is built and verified by others.

Why Dezyne? The goal of this thesis is to use a model-driven design approach and employ model verification (model checking) to result a more reliable software system necessary for the type of mission that Lunar Zebro will embark on. Next step in this process is to find tools to support this way of software development. The Dezyne tool is proposed by Dr. ir. J.J.A. Keiren, an assistant professor at TU Delft and the Open University.

Dezyne is developed in the Netherlands by the company Verum, in close cooperation with members from the Technical University of Eindhoven [10]. Dezyne boasts "…a 50% reduction in development costs, 20% decrease in time to market and a 25% reduction in the cost of field defects" [10]. This claim is not yet substantiated by either Verum or an independent source. Dezyne uses a proprietary language to model software systems and allows for code generation to different languages as, amongst others, C and C++ [10]. Generating code is another step towards mitigating errors, in this case human errors originating during development.

Dezyne can also translate its proprietary language to the Micro Common Representation Language 2 (mCRL2) [16, 20] language to offer model checking capabilities. With mCRL2, Dezyne is able to check a model for deadlock and livelock.

Summarising, Dezyne is one tool that does precisely what is requested for this thesis. An all-in-one solution. Dezyne is free to use for open-source projects making it very accessible. However, Dezyne is not the only tool to do this. For example, Quantum Leaps, LCC [46] offers some of the same opportunities as Dezyne. However, it does not include a formal verifier. Esterel-Scade Suite does include a verifier [48]. Unfortunately, this software is not free to use for open-source projects. Beside the fact that the other product don't offer the same as Dezyne, an extra reason for choosing Dezyne is simply a matter of support. It is expected a Dutch company will be able to deliver more support as it is physically close by and previous contacts between the company and the universities exist.

2.6. Design Process

The following steps are taken in the design process. First, project requirements are translated to dedicated requirements for the LSS software. This is done in chapter 3. Then, the LSS system is modelled, verified and simulated in Dezyne as shown in chapter 4. For the LC, the state machine source code is generated by Dezyne and additional integration is done to accommodate the hardware of the LC. The integration of generated code is discussed in chapter 5.

The leg module is programmed classically without the help of Dezyne. The design process is not formally part of this thesis, but is executed during this thesis.

The leg module is not expected to benefit from the use of Dezyne for two main reasons. First, the ratio of native code and generated code is tipping heavily towards native code. It is expected that the leg module will not have a large state machine and as such only a small portion of the code can be generated with Dezyne.

Secondly, the leg module features a micro processor that is capable of handling hardware interrupts without the processor knowing. This means state changes are possible without the state machine knowing and rendering Dezyne's generated state machine useless. To circumvent this, flags can be set in interrupts to allow events to be scheduled by Dezyne's generated code. However, real-time operation might be lost because of this which is deemed an insurmountable consequence.

3

Requirements

This chapter lists the requirements for the Lunar Zebro LSS software. They are derived from the global requirements for Lunar Zebro set by the systems engineers. The requirement categories are kept identical to the global categories to ensure smooth integration. They are based on documentation of ESA [24].

Two requirement categories have been omitted: Human Factor- and Physical requirements. Human Factor requirements are omitted since the LSS will not interact with humans physically. Software interaction with humans is described in either interface requirements or configuration requirements. Since software has no physical properties, Physical requirements do not apply to software. One can argue that code size or communication speeds can be listed in this category, but these are, if applicable, listed under Configuration requirements.

All requirements are given a priority as the time frame may not allow all requirements to be fulfilled. The priority is marked with asterisks, where (*) is highest priority and (***) is lowest priority.

Outline Ten requirement categories described by ESA's system engineering standards [24] are discussed. Every requirement category enumerates prioritised requirement statements and elaborates upon them.

3.1. Functional Requirements

The functional requirements describe what the LSS shall do in order to fulfil the mission. A list of Functional requirements for the LSS software:

[Func-1] The LSS software shall be able to receive a start command and with it, initiate walking of Lunar Zebro. (*)

[Func-2] The LSS software shall be able to receive a stop command and safely come to a stop or even negotiate to lay down. (*)

[Func-3] The LSS software shall be able to stand up, lay down, walk forward, backward and make left and right turns. (*)

[Func-4] The LSS software shall be able to walk at 3 different incremental speeds. (low, medium, high) (**)

[Func-5] The LSS software shall be idle if the solar panel is deployed. (**)

The first mission objective for Lunar Zebro is to walk 200 m in a certain direction. This can not be done on one battery charge that fits within the mass constraint. Thus, a solar panel must charge the battery. It is decided the solar panel will be deployed with a motor. When the solar panel is deployed, Lunar Zebro is not allowed to walk to prevent tumbling over. Other objectives of Lunar Zebro are to communicate with Earth, take pictures and have a science payload on board. It is unclear if the LSS can be active when the science payload is active.

From the mission objectives, the functional requirements of the LSS are determined. The LSS software is commanded by the OBC. The OBC shall be able to start and stop walking of the LSS whenever necessary. For example when the solar panel is deployed, or a science payload demands the rover to lay down. Currently, it is unknown if the LSS can function while the solar panel is deployed.

Lunar Zebro shall not only be able to walk forward. Regardless of whether or not the rover will have obstacle avoidance, there is the possibility of it getting stuck. The best way to get the rover free again is

to have it move in all other directions: backwards, left and right. These commands shall be accepted and executed by the LSS software. The LSS shall be able to move Lunar Zebro at different speeds. This is required because it is not known how Lunar Zebro will perform in the lunar environment. This environment has rock, boulders, loose soil, craters and many more obstacles. It can be useful to have different walking speeds on different terrains in the lunar environment.

3.2. Mission Requirements

Mission requirements are constraints that are induced by mission scenarios. A list of Mission requirements for the LSS software:

[Miss-1] The LSS software shall not make assumptions about the position of the legs at boot. (*)

[Miss-2] The LSS software shall be able to position the legs appropriately for storage. (**)

[Miss-3] The LSS software can protect the locomotion hardware from damage during deployment. (***)

Lunar Zebro will be mounted on the spacecraft in between two plates. One plate will touch the top of Lunar Zebro, the solar panel. The other plate will touch the bottom of the Lunar Zebro. This requires the legs to be rotated such that they do not surpass the plane parallel to the solar panel or the bottom plate. Forcing the legs to a certain position by applying an external force to the leg is forbidden since it may damage the gearbox. Thus, the LSS shall have a stow-command that brings the legs into the correct position.

During the transit towards the lunar surface, Lunar Zebro shall be turned off entirely. Only when the rover is deployed, shall it become active. It is not clear how Lunar Zebro will be deployed. The worst mission scenario is a drop from a certain height. In this scenario it is necessary to protect the legs and the motor from suffering damage on impact. The only way to do this, is to make sure the bottom of Lunar Zebro makes contact with the ground before any of the legs do so. The LSS software could play a role in this, but it is unknown how until further details about deployment are available.

After the deployment, still assuming a worst case scenario, it is very likely the legs did not retain a previously specified position. Therefore, LSS software may not assume the legs to be at a certain position when the system starts.

3.3. Interface Requirements

The LSS interfaces directly with the Power sub-system, Structures sub-system and top level controller on the OBC.

3.3.1. Power Interface

A list of Interface requirements for the LSS software regarding the Power interface:

[P-Int-1] The LSS software current limit should be lower than the hardware current limit managed by the Power sub-system. (**)

[P-Int-2] The LSS software shall be able to handle shut down or reset of any component, leg module or LC, safely at any time. (*)

The interface with the Power sub-system is simply a set of power lines. The power on these lines is not controlled by the LSS, meaning it can be cut by the Power sub-system at any time. An implication of this is that any part of the LSS can be reset by the Power sub-system at any time.

The Power sub-system delivers regulated 12 V for LSS to use. The system will limit the amount of current the LSS can draw. The LSS shall ensure its own current limit shall be lower than the hardware current limit of the Power sub-system to not lose control of its own system.

3.3.2. Structures Interface

A list of Interface requirements for the LSS software regarding the Structures Interface:

[S-Int-1] Every leg module shall be able to drive any leg at any position in Lunar Zebro. (**)

[S-Int-2] The LSS software shall not impose constraints on the integration of the LSS hardware and the design of the Structures sub-system. (**)

The Structures sub-system of Lunar Zebro is comprised of all the mechanical parts. One could summarise it with 'the body' of Lunar Zebro. The interface with the Structures sub-system is obviously a mechanical one.

The LSS software shall not constrict the design of the Structures sub-system. The Structures sub-system is constricted when the LSS has different mechanical requirements for different modular pieces. Therefore, the LSS shall have identical leg modules, that can drive any leg.

Additionally, the LSS software shall make sure it does not require specific integration of hardware to result a working system. Therefore the LSS software shall be independent of position and function in Lunar Zebro. When components are misplaced, the LSS must be able to adapt.

3.3.3. OBC Interface

A list of Interface requirements for LSS software regarding the interface between the top level controller and the LSS software:

[O-Int-1] The interface shall be synchronous, meaning the LSS software shall only send data to the top level controller when explicitly asked for. (*)

[O-Int-2] The interface shall have a regular heartbeat that ensures failures are detected on either side of the interface or in the interface itself before harming Lunar Zebro. (*)

[O-Int-3] The LSS shall be able to supply the top level controller with all data in the system. (*)

[O-Int-4] The LSS software shall report data over this communication line in SI-units and/or ensure they have meaning. (***)

In the current Lunar Zebro OBC software design, the top level controller is a thread running on the OBC. The sub-systems are separate threads and there is no communication between sub-systems. The top level controller communicates separately to all other threads running on the OBC via sockets. The LSS is one of those threads. The communication is synchronous and all sub-systems are slaves.

Lunar Zebro is designed to have modular software components that can work independently to relieve the top level controller. The LSS software is one of these modular systems. The top level controller is set up to only keep in touch with the LSS software for the bare minimum. The top level controller can not handle asynchronous messages from the LSS since it has synchronous round-robin-like communication with all its sub-threads. Minimal communication can result, best case, in a scenario where the OBC gives the LSS a command to start walking and 15 minutes later a command to stop walking. In between these commands no communication should be required for the system to stay functional. This has the downside that an error on either side may go unnoticed. This is the reason to implement heartbeat functionality between the top level controller and the LSS software. A correctly received heartbeat ensures the interface is still intact.

The top level controller on the OBC shall be able to receive housekeeping or debugging data from the LSS when requested. The debugging information is necessary for the development and integration stage of the system and will overlap to a certain degree with the housekeeping data during the mission. All relevant data originates at the leg modules and the LSS shall be able to report this data.

Lastly, all communication to and from the LSS shall be in SI-units and have meaning. For example, the LSS software should report temperature in degrees Celsius and not as a raw ADC reading. This ensures an even more clear system interface.

3.4. Environmental Requirements

A list of Environmental requirements for the LSS software:

[Env-1] Communication channels in the LSS software shall have error checking like parity or Cyclic Redundancy Check (CRC). (*)

[Env-2] The LSS software should sanitise all input and ignore unrealistic or potentially dangerous values. This should be done for all functions. (**)

[Env-3] All processing units of the LSS shall be able to individually reset by command or by them-selves. (*)

The lunar environment is a harsh environment. If hardware is selected that can withstand this environment, no special care has to be taken during software development to mitigate environmental effects. However, hardware that can withstand the lunar environment is often expensive or may not exist. In this case, software can aid inferior hardware to still produce a reliable system.

The LSS software is required to help the hardware withstand the Lunar Environment by detecting errors anywhere in the system. These can be errors in memory or in communication. It is required to have secure communication channels using error-detection methods. Additionally, all functions should sanitise their input to filter unrealistic or potentially dangerous values.

It is not required to implement proven methods like memory washing or parity bits per register words. This thesis shall show the performance of systems without these measures, but with verified application software.

Aside from detecting errors, the errors also need to be resolved. It is not required to have extensive error solving strategies although this can become a goal in future projects. There is one solving strategy

that is strictly required: resetting. All processing units of the LSS shall be able to individually reset by command or by them-selves.

Lastly, the LSS creates self-induced environmental effects: magnetic fields, thermal loads and vibrations. These effects are also mitigated by the requirements listed above.

3.5. Operational Requirements

A list of Operational requirements for the LSS software:

[Oper-1] The LSS software shall always listen for communication from the top level controller and shall not be able to turn itself off. (*)

[Oper-2] The LSS software shall have a (reset) function that is guaranteed to return to a safe state for the LSS and Lunar Zebro as a whole. (*)

[Oper-3] Lunar Zebro shall be able to reset and the LSS software, as such, shall be able to reset at any given time. (*)

[Oper-4] The LSS software shall be able to reset every leg module independently. (**)

[Oper-5] The LSS software shall always be able to go to an emergency state where error solving can take place. (*)

[Oper-6] When the LSS software is in an emergency state, it can only move to the idle state and this move shall be made when all errors are solved. (*)

[Oper-7] Starting or stopping any operation shall leave the LSS in a safe condition. (*)

[Oper-8] On a test-bed of sand, provided by the faculty, while subject to Earth's gravity, Lunar Zebro shall walk with a speed of at least 1 mm/s. (***)

The LSS shall never be completely off, it needs to listen for commands constantly. The sub-system shall not have an implementation where it turns off and automatically wakes after a certain amount off time. In short, when Lunar Zebro is turned on, the LSS is functional and listening. This state shall be guaranteed to be safe.

The LSS software shall be commanded to start and stop operation. Both shall be done in a safe way. The LSS can not just start or stop any action without physical consequences. To start operations, system variables shall be checked first. To stop operations, actions shall first be finished successfully, if at all possible.

In the event of a fatal error, the LSS software shall always be able to enter an emergency state immediately. This state ensures the LSS is only listening to commands and will not accept operational commands. From this state the LSS shall have the functionality to test different sub-systems for errors. This is necessary to determine if a leg is stuck or drawing too much current due to, for example, ageing of the gearbox.

Lunar Zebro's mission is to walk 200 m during one lunar day. It is expected that Lunar Zebro can not use the entire day for walking. This depends, among others, on when Lunar Zebro is deployed. On top of that, it is also an objective to have a full battery when the lunar night sets in. Because of these reasons, the window for Lunar Zebro to walk the desired 200 m in is shortened to an estimated 240 hours. Lunar Zebro will not be able to walk continuously. It is assumed that Lunar Zebro will be draining the battery four times faster than charging the battery using solar energy when walking. This leaves 60 hours that are actually spent walking. The minimum speed to reach the goal of 200 m now becomes 1 mm/s.

3.6. Logistics Requirements

The logistic requirements for the LSS software are defined as "conditions needed for the continuous utilisation of the system" [24]. A list of Logistic requirements for the LSS software:

[Log-1] The source code of the LSS software shall be documented during and after the development. (**)

[Log-2] The source code of the LSS software shall be stored on a location accessible to the Lunar Zebro team. (**)

The LSS software's life cycle does not span further than the mission duration. During the mission, the LSS software is utilised. When errors in the LSS software are discovered during the mission, there are requirements for *continued* utilisation of the system.

The error shall first be found and fixed on earth. This requires the LSS software to be properly documented and the source code to be stored in a location that can be accessed by the project team.

Properly documenting the functionality of the source code has the added benefit that, in case of emergency, other people can easily get involved in the development of the project.

3.7. Product Assurance Requirements

A list of Product Assurance requirements for the LSS software:

[Prod-1] The LSS software shall be 100% reliable during the mission. (*)

[Prod-2] The LSS software shall be 100% available during the mission. (*)

[Prod-3] Dezyne shall be used to increase the maintainability of the LSS software. (***)

[Prod-4] The LSS software shall protect the LSS hardware against over-current. (*)

[Prod-5] The LSS software shall monitor the temperature of all LSS hardware and report this to the top level controller. (**)

[Prod-6] The LSS software shall satisfy all requirements that are marked with the highest priority, as stated in this report. (*)

This category of requirements covers the reliability, availability, maintainability, safety, and quality of the product.

The reliability of a product can be described as how well the product is performing its functionality when it is operational. The reliability of the LSS software shall be 100%. The software shall always execute the correct pieces of code at the correct moment to ensure the system functions in a 100% predictable manner. When the LSS software has a 100% reliability, a 100% reliably walking Lunar Zebro is not guaranteed as this depends on much more than software. The software can only guarantee the same behaviour under the same circumstances.

Availability describes how much of the time the system is available to perform its job. The availability of the LSS software shall be 100% as well. From the moment the LSS software comes online, until Lunar Zebro is shut down, it shall be fully operational. A major factor that can cause a lower availability, is radiation from the lunar environment. If the LSS software is not able to properly solve or mitigate errors caused by radiation during operation, a system reset will be required, lowering the availability.

The maintainability of the LSS software is different for Lunar Zebro compared to DeciZebro. When it comes to Lunar Zebro, the LSS software does not need to be maintained since the product is shipped and no other products are expected to be shipped afterwards. However, if this thesis wants to contribute to DeciZebro, the LSS software should be easily maintainable. The use of Dezyne is expected to increase the maintainability of the project. It clearly separates code for state machining from implementation specific code and helps create separate frameworks for handwritten code to be integrated in.

The safety of Lunar Zebro is partly covered by the LSS software. The software shall ensure that the LSS is not used outside of its designed operating specifications. The operating specifications that are important to protect the LSS from damaging itself are temperature and current specifications. The LSS can monitor both. The LSS software shall protect the LSS against drawing too much current. The LSS software does not have to protect itself against over- or under-temperature. The LSS shall only measure temperature and report this back to the top level controller running. This allows the OBC to take the decision to operate the LSS in a broader temperature range if deemed worth the risk of damaging Lunar Zebro.

The quality of the product depends on whether it fulfils the requirements listed. Fulfilling the highest priority requirements satisfies the desired quality of the product. Fulfilling lower priority requirements adds to the quality of the product but is not strictly required.

3.8. Configuration Requirements

The Configuration Requirements specify the composition of the LSS software. A list of Configuration requirements for the LSS software:

[Conf-1] The LSS software shall have one data interface accepting commands and giving feedback. (*)

[Conf-2] The LSS software architecture shall have a framework in which the locomotion algorithm can be placed. (*)

[Conf-3] The LSS software architecture shall account for a framework in which error solving strategies can be implemented. (**)

The OBC shall be able to command the LSS. This requires the LSS software to have at least one communication interface to control the system. All functionality and data shall be available via this interface.

There is a wish to integrate the locomotion algorithm developed in [31]. This shall be done by implementing the algorithm in its own separate framework to keep the code maintainable during development. This framework shall be set up such that it can be updated without the rest of the LSS having to be updated.

The implementation method for the locomotion algorithm is also applied for implementing possible error solving strategies. Whenever the LSS detects a fault, the system shall try to solve or mitigate the error. In the first iteration of the LSS software, the error solving strategy can only consist of a resetting function. However, if the schedule of the project allows for more research, implementing different error solving strategies should be among the top priorities. These solving strategies can range from mitigating overheating of the motors by a different type of intermittent use, to creating walking patterns with less than six legs. To have the opportunity of incrementally improving the LSS in this way, this error solving framework is required.

3.9. Design Requirements

Design requirements specify the use of standards, components or margins etc. A list of Design requirements for the LSS software:

[Des-1] The LSS software shall be developed using the Dezyne tool. (*)

[Des-2] The LSS software shall be written in C/C++. (*)

[Des-3] The LSS software should be documented using Doxygen. (***)

The goal of this thesis is to deliver a reliable software implementation for the LSS. To reach this goal, a model-driven design approach using Dezyne shall be used. Dezyne is a model-driven design tool that can also formally verify a model and generate source code. These functions both reduce the probability of human errors ending up in the final design of the software and thus contribute to reaching the goals of this thesis.

All software written as part of the Lunar Zebro project shall be written in the C or C++ language since this is the de facto standard of embedded programming. However, Dezyne offers considerably more support for C++ compared to C since it makes extensive use of lambda-functions. The code should thus be written in C++ as much as possible to maximise compliance with existing standards.

For documenting all code, an industry standard like Doxygen is suggested.

3.10. Verification Requirements

Verification requirements ensure verification is performed in the correct manner. A list of Verification requirements for the LSS software:

[Ver-1] Lunar Zebro shall be tested for its walking speed in Earth's gravity field in different types of sand. (**)

[Ver-2] Lunar Zebro shall perform all functions described in by the functional requirements during tests while subject to the environment on Earth. (*)

[Ver-3] Lunar Zebro should perform all functions described in by the functional requirements during tests while subject to an environment similar to that of the moon. (***)

Dezyne can verify a system is without deadlock or livelock. It cannot verify if a system fulfils a specific functional requirement. Therefore, functional requirements should be verified by subjecting Lunar Zebro to various functional tests in different environmental settings. These functional tests are: safely starting and stopping the LSS, standing up and laying down, walking at different speeds in all directions and lastly, deploying the solar panel.

These system tests should be performed with Lunar Zebro in a thermal vacuum chamber. It can also be useful to place Lunar Zebro within a radiation beam to observe effects induced by radiation. If the LSS is properly designed, no unwanted behaviour will be observed. However, if the Lunar Zebro team is not able to test the rover under extreme environmental circumstances, functional tests in earthly conditions will have to suffice.

4

System Design

The LSS software is designed using the Dezyne tool which can verify and visualise system designs and subsequently generate source code from the verified design. To do this, Dezyne uses its proprietary language. This language adheres to a strict terminology to describe a system.

Outline This chapter describes the LSS software design using Dezyne terminology. It explains the necessary terminology to understand the system, gives an overview of the system and explains the implementation in detail. Lastly, it covers the system verification done using Dezyne.

4.1. Dezyne Terminology

The Dezyne language is a small language of 29 keywords [14]. Some of these keywords are at the same time regarded as a concept to describe and reason about a system [12]. To start the description of the LSS software, the most important concepts to understand are `system`, `component` and `interface`. These three concepts also happen to be keywords in the language.

System A system is defined as a collection of components and their interfaces. A system does not define specific behaviour, it merely defines which components are connected to each other via which interfaces. Listing 1 shows a system from a Dezyne tutorial. Lines 3 and 4 initialise the two components that are in the system. A `led` component of type `LED` and a `switch` component of type `Switch`. The definition of the component types starts at lines 9 and 24 respectively.

After defining the components in the system, the ports are connected. Both components have one port that is of the same interface type: `iLed`. The interface `iLed` connects the switch and the LED.

Component A component defines the behaviour of a specific part of a system. This can be an arbitrarily large or small part of the system, depending on the choice of the designer. Every component has at least one port to connect it to another component or system via an interface. This port can be a `provides` or a `requires` port. A `provides` port is defined as a port at the top of a component. Data that is sent downwards on this interface, is defined as going `in` to the component. Data being sent upwards by the component is regarded as going `out` of the component.

The opposite of the `provides` port is the `requires` port. This port is always located at the bottom of a component. Data going out of the component, travelling downwards over this interface, is defined as `in`. Data coming in to the component, moving upwards over the connected interface, is regarded as `out`. Although these definitions might not seem trivial at first, they do help to keep track of the direction of data.

Listing 1 shows `LED` provides a port and `Switch` requires a port. Figure 4.1 displays the result of these definitions, with `Switch` having it's port at the bottom of the component and `LED` at the top.

The behaviour of a component shall list a response to *all* possible incoming events. These events are specified by the connected interfaces. In the case of listing 1, the connected interface defines two events: `turnOn` and `turnOff`. Currently, the `LED` component has no responses implemented, as nothing is specified between the curly braces after both events. The component will just accept the events and not act upon it.

```
component LEDSwitchSystem {
  system {
    LED led;
    Switch switch;
    switch.iLed <=> led.iLed;
  }
}
component Switch {
  /* "requires" means calls ILED functions */
  requires ILED iLed;
}
interface ILED {
  in void turnOn();
  in void turnOff();
  behaviour {
    on turnOn: {}
    on turnOff: {}
  }
}
component LED {
  /* "provides" means implements ILED functions */
  provides ILED iLed;
  behaviour {
    on iLed.turnOn(): {}
    on iLed.turnOff(): {}
  }
}
```

Listing 1: "Simplest Usable Dezyne System Model" from the Dezyne Introductory Tutorial. [10]

Interface Interfaces define the communication allowed between precisely two endpoints. These two endpoints may be ports in components, or ports at the boundary of a system. They can be defined with or without states. Defining an interface without state means listing the communication that is possible on this link. Listing 1 shows the `ILed` interface that does just that. It lists two events that are always allowed to be triggered on the link. The checker will generate an error if a component tries to send an unknown command over the interface.

An interface can also be defined with state behaviour, Now, a command going over the interface can trigger a state transition within the interface. An example of this is shown in listing 2. The interface can be in two states that are listed in `enum State`. The state of the system is initialised to be idle.

All states shall be listed once within the square brackets. These statements are called `guards` and can be interpreted as a switch-case statement. In the idle state, an `on start:` event will start a timer and put the interface into the running state. An `on cancel:` event can happen, but since the timer is already idle, nothing needs to be done. The event could also be declared `illegal` if deemed appropriate. In the running state, the timer can either be cancelled or a timeout can occur. Regardless of when the timeout happens, it is known that it shall happen when no cancel event is generated. This is exactly what

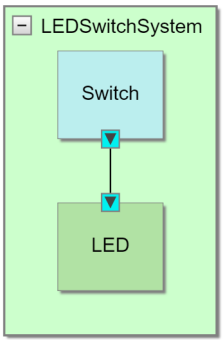


Figure 4.1: System view belonging to "Simplest Usable Dezyne System Model" generated by Dezyne.

```
interface ITimer {
  enum State { Idle, Running };
  in void start();
  in void cancel();
  out void timeout();

  behaviour {
    State state = State.Idle;

    [state.Idle] {
      on start: state = State.Running;
      on cancel: {}
    }
    [state.Running] {
      on start: illegal;
      on cancel: state = State.Idle;
      /* on inevitable ALWAYS happens eventually, unless another event
         happens first. In this case, only the cancel event can happen
         first. */
      on inevitable: {
        state = State.Idle;
        timeout;
      }
    }
  }
}
```

Listing 2: Timer Interface from the Dezyne Introductory Tutorial

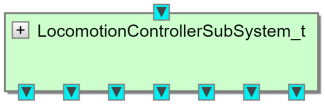


Figure 4.2: LC Sub-System showing the interfaces connected to the edge of the system. One provides port for communication with the OBC and seven requires ports to communicate with all motor drivers.

`on inevitable:` does. When no other event is triggered, the verifier will trigger this event. Starting the timer in the running state is `illegal` and will generate an error during verification and an assert in code.

When the `ITimer` interface is attached to a component, the component is forced to keep track of the same states. If the component does not do this, it will try and start the timer while it is already running, generating the assert. The verifier in Dezyne will alert the user of this flaw when the model is verified. In short, both components at each end of an interface need to switch state synchronously in order for the verification to succeed.

4.2. System Overview

As described in section 2.3, two distinct software parts make up the LSS of Lunar Zebro. The first part is the Locomotion Controller (LC) which is running as a separate thread on the OBC of Lunar Zebro. This controller receives commands from the top level controller running on the OBC and translates them into useful commands for the motor drivers. For readability, the LC will receive commands from the OBC instead of the top level controller running on the OBC. The communication channel between the OBC and the LC is modelled as an interface from the boundary of the LSS system toward a LC component. The communication channels between the LC and Motor Drivers are modelled as individual interfaces from the LC to the boundary of the system. The LC System with one provides port and seven requires ports is shown in fig. 4.2.

The second part of the LSS software, is the part running on the leg module. Commands from the LC are translated into movement of the motor or answered with requested data. Every motor driver has its own conceptual interface with the LC. In reality all motor drivers are connected to a bus on which the LC is the master and the motor drivers are slaves.

To gain a better understanding of the functionality of the LC, the interfaces connected to the boundary

of the system are discussed in depth. The provides port of the system shown in fig. 4.2 will be attached to the OBC, via the OBC Interface. The seven requires ports are for the leg modules and the solar panel motor driver and will be connected via the Motor Driver Interface. Both interfaces will be discussed next.

4.2.1. OBC Interface

The OBC opens different threads and communicates with them via sockets. The LSS is one of those threads. The OBC implementation requires synchronous communication from all sub-systems. Therefore, the LSS will only speak when spoken to. The list of commands or events that are accepted and generated by the LSS software:

1. Receive Messages from the OBC (asynchronous)
   (a) Start Motor Control
   (b) Stop Motor Control
   (c) Stand Up
   (d) Lay Down
   (e) Walk (direction, heading, speed)
   (f) Request data (data requested / heartbeat)
   (g) Emergency Stop
2. Send Messages to the OBC (synchronous)
   (a) Reply requested data (requested data / heartbeat)

Interface Functionality for Driving To explain the choice of walking functionality the OBC is able to call on the LSS, the analogy with a car is made as an addition to the functional requirements set in section 3.1.

Before one can drive a car, it must be started. For Lunar Zebro this is translated to turning on the motor control so the motors hold the current position of the legs regardless of what this position is. Then, before driving can commence, the car should be put into gear, or be 'unparked'. In the Lunar Zebro case, this means standing up.

While driving, a car is controlled via pedals and a steering wheel to influence speed and direction. Lunar Zebro requires the same functionality which is integrated in 1 function, `walk`. This function takes the arguments of *direction*, *heading* and *speed*. Direction is either forward or backward while heading is a turning angle with a maximum of 90° clockwise or counterclockwise. Speed will be an incremental measure that will have a slow, medium and fast speed. A speed of zero means Zebro is standing still.

In the event of a failure during operation, Lunar Zebro shall immediately halt the system. In a sense it can be viewed as the ABS functionality of a car. This is the Emergency Stop command. It shall be present to give the OBC the power to shut down the entire system without ordering the Power sub-system to cut the power to the LSS. Both the Emergency Stop command and cutting the power to the LSS are last-resort measures. However,they are necessary when unexpected failures occur and high currents are flowing through the system. When the LSS is fully mature and is guaranteed to function safely, this command can be abstracted away from the user (OBC) much like ABS braking in modern cars is often triggered by radar systems in front of the car and not by the user pressing the brake pedal. However, this mission still requires the OBC to have total control over all sub-systems and this is the rationale of leaving this functionality accessible.

After driving to a destination, it is time to park the car (Lunar Zebro lays down) and turn off the car (Zebro's motor control is turned off after the last command is executed and the motors do not actively hold their position anymore). Turning the motor control off saves energy. Parking Lunar Zebro can also happen before the destination is reached, if for some reason Zebro should wait for a bit. No traffic lights or traffic jams are expected on the moon, but this 'BlueMotion' [52] type of system can potentially save energy.

Lunar Zebro differs from a car in one major aspect: it has legs instead of wheels. For energy efficient walking, the position of the legs shall be known at all times. This is solved by the leg module itself and the LC does not have to worry about this.

Other Interface Functionality The LSS shall be able to keep the OBC informed of its status and if necessary supply further details. To establish this over a synchronous communication channel, the OBC must always ask for certain data and wait for a reply.

It is possible for the OBC and the LSS to lose their connection. Both systems shall be able to detect this and halt their activities as a result. To detect a lost connection, frequent communication between the two systems is required. Any command coming in from the OBC indicates a functioning communication channel. If the OBC does not need to send a command for some time, a heartbeat command shall be sent to indicate to the LSS that the communication channel is still functioning. To increase the efficiency of the channel, the heartbeat is implemented as a request for a system status packet of the LSS.

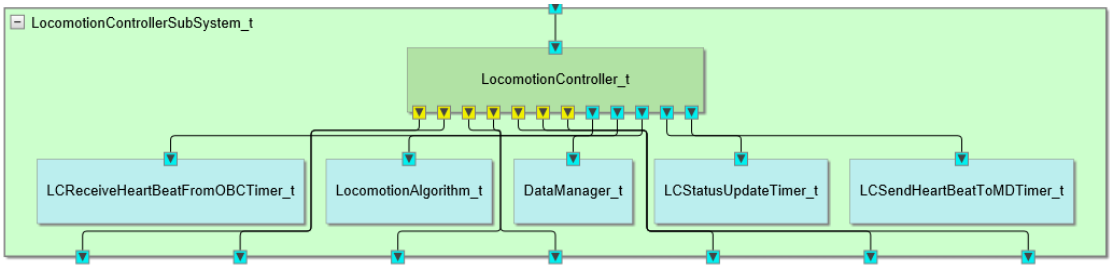


Figure 4.3: LC Sub-System showing all components and interfaces.

4.2.2. Motor Driver Interface

The LC is the sole master on an RS485 bus that connects seven motor drivers. Six of these motor drivers are driving Lunar Zebro's legs and one is actuating the solar panel. The slaves cannot communicate asynchronously to the master on this bus. The list of commands or events that are accepted and generated by the motor drivers:

1. Receive Messages from the LC (asynchronous)
   (a) Start Motor Control
   (b) Stop Motor Control
   (c) Rotate Motor Forward (position, delta time)
   (d) Rotate Motor Backward (position, delta time)
   (e) Request data (data requested / heartbeat)
   (f) Emergency Stop
2. Send Messages to the LC (synchronous)
   (a) Reply requested data (requested data / heartbeat)

Interface Driving Functionality The functionality of the LSS that can be called by the OBC, results in a set of commands that the LC shall in turn be able to call for every leg module. The LC shall be able to turn motor control on or off for every motor driver. This is necessary for the LSS in different states to guarantee the lowest current consumption.

Making Lunar Zebro walk can be done in many different ways. Research done in the past years at TU Delft indicates Max-Plus algebra has great benefits controlling the locomotion of a Hexapod [29, 31, 36]. At the moment further research is done on this Locomotion Algorithm within the terrestrial Zebro Project and the knowledge gained will be used in this implementation of the LC.

Using Max-Plus algebra, the commands preferred to rotate a leg are `Rotate Motor Forward` and `Rotate Motor Backward`. These commands have two arguments. The first argument is a position in degrees, 0°-360°. The second argument is an amount of time the leg module gets allotted to execute a movement to the previously specified position. This way the Locomotion Algorithm has direct control over the direction, final position and speed of the movement.

Other Interface Functionality Since all relevant data of the LSS originates from the leg modules, the LC shall be able to call the same `request data` functionality from the leg modules as the OBC can call from the LSS. As all leg modules are connected on a bus, this shall be synchronous communication initiated by the LC.

Analogous to the communication channel of the OBC interface, communication between devices can be lost. To detect this, the same mitigation strategy as employed in that interface is used. Any communication received by the leg module from the LC is interpreted as the channel still being functional. In the event of the LC not having to send any data to the leg modules, the leg module's status is periodically polled, resembling a heartbeat.

4.3. Dezyne Implementation

The functionality on the boundaries of the system shall be translated to functionality within interfaces and components in Dezyne. The result of this is shown in fig. 4.3. The reasoning behind this design is further explained in this section. First, the states used for the system are explained, since they influence the rest of the design. Secondly, the implementation of the interfaces detailed in section 4.2 is discussed. Additionally, other internal interfaces that are created are discussed. Lastly, the choice of components and their implementation is discussed.

4.3.1. System States

Dezyne can handle many states and does not require the user to define as few states as possible to optimise for verification time. The user is completely free in choosing the number of states.

The LSS can be in one of four states. These states are chosen because they represent the basic functionality of the system. Extra necessary state machining can be implemented by the use of state variables inside each state. For example: a Boolean that is true when motor temperatures are within operating range and false when they are outside. Checking this Boolean in one of the four states essentially creates sub-states. The four states are:

1. `Idle`
2. `Operational`
3. `Fatal Error`
4. `Solve Errors`

Idle State The idle state is the starting state and shall be guaranteed to be safe. In this state it is not possible to use functionality of the system that draws large currents. It is possible to remain in contact with all components of the system. From the idle state it is possible to stay in this state, move to the operational state or move to the fatal error state.

Operational State The operational state can only be reached from the idle state. This state incorporates all activities of the LSS that consume a lot of power. Additionally, the state also contains all functionality that is present in the idle state. From the operational state it is possible to stay in this state, move to the fatal error state or move back to the idle state.

Fatal Error State The fatal error state is reachable from any state when a severe error occurs. This state has the same functionality as the idle state. However, the only exit from this state is towards the solve errors state. This state transition is initiated by a command from the OBC. Because of this, the behaviour after a fatal error is defined and subsequent actions are in control of the OBC.

Solve Errors State Requirement [Conf-3] requires the presence of a framework in which errors can be solved. This is the purpose of this state. The state can only be reached after the OBC has given a command to move from the fatal error state, to this state. In this state, a full system reset can be executed as the bare minimum of error solving strategies. Other error solving strategies can be implemented at every point in time after the development of the system. When an error solving strategy is not successfully executed, it can be decided to move back to the fatal error state and try again afterwards. If errors are successfully solved, the system can move back to the idle state.

4.3.2. Interfaces within the System

As mentioned in section 4.1, interfaces can be defined with or without state. It is important to realise there are interfaces between only Dezyne components and interfaces that interact with the boundaries of the system. The interfaces that interact with the boundary of the system have special requirements.

The interfaces within the system that are only connected between Dezyne components have no special requirements. They may use the assumptions that Dezyne has a single threaded, run-to-completion execution. These interfaces may also assume communication never fails and is guaranteed to be synchronous. Because these interfaces may make these assumptions, it is beneficial and a good practise to define these interfaces with state, if required.

OBC Interface The OBC interface connects the `provides` port of the LSS sub-system to the `provides` port of the `LocomotionController_t`. This interface is implemented with the functionality as described in section 4.2.1. Because this interface is connected to the boundary of the system, it is designed to be state-less. This ensures the LC and OBC can have asynchronous state changes.

Motor Driver Interface In this interface, the functionality as described in section 4.2.2 is implemented. Again, as with the OBC interface, the interface is state-less because it is connected to the boundary of the system. This allows the LC and the motor drivers to switch states asynchronously.

Timer Interface Currently, there are three timers in the design. Their interfaces are the same as shown in listing 2. The interface has two states to indicate the timer is running or not. The functionality that is available is a start timer command, a stop/cancel timer command, and a timeout outgoing event. The state machine for this timer is straight forward and explains itself in listing 2.

If this interface were to be connected to a timer implementation that could not guarantee single-threaded, run-to-completion execution, a stopping state is required. This is to prevent a timeout occurring while a cancel event is being processed. All details of this type of implementation can be found in Dezyne's `external` tutorial [11]. Dezyne has a built-in timer functionality that guarantees single-threaded run-to-completion behaviour, allowing the interface to be the same as in listing 2.

Locomotion Algorithm Interface Requirement [Conf-2] states that the LSS shall account for a framework in which the Locomotion Algorithm from [31] can be implemented. This framework is a handwritten component called `LocomotionAlgorithm_t`.

The Locomotion Algorithm needs the following functionality in its interface to perform its job correctly:

- `in` Events
  - Switch Locomotion Algorithm on and off.
  - `Walk`, `Stand Up` and `Lay Down`.
  - Receive current position of every leg module.
- `out` Events
  - `Rotate Motor Forward` and `Rotate Motor Backward` command for every leg module.
  - Request current position of every leg module.

The Locomotion Algorithm has two states over which the functionality is distributed. The first state, idle, allows the command to turn the algorithm on. All other commands are flagged as `illegal`. The second state, operational, allows all commands except the command to turn the Locomotion Algorithm as it is already on.

Data Manager Interface The data manager stores all information gathered by the LSS and makes it accessible for any component in the system. The interface is state-less and has three events: push data, pull data and return data. The arguments of the functions indicate what data is actually requested. The interface implementation in Dezyne is shown in listing 3. As Dezyne does not understand data types, the

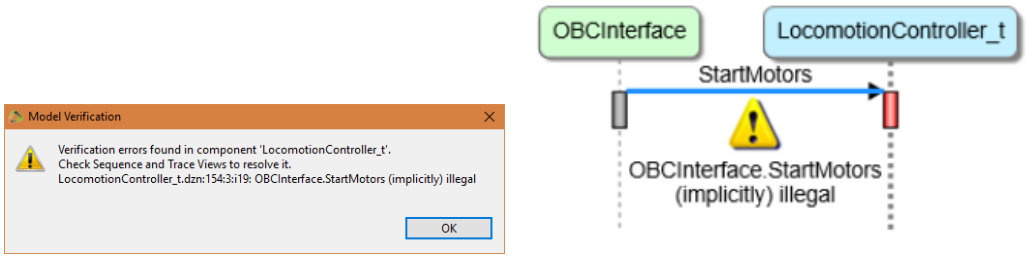
```
1  extern Device $std::string$;
2  extern DataType $std::string$;
3  extern Data $int$;
4
5  interface DataManagerInterface_t
6  {
7    in void pushData(in Device device, in DataType dataType, in Data
       ↪ data);
8    in void pullData(in Device device, in DataType dataType);
9    out void returnData(in Device device, in DataType dataType, in Data
       ↪ data);
10
11   behaviour
12   {
13     on pushData: {}
14     on pullData: {returnData;}
15   }
16 }
17
18 component DataManager_t
19 {
20   provides DataManagerInterface_t DataManagerInterface;
21   // Handwritten
22 }
```

Listing 3: Dezyne Implementation of the Data Manager Interface and Component.

data manager can not really manage data. It functions as a guard to ensure no race-conditions are present when accessing data.

4.3.3. System Components

The system shown in fig. 4.3 consists of six components. A component in Dezyne defines `ports` and `behaviour` [12]. The components in fig. 4.3 have two different colours: blue and green. The green components are components with functionality defined in Dezyne. The blue components on the other



(a) Warning when Dezyne finds an invalid trace.

(b) Sequence view of the events that causes the invalid trace.

Figure 4.4: Verification Error.

hand, have no functionality in Dezyne. These are handwritten components. Handwritten means they are implemented in native code, by the user of Dezyne, after the system is designed.

Handwritten Components The Dezyne implementation of the data manager is shown as an example of a handwritten component in listing 3. In Dezyne it is merely an interface definition inside an otherwise empty component definition. The functionality of the handwritten components is solely determined by the functionality in the interface. The handwritten code is responsible for its own correct functioning. If it triggers the wrong events at the wrong times, the interface will generate an `assert` and fail. During the code integration, these handwritten components are discussed in depth.

The data manager component is created to collect all functionality regarding data storage. The interface allows the system to command the data manager to retrieve or store information. The method of storage can change, but if the interface stays the same, the system will be unaffected.

Next, there is the Locomotion Algorithm component. The functionality of this component is handwritten in [31]. From this implementation, an interface is constructed that has all the necessary functionality. This will allow the component to change over time, without affecting the entire LSS.

Lastly, there are three timers with an individual task. The OBC-heartbeat timer generates an event for the LC to check whether or not communication was received from the OBC. The status update timer will generate events for the LC to indicate the system status must be updated. The motor driver heartbeat timer generates an event for the LC to indicate a heartbeat must be send to a motor driver.

Locomotion Controller Component The LC component consists solely of functionality described in Dezyne's language. The LC component only provides one port. This port is used to establish an interface with the OBC. All communication described in section 4.2.1 is allowed on this interface and its functionality is implemented in the behaviour of the LC component. On the other hand, the LC specifies 12 requires ports. Seven of these ports are for the motor drivers present in the system. The other five ports are connected to the handwritten components with their respective interfaces, both described earlier. All the functionality described by the interfaces is listed with an appropriate response in all states shown in section 4.3.1.

In all states, the LC keeps track of whether or not it has had communications with all connected devices. If the OBC or any of the motor drivers were to stop communicating with the LC, the system will move to the fatal error state.

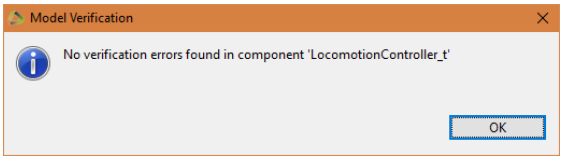
4.4. Dezyne Verification and Simulation

After conceptually designing the system, the verifier and simulator can be used.

4.4.1. Verification

Verification is done by simply pressing a button. When part of the system is not correctly defined in Dezyne, an error will occur and a helpful trace will show where the error occurred. This is shown in fig. 4.4. In this case, the `OBCInterface` allows an event called `StartMotors` that is not allowed inside the component `LocomotionController_t`. From here, one can look in the definition of `LocomotionController_t` for the problem. Another option is to check if the interface is not mistakenly allowing the `StartMotors` event. However, since the `OBCInterface` is without state, the event must be defined and is thus allowed in any state of the LC.

If the verification is completed successfully, the message in fig. 4.5a is shown. Afterwards, the result is shown in fig. 4.5b. Here Dezyne shows which interfaces were checked during the verification and it shows



(a) Message showing verification succeeded.

LocomotionController_t.dzn

Check	Action	Time	States	Transitions	Done	Result
OBCInterface_t						
Deadlock		0:00	3	10	100%	✓
Livelock		0:00	3	10	100%	✓
MotorDriverBusInterface_t						
Deadlock		0:00	3	10	100%	✓
Livelock		0:00	3	10	100%	✓
StatusInterface_t						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
TimerInterface_t						
Deadlock		0:00	5	7	100%	✓
Livelock		0:00	5	7	100%	✓
LocomotionAlgorithmInterface_t						
Deadlock		0:00	22	49	100%	✓
Livelock		0:00	22	49	100%	✓
DataManagerInterface_t						
Deadlock		0:00	3	4	100%	✓
Livelock		0:00	3	4	100%	✓
LocomotionController_t						
Deterministic		1:01	122195	147722	100%	✓
Illegal		1:01	122195	147722	100%	✓
Deadlock		1:01	122195	147722	100%	✓
Livelock		1:01	122195	147722	100%	✓
Compliance		0:02	122195	147722	100%	✓

(b) Result of verification showing the checked interfaces and components as well as the number of states checked. Dezyne version 2.7.2.

Figure 4.5: Verification complete.

the verified component. The current number of states in the LC is 122195. This is quite a lot considering the four states described in section 4.3.1. The number can be explained by, amongst others, remembering that in every state, every motor driver could have had contact with the LC, or not. A Boolean keeping track of this creates a sub-state in the current state. In this manner, the model grows exponentially. A completed verification ensures deadlock and livelock are not present in the interfaces and component. These are two of the most important aspects of a system. Additionally, Dezyne checks if all events are uniform over interfaces and components. Replies and returns of events are also checked for their type. A complete overview of what is checked by Dezyne can be found on their website [9].

4.4.2. Simulation

Simulation is essentially a human-triggered verifier. The simulation functionality in Dezyne allows the user to select an event to be triggered next from a list of possible events in the current state. During verification, the verifier does this as well, but faster and with all possible combinations of events!

An example trace is shown in fig. 4.6. This trace shows how the OBC commands the LC to turn the motor drivers on and start walking. During walking the connection between OBC and LC is lost and a timeout occurs from the timer that is monitoring communication with the OBC. Because of this, locomotion is stopped, all motor drivers are put in their emergency states and lastly the LC also moves to the emergency state.

Because of the simulator it is easy to visualise whether or not the system is behaving as expected. One could also test this in real life, but doing a simulation helps spot errors early on in the process.

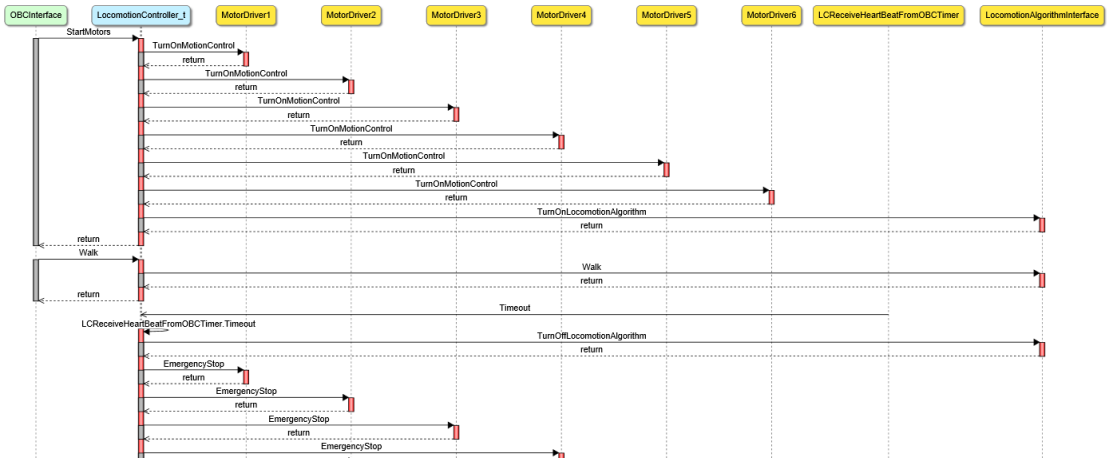


Figure 4.6: A simulation trace showing the OBC giving commands and a timer time out.

5

Code Generation and Integration

When the system design in Dezyne satisfies the requirements, it needs to be transformed to code that can run on the target device. This device is the OBC of Hyperion Technologies B.V., shown in fig. 5.1, which runs a Linux distribution. Programming for Linux gives great freedom in the choice of source code language. However, this freedom is limited by Dezyne. In theory Dezyne supports, amongst others C, C++, Java, JavaScript and Python. In practice, C++ has better support than the other options. This has partly to do with the fact that the generated code can rely heavily on the Lambda expressions used in C++. These Lambda expressions are essentially inline functions that can easily be passed to other functions [5]. This makes sense when realising that a trace of events in Dezyne can be represented as a specific order of function calls. With Lambda expressions, these function calls become very readable and easy to implement by the user. Because of these advantages, C++ code will be generated from the Dezyne model.

When code is generated, it must be integrated in the native code base. This integration connects events triggered by handwritten code, to the correct events in Dezyne's generated code and vice versa. Additionally, the functionality required from the handwritten components and the leg module is implemented.

Outline This chapter describes the generation of C++ code from the Dezyne model. It explains the concept of skeleton definitions needed for integration. Next, it shows how events are mapped on both the OBC- and motor driver interface of the LC. Lastly, the integration is described for all handwritten components and the development of the leg module is discussed.

5.1. Code Generation

Generating code from Dezyne can be done via a button in the user interface, or from command line. The language and target location must be specified, as well as which files to export. Now, Dezyne generates source code file-to-file. The result of this is a `.cc` and `.hh` file for every `.dzn` file. If a file with the specific file name would already be present, Dezyne does not overwrite this file for safety reasons.

Figure 5.1: The OBC or, CP400.85 processing platform, of Hyperion Technologies B.V. [8]

Skeleton Definitions To ease the integration of handwritten components, Dezyne generates skeleton definitions for all handwritten components. The placement of these definitions varies depending on how the user handles the file-naming and which files are used for code generation. In this case, the generated `.hh` file containing the system component, also contains a `struct` defined inside the namespace `skel` for every handwritten component. In a separate handwritten file, a class can be declared that inherits from the skeleton class defined by the Dezyne generated code. To do this, an include of the `.hh` file containing the system component is placed in the handwritten `.cc` file. No includes may be placed in the handwritten `.hh` file as this results a conflict.

The user can define the *content* of the events, without having to worry about how Dezyne will call them. This is all taken care of by the skeleton definition. An example of this is shown in listing 5 defining the Data Manager implementation.

Thread Safe Shell An extra option when generating code, is to generate a thread safe shell for the system component of the model. A thread safe shell ensures external events are placed in a queue and executed in their arrival order [15]. This is necessary to maintain the single-threaded run-to-completion criteria. It is especially helpful in a system where different communicating threads are present. The thread safe shell essentially eliminates race-conditions.

Runtime Beside generating code, the Dezyne *runtime* is downloaded from the Verum servers and added to the project. The runtime contains important functionality like a locator library that distributes all objects throughout the project. Because of this locator, it is possible to call any object, anywhere in native code.

Another important library contains the *pump* functionality. This library ensures all events in the system are properly queued for single-threaded execution. Additionally, the *pump* library contains an implementation for timer functionality.

5.2. Code Integration

After generating the code, a main file is written in which the Dezyne functionality is initialised. Dezyne can also generate a main file if necessary. However, this main file is meant to check all bindings of the system and output traces of events to terminal. This makes the generated main file ill-suited for this project. It is more suitable as a debug tool to check bindings.

The initialisation of Dezyne's generated code consists of setting the locator of the system object and connecting all events going out of the Dezyne system to handwritten functions. The handwritten functions are coupled to Dezyne events via the Lambda functionality in C++. A shortened example of the initialisation is shown in listing 4. After connecting all the boundary events, it is recommended to use Dezyne's built-in `check_bindings` function. This function checks if there are any unconnected events at the edge of the system. The function can be omitted, but it is recommended for production code to always be sure all events are connected.

The LC design is integrated on a Raspberry Pi (RPi) Zero Wireless. The RPi's UART is connected to two RS485 chips to allow communication with the leg modules over the RS485 bus.

5.2.1. Integration of System Boundary Interface Events

The events going in and out of the LSS system as shown in fig. 4.3, are all part of a communication interface with higher level (OBC) or lower level (leg module) hardware.

The OBC interface is implemented as a socket in Linux. Commands on this interface are directly mapped to Dezyne event calls. Whenever data is requested by the OBC, data is replied in SI units or another sensible unit as per requirement [O-Int-4]. The requesting party must know what it is receiving to interpret the data correctly.

The motor driver bus-interface is an RS485 bus. To be able to communicate over this bus, a protocol is required. An industry standard is chosen: the MODBUS-ASCII protocol.

RS485 Communication The MODBUS-ASCII protocol is set up as follows [39]. The START of a packet is indicated by a colon or, in HEX, `0x3A`. The end of a packet is indicated by a carriage return together with a line feed (\r\n). In between, only hexadecimal characters are allowed, meaning characters 0-9 and A-F. An example packet is shown in table 5.1.

Two alterations are made regarding this protocol. First, instead of sending both \r and \n as STOP-characters, only \n is sent. This is done to ensure better compatibility with line parsing functions already present in Linux.

Second, the data portion of the packet has a fixed length instead of n chars. A fixed length protocol is preferred because buffers that receive messages can now be of fixed length. Overflow of these buffers is

```
#include <dzn/locator.hh>
#include <dzn/runtime.hh>

LocomotionControllerSubSystem_t& init_dzn_functions(void)
{
  static dzn::locator loc;
  static dzn::runtime rt;
  static dzn::pump pump;
  static LocomotionControllerSubSystem_t lc(loc.set(rt));

  lc.MotorDriverBusInterface1.in.EmergencyStop = [&] ()
   ↪ {EmergencyStop("motorDriver1");};
  lc.MotorDriverBusInterface1.in.ResetDriver = [&] ()
   ↪ {ResetDriver("motorDriver1");};
  lc.MotorDriverBusInterface1.in.TurnOnMotionControl = [&] ()
   ↪ {TurnOnMotionControl("motorDriver1");};
  lc.MotorDriverBusInterface1.in.TurnOffMotionControl = [&] ()
   ↪ {TurnOffMotionControl("motorDriver1");};
  lc.MotorDriverBusInterface1.in.RotateMotor = [&] (int8_t direction,
   ↪ uint16_t position_setpoint, uint16_t delta_time)
   ↪ {RotateMotor("motorDriver1", direction, position_setpoint,
   ↪ delta_time);};
  lc.MotorDriverBusInterface1.in.RequestData = [&] (std::string
   ↪ device, std::string dataType) {RequestData(device, dataType);};
  lc.OBCInterface.out.ReplyData = [&] (std::string device,
   ↪ std::string dataType, int data) {OBCReplyData(device, dataType,
   ↪ data);};

  lc.check_bindings();

  return lc;
}
```

Listing 4: Shortened initialisation of Dezyne's generated code. All events for OBC and leg module 1 are connected to handwritten functions via the Lambda functionality of C++.

…	Start	Address	Command	Data	CRC	Stop
Length	1 Char	2 Chars	2 Chars	n Chars	2 Chars	2 Chars
Data	':'	12	214	0	111	'\r\n'
Hex	0x3A	0x0C	0xD6	0x00	0x6F	0x0D & 0x0A
Send	0x3A	0x00 0x43	0x44 0x36	0x00 0x00	0x36 0x46	0x0D 0x0A

Table 5.1: Example packet of the MODBUS-ASCII protocol.

defined when a certain amount of messages is not handled in time. An overflow of these buffers can be logged, while the oldest message is deleted.

During communication, bytes are received and placed in a buffer. If a STOP character is received, it is placed in the buffer and the counter `receivedMessages` is incremented to indicate a packet is received. If `receivedMessages` is non-zero, packets are parsed by extracting a fixed number of bytes from the buffer. If the first byte retrieved from the buffer is not a START-character, the packet is deemed lost. The buffer is then cleared until a START-character is found, or until the buffer is empty. If a STOP-character is retrieved from the buffer before the packet size is reached, a byte from the packet is deemed lost and the packet is discarded. When the calculated CRC does not match the received CRC, the packet is discarded.

To keep packet parsing as simple as possible, packets are dropped when they seem incorrect and no message is send back to indicate the package is dropped. The benefit of this that the bus is not overcrowded and the packet parsing function is straightforward. The communication on this bus is fully asynchronous. The LC must periodically monitor if the desired commands are actually executed. If this is not the case, commands can be re-sent or error solving can commence. This method of communication is safe as the leg modules can go into an emergency mode asynchronously and protect themselves in this way. This completely removes the need for intensive synchronous communication.

5.2.2. Integration of Handwritten Components

The functionality of handwritten components *inside* Dezyne is implemented according to Verum's specification [13]. Unfortunately, [13] is not completely up-to-date. What follows is the description of a working integration method, after consulting Verum.

A source- and header file for every handwritten component is created with the exact name of the Dezyne component. The source file of the handwritten component includes the header file of the generated Dezyne system file. This enables the use of the skeleton definitions defined in this file. The handwritten component inherits the interfaces defined in the skeleton definition. The last step in the process is to implement the functionality of the events of these interfaces.

Data Manager In the Dezyne implementation, extensive use is made of the `extern` keyword as can be seen in listing 3. This keyword gives Dezyne the possibility to handle data types that cannot be defined inside Dezyne, as Dezyne has no provisions for this.

The data manager interface describes two events: a data-push and data-pull event. These functions have strings as their parameters that point to objects. This is shown in listing 5. These objects are created for every device present in the system and store all data concerning that device. The objects also include get- and set functionality that can easily be called by the push and pull data functions. These objects are defined in `Types.hh`.

Currently, the data manager does not include logging functionality. It only retains one value for every defined device metric.

```
#ifndef DATAMANAGER_T_HH
#define DATAMANAGER_T_HH
class DataManager_t : public skel::DataManager_t {
public:
DataManager_t(const dzn::locator& loc);
  void DataManagerInterface_pushData (std::string device, std::string
   ↪ dataType, int data);
  void DataManagerInterface_pullData (std::string device, std::string
   ↪ dataType);
};
#endif // DATAMANAGER_T_HH

---

#include "LocomotionControllerSubSystem_t.hh"
#include "Types.hh" // DeviceMap

DataManager_t::DataManager_t(const dzn::locator& loc) :
 ↪ skel::DataManager_t(loc)
{
}
void DataManager_t::DataManagerInterface_pushData (std::string
 ↪ device, std::string dataType, int data)
{
  getDeviceMap.at(device).setData(dataType, data);
}
void DataManager_t::DataManagerInterface_pullData (std::string
 ↪ device, std::string dataType)
{
  int data = getDeviceMap.at(device).getData(dataType);
  DataManagerInterface.out.returnData(device, dataType, data);
}
```

Listing 5: The `.hh` file and `.cc` file of the handwritten implementation of the DataManager.

Locomotion Algorithm The interface of the Locomotion Algorithm with its possible events, is described in section 4.3.2. The native implementation of all events is provided by Kinkelaar [31]. This implementation only needs a slight adaptation to fit the current design.

Timers Dezyne has provisions for timer implementations as part of the Dezyne runtime library `pump.hh`. Instantiating a pump object, allows to call the `handle` function, which starts a timer and defines a function call to a function when a timeout occurs. The `remove` function of the pump object stops/cancels the timer.

5.2.3. Development of Leg Module

The interface between the LC and the leg modules is well defined because of the use of Dezyne. Section 4.2.2 describes all functionality that the LC shall be able to call from the leg module. This functionality is implemented in C on the leg module board V1.0 as developed in [38].

As the motor driver bus-interface is without state, the motor driver is also developed without any state behaviour. Any command from the LC will always immediately be executed. The LC is fully responsible for the correct invocation of commands.

The communication is performed with the protocol described in section 5.2.1.

Emergency Stop The leg module performs an emergency stop by itself when its operating requirements are not satisfied. First, the leg module resets a watchdog timer every time a packet with the correct address is received from the LC. If a packet is not received every eight seconds, the watchdog timer is not reset and the leg module performs an emergency stop.

Secondly, the leg module monitors its power consumption when it is driving the motor. If too much power is consumed, an emergency stop is triggered to protect the motor driving hardware.

Third, an interrupt is triggered when a fault occurs on the motor driver chip. This interrupt performs an emergency stop and thus disables the motor driver chip.

When the leg module performs an emergency stop, it does not move to a special state. The LC can send commands directly after an emergency stop and they will be executed. If the operating requirements are again not satisfied, the leg module will again perform an emergency stop.

Absolute Leg Position The motors of Lunar Zebro do not have an absolute, but a relative encoder as explained in appendix A. Therefore, it is unknown at what position the legs are when Lunar Zebro boots. The legs must spin and pass a sensor in order to determine their absolute position. Having the LSS do this automatically at boot is decided to be undesirable. There are multiple ways to solve this issue on the leg module itself.

First, one can spin the legs back and forth hoping to pass the sensor located in the top half of the circle that is traversed by a leg. This requires the assumption that Lunar Zebro is lying on its belly on flat ground and the legs spin until they hit the ground on either side. Upon touching the surface, the rotation direction is reversed. This pattern is repeated until the sensor is triggered. Unfortunately, there are numerous problems with this scheme. Detecting when a leg touches the ground by measuring the current drawn by the motor is heavily dependent on the environment and the ageing of the motors. Also the assumption that Lunar Zebro is laying down on its belly is dangerous since the friction in the gearboxes is potentially high enough to keep the rover upright without the motors engaged.

Second, every leg can spin forward until the sensor is found. Backward is also possible, but this has the added risk of the rover digging itself into the ground. Unfortunately, having the legs rotating forwards will result a wobbling Lunar Zebro since all legs are probably in different starting positions. The chassis shall be able to handle these shocks, but the method is everything but smooth.

Third, another method is to accept the unknown position of the legs and just start executing walking commands. Whenever the sensor is found while moving, the motor driver re-calibrates its position. It will result in the same unwanted behaviour as in the second option, with the benefit of the LSS not having to have a special initialisation sequence. With this method, the system only needs to monitor whether or not every motor driver calibrates at some point in time.

The third option, although mostly the same as the second, is the preferred option since it does not require certain assumptions, reduces complexity of the LSS interface and continues operation even though no calibration is done in the event of a broken sensor. Walking with non-calibrated legs will most likely have a very bad efficiency, but at least the legs will be rotating.

Motor Control The motor control for the leg modules is implemented as described in Rouwen [47]. The implementation is adapted to support a Brush-Less Direct Current (BLDC) motor.

6

Results

The first goal of this thesis is to deliver a reliable LSS software design for Lunar Zebro. The second goal is to implement this software design on the LSS hardware of Lunar Zebro. The third goal is to list the advantages and disadvantages of the use of Dezyne during the design process.

To determine whether the first and second goal are reached, two intermediate steps are performed. The first and most important step is to revisit the requirements from chapter 3 and determine the design's compliance with them. The second step is to reflect on the reliability of the software and how much of this reliability can be attributed to Dezyne. The third goal is reached by listing the advantages and disadvantages of the use of Dezyne encountered during the design, along with a reasoning.

Outline This chapter describes the final product delivered by this thesis and tests it for its compliance with the set of requirements. The reliability of the software is evaluated. Lastly, the advantages and disadvantages of the use of Dezyne, as experienced during this project, are listed.

6.1. Reflection on Requirements

The set of requirements for the LSS software is revisited. If requirements are not fulfilled, a reason is given why this is not the case. If they are fulfilled, it is briefly explained how this is tested.

Functional Requirements

[Func-1] Partially fulfilled. A start command starts motor control in all leg modules, but does not initiate walking.

[Func-2] Partially fulfilled. A stop command stops motor control in all leg modules, but does not negotiate to lay down.

[Func-3] Partially fulfilled. All functionality is periodically tested on an office floor and is verified to work. However, the latest iteration of the software has not been verified.

[Func-4] Partially fulfilled. See requirement [Func-3].

[Func-5] Partially fulfilled. The design does not *guarantee* the LSS to be idle when the solar panel is deployed. It can be the case, but this is controlled by the OBC.

Mission Requirements

[Miss-1] Fulfilled. The position of the legs is initialised with zero and becomes known after communication with the leg modules.

[Miss-2] Partially fulfilled. The storage position of the legs is the same as the position of the legs when Lunar Zebro lays down. The laying down functionality is partially fulfilled and thus this requirement is partially fulfilled as well.

[Miss-3] Fulfilled. The LSS software can rotate the legs to any position deemed favourable during deployment and thus protect the locomotion hardware. It is required that the LSS software may be active at the time of deployment.

Interface Requirements

[P-Int-1] Fulfilled. The leg module software is verified to perform an emergency stop before the leg module hardware performs a board reset.

[P-Int-2] Unknown. This requirement is not yet tested.

[S-Int-1] Fulfilled. Every leg module contains the same code and functions correctly.

[S-Int-2] Fulfilled. All leg modules can be manufactured identical from a structures point of view.

[O-Int-1] Fulfilled. The LC does not send packets to the OBC when not explicitly asked for.

[O-Int-2] Fulfilled. A heartbeat packet is verified to be sent when no other communication is sent.

[O-Int-3] Partially fulfilled. Currently, only a subset of all data is available. The framework can easily be extended to include all data.

[O-Int-4] Fulfilled. All data going over this interface is either in SI units or is meaningful.

Environmental Requirements

[Env-1] Fulfilled. Both the OBC interface and the motor driver interface have a CRC as part of their packet structure.

[Env-2] Partially fulfilled. Because of the use of Dezyne, state transitions that are illegal are automatically handled correctly by the Dezyne generated code. However, also because of the use of Dezyne, input variables are not checked within the Dezyne generated code as Dezyne does not understand the concept of data types. Sanitising input values must be done with handwritten code and at the moment it is not guaranteed that every function is sanitising its input.

[Env-3] Not fulfilled. The functionality is implemented, but unused at the moment.

Operational Requirements

[Oper-1] Fulfilled. The LSS is not able to *not* listen to the OBC when active.

[Oper-2] Partially fulfilled. The LSS starts in the idle state, which is a safe state. The reset functionality in the LSS is implemented, but not used. This requirement is therefore partially fulfilled.

[Oper-3] Partially fulfilled. Every part of the LSS can be reset externally at any point in time, but the LSS can not do this by itself.

[Oper-4] Partially fulfilled. A command to reset every leg module is available, but currently unused.

[Oper-5] Fulfilled. The software design is formally verified to be without deadlock and livelock. Additionally, the state diagram shows that from every state, a transition to the fatal error state is possible.

[Oper-6] Not fulfilled. Currently, the only manner to go from the fatal error state toward the idle stat is by resetting the LC.

[Oper-7] Fulfilled. Starting and stopping any operation is verified in tests on an office floor and does not result in unsafe conditions.

[Oper-8] Not fulfilled. The tests in a test-bed of sand are not conducted due to time constraints.

Logistics Requirements

[Log-1] Fulfilled. Documentation is provided as comments in code.

[Log-2] Currently, the Lunar Zebro team does not have access to the source code of this thesis. Hyperion Technologies B.V. shall be consulted to determine if the source code can be made available to the project team.

Product Assurance Requirements

[Prod-1] Unknown. The reliability of the software is not determined.

[Prod-2] Unknown. The availability of the software is not determined.

[Prod-3] Partially fulfilled. Dezyne is used to design the LSS system, but it is still unknown if this increases the maintainability of the code.

[Prod-4] Fulfilled. The software triggers an emergency stop when an over-current situation arises. This happens before the protections in hardware shut down the system.

[Prod-5] Fulfilled. The leg module monitors the temperature of the motor and can send this information to the LC.

[Prod-6] Not fulfilled. The LSS software does not fully satisfy all requirements of the highest priority.

Configuration Requirements

[Conf-1] Fulfilled. The LSS software has one interface, the OBC interface, through which the entire system can be commanded and data is returned to the OBC.

[Conf-2] Fulfilled. The interface between the LC and the locomotion algorithm is described and a handwritten component is implemented in the LSS software design. The locomotion algorithm is implemented in this framework.

[Conf-3] Not fulfilled. At the moment, no framework is implemented for solving errors. Within the time frame of this project it is proven infeasible to develop a suitable solution.

Design Requirements

[Des-1] Fulfilled. The LC component is fully developed with the help of Dezyne.

[Des-2] Fulfilled. The first part of the software, running on the OBC, is written in C++. The second part, running on the leg module, is written in C.

[Des-3] Not fulfilled. The code is documented using regular comments in code instead of the use of Doxygen. The added value of Doxygen is deemed too small at this moment to invest time in this style of documentation.

Verification Requirements

[Ver-1] Not fulfilled. Lunar Zebro is tested for its walking speed in an office environment.

[Ver-2] Partially fulfilled. Different versions of the LSS software are tested for their compliance with the functional requirements during development. These versions are compliant with the functional requirements. However, the latest version is not yet verified.

[Ver-3] Not fulfilled. The Lunar Zebro team is not able to schedule tests for sub-teams to test their hardware and software in a moon-like environment.

6.2. Reflection on Reliability

In an office environment different versions of Lunar Zebro's LSS software perform the functions from the functional requirements without signs indicating obvious flaws in the software. The latest software iteration is still under development.

Does Dezyne Increase Reliability? It is not possible to conclude whether or not Dezyne increases the reliability of software. It is possible to conclude that Dezyne can deliver software that seems reliable at first glance.

6.3. Advantages and Disadvantages of Dezyne

The third goal of this thesis is to track the advantages and disadvantages of the use of Dezyne during the development of the LC. The advantages and disadvantages are split into an essential and a non-essential category. Essential advantages and disadvantages heavily influence the quality of the product made with Dezyne. Non-essential advantages and disadvantages are characterised by making the life of the software engineer easier or harder, without influencing the final product.

6.3.1. Essential Advantages and Disadvantages

Advantages The largest advantage of Dezyne is its proprietary language. A great deal of thought has been put into this language and its use, which is noticeable. The user is forced into a straitjacket, which is often a disadvantage, but in this case it is definitely an advantage. The use of the `interface` and `component` keywords forces the user to abstract away from physical implementation and define system components on a functional level. Often it will become clear that a component with multiple functions can be split up into multiple components that perform only one core function. This makes for a very organised design that is easy to communicate to others.

A second advantage is the fact that Dezyne guarantees single-threaded and run-to-completion execution of the design. Conceptually, it makes reasoning about a system very intuitive and easy. All events are handled via a First-In-First-Out (FIFO) queue. However, a lot of systems are not single-threaded. To handle multi-threaded systems while maintaining the aforementioned advantage, Dezyne introduces the keyword `external`. This keyword tells the verifier to check extra state-transitions to spot erroneous behaviour when asynchronous events come in from another thread. The extra

state machining required to have safe behaviour for external interfaces can be aggregated in an armour component. If the user wishes, it can also be added to the already existing state machine in the component using the external interface.

The last advantage of Dezyne is the fact that the verifier is able to check for deadlock and livelock. This gives the designer much confidence that, at the very least, the system will be operational. Currently, the verifier can not check for user-specified requirements. Verum B.V. is actively working on implementing a framework to verify user-specified requirements.

Disadvantages A disadvantage of Dezyne is the unknown real-time capability of the runtime library, which contains the event handling procedures. Since Dezyne uses FIFO queues to schedule its events, it is essentially an operating system. However, since the scheduling is done in a FIFO manner, there is no guarantee for a certain real-time capability. Additionally, it is unclear how much computational overhead the Dezyne generated code introduces and how that affects its operational speed.

A second disadvantage is the fact that Dezyne is not for every device. Some micro controllers make use of peripheral interrupts in combination with an interrupt routing network to trigger functionality on other peripherals. This type system can start actions on a micro controller from an interrupt without the processor, and thus Dezyne, even knowing. State machining can then be done without Dezyne tracking this. For this type of device, Dezyne is definitely not the tool for the job. However, these devices are often found in Embedded Systems.

The last disadvantage of Dezyne is the fact that it is still under development. An example of this is the verifying capability. During this project the same model is verified using different version of the Dezyne verifier and each version of the verifier outputs a different answer. Verum B.V. is always working on expanding their collection of regression tests that is used to verify the verifier. This is a good start towards showing the correctness of the verifier. If the test set would be checked by a third party, it would benefit the trustworthiness of the software.

6.3.2. Non-essential Advantages and Disadvantages

Advantages The ability to visualise complex systems with system- and state diagrams, helps the user tremendously in reasoning about the system. For example, the system view of Dezyne, as shown for this project in fig. 4.3, is effective. It also allows for sharing system ideas in an effective way. People who have no prior knowledge of the system are still able to quickly reason about it and thus aid in the design.

Another visualisation tool, the simulator, is great to help spot design errors. As shown in fig. 4.6, the functionality of the system can be displayed in a clear and compact manner.

Disadvantages Visualisation of systems is far from perfect yet. When creating state diagrams, the first listed variable is used as the basis for all states, even if the variable has nothing to do with the states in the system. On top of that, the state diagram does not contain an option to visualise sub-states, which would be a valuable addition.

The ordering of components in the system view is dependent on the order of declaration. There is no way to permanently tell Dezyne to route the lines between interfaces in a certain manner. The same goes for the positioning of the components relative to each other.

Dezyne does not support exporting diagrams. This leaves the user taking screen-shots which is definitely not beneficial for the image quality. This feature is definitely required and needs to be added when writing reports for clients that demand these diagrams.

Lastly, the documentation for Dezyne is often slightly outdated and sometimes even non-existent due to the fast pace Verum B.V. is moving at. For the tool to become more useful, the documentation needs to improve. Main points for improvement are the completeness and searchability of the documentation.

7

Discussion

Before drawing the conclusions from this thesis, every element of this thesis shall be reflected upon to better judge the value of its results.

Outline This chapter reflects on all parts of this thesis chapter by chapter. It shows the lessons learned and translates these to recommendations for future work.

7.1. Reflection

According to [21], a design project should adhere to the following structure:

- Introduction
- Requirements
- Concepts Study
- Concept Selection
- System Design
- Testing
- Conclusions and Recommendations

However, this thesis does not just design a product, it also tests a new method for designing the product. On top of that, this thesis commissions its own design study as opposed to receiving a set of high level requirements from a problem owner. This leads to a mix-up in the standard structure that is not beneficial for the end result. This is noticeable throughout the report as is detailed per chapter.

Background To give Lunar Zebro a proper chance of walking on the moon, the question how to best design the LSS for Lunar Zebro shall be answered. To create the most reliable software in the shortest amount of time, chapter 2 suggests a model-driven design approach. On top of this, it suggests the use of model checking to verify the functionality of the software. These suggestions are not based on previous scientific work proving these type of development techniques improve the end result.

Dezyne is a sensible choice for a tool that uses the two proposed design methods. However, it is unknown whether Dezyne is the *most* sensible tool to develop this type of software with. This thesis does not list other comparable tools and neglects to establish a framework to rank these different tools according to their performance.

Requirements The requirements are categorised using the standard ESA ECSS system for listing requirements. This causes difficulties when categorising requirements for software design, but on the other hand, it provides a framework where the designer is forced to think about different aspects or categories of the product. This reduces the risk of forgetting certain requirements and functionality.

System Design One of the reasons for using a model-driven design approach, is to increase the availability of Lunar Zebro. Currently, it is undecided what a six legged robot should do when one or more legs stop functioning. This type of faults, that arise during operation of the device, threatens the availability of the system. A model-driven software design approach allows for the construction of a framework for extensive state machines, that can help mitigate the effects of the aforementioned type of faults.

The design implements a state in the system in which errors can be solved. This state can only be reached from the emergency stop state and can only be left to that same state, or the idle sate. No

handwritten component is implemented in which extensive error solving can take place. This is due to the fact that a lot of external data is required to make decisions for solving errors and Dezyne cannot interpret this data. In order to be useful in a Dezyne component, events must be generated from interpreted data. A clear separation between the interpretation of data and the error solving state machine is not yet conceptualised.

A second element that is missing in this design, is the start and shutdown procedures of the system. Dezyne provides no conceptual framework in which start and shutdown behaviour of a system is captured. Upon mentioning this, Verum B.V. acknowledged that this might be an interesting direction for further research. In the LC, the start procedure that is implemented is the starting of different timers. These need to be running when the program starts, but this is done outside of Dezyne. This is a very dirty work-around, considering the Dezyne philosophy where all events are checked for the manner in which they are used. Starting the timers in this way is not verified by Dezyne and could potentially break the system.

Lastly, the LC component in the design is a spider in the middle of a web. A lot of functionality inside of this component is there to protect the system from unexpected events coming in over the interfaces. A better way of solving this, is to use armour components. Armour components are a purely conceptual type of component. These components are used to sanitise events over interfaces. This allows the LC to contain a behavioural state machine. All non-behavioural state-machining, like handling unexpected events, can then be done in the armour components.

Code Generation and Integration During the project, Verum B.V. changed the method of generating skeleton files. The method described in this report is the most recent.

Unfortunately, the code integration is not performed on the OBC of Hyperion Technologies B.V. Instead a Raspberry Pi (RPi) is used. To use the OBC of Hyperion Technologies B.V., a carrier board that breaks out all the connectivity is needed. This carrier board is developed by other team members of the Lunar Zebro project. The board is still under development, requiring the use of an alternative integration platform. An RPi is a good fit for temporarily replacing the OBC, as it also uses a Linux distribution and has the necessary hardware on board.

During the conceptualisation of the subject for this thesis, it was thought that Dezyne would also be used to aid in the design of the software for the leg module. During the project it has become clear that this is not possible. The leg module does not contain enough relevant state-machining actions to justify the use of Dezyne. Additionally, as discussed earlier, some micro controllers can bypass the processor and thus completely bypass Dezyne's state machine. Therefore, Dezyne is decided to be unsuitable for the development of the leg module. One can even question the use of Dezyne for development on micro controllers in general.

Results Some requirements are not fulfilled or only partly fulfilled. Regarding the functional requirements for example, to stop walking, a walking command is sent to the LSS with speed defined as zero. This is done instead of combining this functionality with the stop command. This is deemed the responsibility of the OBC. The reasoning behind the functional separation, is the request from the Lunar Zebro team to have the OBC have complete control over the LSS. The same reasoning is applied to deploying the solar panel, which is completely in the control of the OBC. This way, the OBC receives more responsibility, but also greater freedom.

Other important requirements that could not be fulfilled, are related to the reliability of the designed solution. This thesis shows that designing a system with Dezyne is possible, but no results are generated concerning the possible improvement of the reliability of the system. This is unfortunate, since the reliability problem is a valuable part of this thesis. There are two main reasons for the inconclusiveness of this report on this part.

First No framework to quantify the reliability of the LSS software is created in this thesis. The more traditional design approach mentioned at the beginning of this chapter would have included this. Unfortunately, this thesis did not follow this approach.

Second Measuring of reliability in embedded systems is not a trivial task as is described by [44]. The hardware, especially in embedded systems, is a major contributing factor to the difficulty in establishing the reliability of the software in a system.

Dezyne There are disadvantages concerning the use of Dezyne. Especially the fact that it requires a learning curve to understand not only how Dezyne works, but also the concepts behind the software.

There is also the question of vendor lock-in. If Dezyne is adopted as the primary development tool, but needs to be replaced a few years later, how will the projects be maintained afterwards? The generated code can still be used, but even a small change in the state machine requires extensive alteration of the previously generated code. Since this generated code is not properly understood by the user, and nor

should it, it is very likely the entire project needs to be redone, undercutting the whole advantage of using a code generating platform.

7.2. Recommendations for Future Work

This section is split into two categories. The first category discusses future work directly related to the current LSS software design. These recommendations can immediately be picked up when new team members of the Lunar Zebro project continue the development. The second category is related to the model-driven development method and the usage of Dezyne.

7.2.1. LSS Software

The recommendations for future work on the LSS of Lunar Zebro are, highest priority first:

Logging The data manager needs the addition of logging functionality. Logging must be a function of frequency, destination (file or I/O) and presets for what to log.

Error Solving A proper framework for solving errors must be established. The main challenge is to draw a line between interpreting data gathered in the system and generating events as a result of this data. This requires an intricate interplay between handwritten components and Dezyne components. At the same time, uncontrolled growth of this framework must be prevented. Adding events one at a time for every error scenario is considered a wrong approach. It will leave too much opportunity to forget an error scenario.

Armour The concept of armouring must be applied to the current design. Every interface connected to a handwritten component or an external interface must be fitted with an armour component. This component turns a robust interface into a strict interface as described in [11].

Concerns The separation of concerns in the Zebro Project ensures the leg module can protect itself and offload the other control layers. Lunar Zebro is partially undoing this strategy by demanding that the OBC must be in charge at all times. This is wrong. The leg module and the LC must be able to protect themselves without the OBC.

Therefore it is suggested to develop a second version of the LSS software that can be started by the OBC if it wants unlimited power over the LSS. This alternate version of the LSS functions as a bridge between the OBC and the commands that can be sent to the leg module. This way, the OBC receives direct control over each leg module.

Locomotion Algorithm The locomotion algorithm is contained in a handwritten component. It is suggested to look into moving this functionality onto a different thread. This would provide an even more modular software design for people to work on.

7.2.2. Development Method

Regarding the development methods used in this thesis, these are the recommendations:

Dezyne It is highly advised to continue the use of Dezyne. Despite the difficulties of its use, there are also possible benefits. The benefit of increased maintainability during development is very useful for a student project in which people often come and go. The visualisation tools of Dezyne can facilitate a smoother transfer of knowledge.

If Lunar Zebro reaches the moon, the LSS software is finally thoroughly tested. If this test is a success, software development as an industry will be one step closer to innovating itself and Lunar Zebro will be leading the way.

ECSS The official systems engineering method of writing requirements for systems is very helpful. Even though systems engineering is sometimes frowned upon, no system is unique and they can all be described using these requirement categories. It is recommended to continue the use of this standard.

Tools Before diving deeper into the assessment of the usage of Dezyne, a study of the state-of-the-art concerning model-driven software development tools shall be conducted. It is recommended to look into tools that, like Dezyne, combine a *holy trinity* of software design methods: i. Model-driven Design Approach ii. Model Checking iii. Code Generation. Having separate tools for these functions will avoid vendor lock-in, but requires agreement on uniform standards between software developers.

Reliability Determining the reliability of software and thus concluding the reliability of the development method is difficult. It is worthwhile to study how the reliability of embedded systems can be quantified. It is recommended to look into separating the software and hardware for this study. Embedded software should be platform agnostic, even though it is embedded software. Regardless of which micro processor or sensor is used, the functionality stays the same. Determining the reliability of software, without hardware, could be a step in the right direction.

8

Conclusion

The TU Delft is fortunate to have the opportunity to bring the Zebro concept to our moon. This thesis describes the design of the Locomotion Sub-System (LSS) software for Lunar Zebro. To maximise the chances of success, the software is developed using a novel design tool called Dezyne. The goals for this thesis are:

i. Deliver a reliable software design for the Locomotion Sub-System (LSS) of Lunar Zebro.
ii. Deliver a working implementation of the Locomotion Sub-System (LSS) software design on Lunar Zebro's hardware.
iii. Investigate the advantages and disadvantages of the use of the model-driven software design tool *Dezyne*.

These goals are only *partly* reached. The goal to deliver a reliable LSS software design is partly reached. The design is lacking a framework in which errors that occur during operation can be resolved by means of extensive state machining. This framework is key in increasing the availability and reliability of the LSS. Dezyne is a tool that can aid in the design of such state machines, but this is not explored in sufficient detail in this report due to time constraints.

It can be concluded that a reliable system can be created with Dezyne, but it is unknown if this system is more reliable than a classical implementation of the system. This is in part due to the fact that this thesis starts with the premise that Dezyne can increase reliability of systems, without researching how reliability in embedded systems can actually be determined.

The goal of implementing the design on the hardware of Lunar Zebro is only partly reached, since the required hardware is not available. An implementation of the software is delivered on partially different hardware. It is made sure that this hardware is comparable to the hardware of Lunar Zebro.

The goal of listing the advantages and disadvantages of the use of Dezyne is achieved. The largest advantage of the use of Dezyne is the structure that is used in the proprietary language. It forces the developer to split up the system into small blocks of functionality and define how these blocks communicate. This helps the designer tremendously in creating a well-designed system for the required functionality.

The largest disadvantage of the use of Dezyne is the extra effort required to use the tool in combination with the fact that the benefits are very difficult to quantify. This thesis tries to make a start with the quantification of the benefits, but future work is definitely required.

There is much work to be done for both Lunar Zebro as well as Dezyne. Lunar Zebro's LSS is missing components that still need to be added before it is ready to leave for the moon. Dezyne should increase the stability of the verifying capability and improve the user-friendliness of the tool. Although both are not perfect, it does not mean that there is no bright future ahead. If the right people come together and put in the effort, it might be …

> "...one small step for a Zebro, one giant leap for The Netherlands."

A

Motor Selection and Tests

There are many requirements that the motor must meet, but most of them are in fact wishes. Only two requirements must be satisfied no matter what. First, the motor shall deliver the torque necessary for Lunar Zebro to walk and secondly, the motor shall survive the lunar environment for the entirety of its mission duration. These two requirements are most important when selecting the motor.

After selection, small tests are done on the motor using a motor driver from the same manufacturer to verify the motor meets the above stated requirements.

A.1. Motor Forces Calculation

An estimate of the torque the motor shall deliver to make Zebro walk properly is made using static analysis. In [30] a rough estimate is made of the static forces acting on one leg. This calculation was simplified and slightly corrected in [2]. The static forces model is shown in fig. A.1.

It is decided each leg shall be able to support the entire mass of Lunar Zebro, 1.5 kg, on its own. In normal operation a leg carries only half the mass of Lunar Zebro at a maximum. This is the middle leg on either side of the rover. Calculating the required torque with the full mass of Lunar Zebro is thus a safety factor of 2. Using the full mass and the model for static forces displayed in fig. A.1, it is possible to calculate the necessary driving torque on the shaft for different leg diameters and inclination angles. The two equations to do so are shown in eqs. (A.1) and (A.2). Equation (A.1) is used for β lower than $\pi/2$ and higher than $3\pi/2$ and eq. (A.2) for β higher than $\pi/2$ and lower than $3\pi/2$.

The results are shown in fig. A.2. From this diagram it is concluded that the maximum torque necessary to rotate a leg with a radius of 6 cm is 150 mN m. A scenario when this would be necessary is when Lunar Zebro lifts itself by spinning its legs backwards.

$$\text{Motor Torque [mNm]} = \mathrm{F}_{\mathrm{g_{moon}}} \cdot ((R - d) \cdot \cos(\alpha - \beta) + R \cdot \sin(\alpha)) \tag{A.1}$$

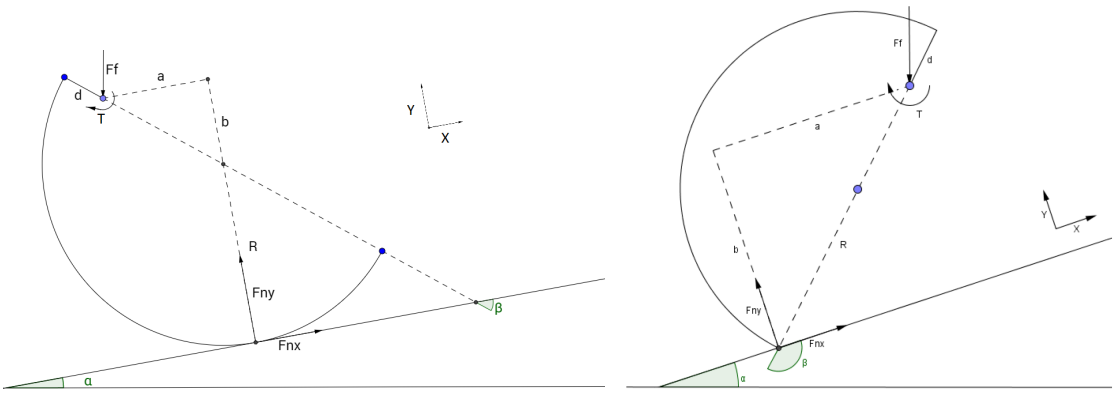
$$\text{Motor Torque [mNm]} = \mathrm{F}_{\mathrm{g_{moon}}} \cdot -(2R - d) \cdot \cos(\alpha - \beta) \tag{A.2}$$

Additionally, there are other forces acting on the leg and the axle of the gearbox or the motor. The nominal radial force acting on the axle is defined as the mass of Lunar Zebro times the gravitational pull of the moon. This is shown in eq. (A.3). It is impossible that the entire mass of Lunar Zebro will be loaded radially on the shaft of the motor or gearbox. Using the entire mass of Lunar Zebro again serves as a safety factor of 2.

The radial shock load is defined as the load applied when Lunar Zebro falls from a height of 2 m. The radial shock load is shown in eq. (A.4) and is 487.5 N. It is assumed that when Lunar Zebro hits the ground, there is 1 cm of shock absorption. The validity of this number is quite questionable since there are a lot of unknowns when falling onto the Moon's surface. The only definitive way of checking whether or not Lunar Zebro can handle the radial shock loads, is by dropping it on earth from a corrected height on different surfaces.

$$\mathrm{F}_{\mathrm{Radial\_Nom}} = \mathrm{M}_{\mathrm{Lunar\_Zebro}} \cdot \mathrm{g}_{\mathrm{Moon}} \mapsto 1.5 \cdot 1.625 = 2.44\,\mathrm{N} \tag{A.3}$$

$$\mathrm{F}_{\mathrm{Radial\_Shock}} = \frac{\mathrm{M}_{\mathrm{Lunar\_Zebro}} \cdot \mathrm{g}_{\mathrm{Moon}} \cdot \mathrm{h}_{\mathrm{drop}}}{\text{deceleration distance}} \mapsto \frac{1.5 \cdot 1.625 \cdot 2}{0.01} = 487.5\,\mathrm{N} \tag{A.4}$$



(a) Inclined position for β lower than $\pi/2$ and higher than $3\pi/2$ [2].

(b) Static forces and dimensions of the leg in inclined position for β higher than $\pi/2$ and lower than $3\pi/2$ [2].

Figure A.1: Static forces and dimensions of the leg.

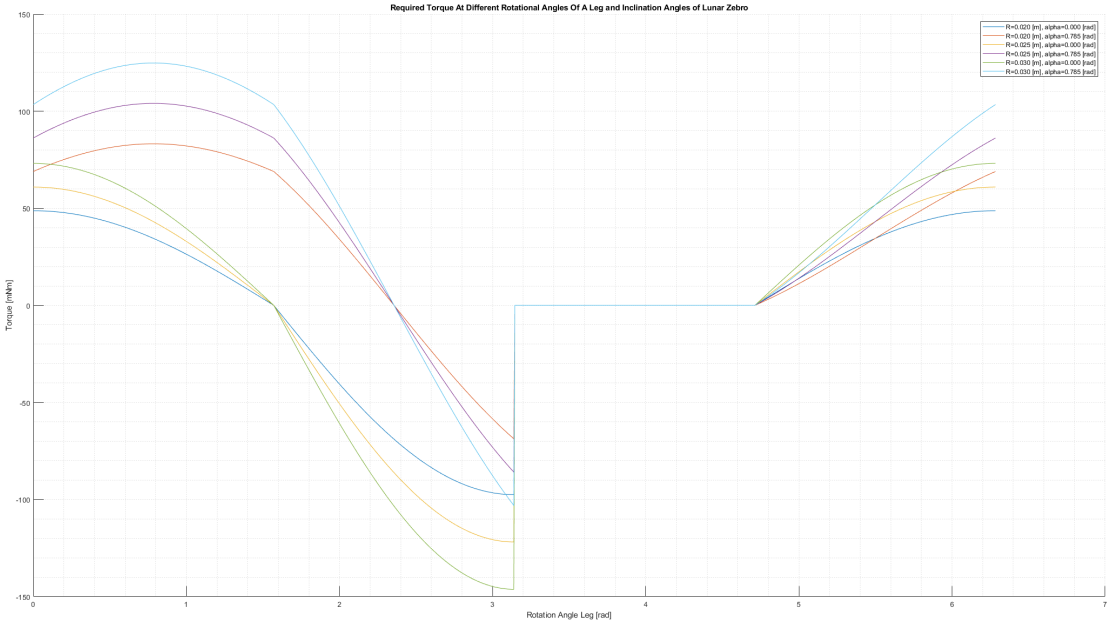


Figure A.2: Required torque of a c-shaped leg of different lengths with Lunar Zebro in different inclinations. Between π and $3\pi/2$, the leg is not touching the ground.

The axial loads are in theory non-existent if Lunar Zebro walks in stable and straight manner. However, when walking on slopes or receiving sideways disturbances, there can be axial loads. It is assumed that there are no axial shock loads, but only nominal loads in certain situations. It is expected that the highest nominal axial loads are experienced when Lunar Zebro is laying on it's side and it's entire mass is supported by 1 leg. This force is the same as the nominal radial force: 2.44 N.

A.1.1. Design Space Exploration for Motors

The motor, gearbox and motor driver selection is heavily intertwined for obvious reasons. The main selection criteria are torque, mass and volume.

There are 4 electric motor types that can be considered since Lunar Zebro is battery powered. These types are: brushed DC motor, brush-less DC motor, stepper motor or piezoelectric motor. All these motors can be delivered by either Maxon Motors, Faulhaber or both.

Brushed vs. Brush-less DC Motor Brushed DC motors have one great advantage over brush-less DC motors: simple control. There is no need for sensors to measure the rotor position and no computing power required to commutate the motor. Beside this advantage, a brush-less DC motor out-performs its brushed competitors on every criteria what the mission is concerned.

One of the problems with brushed DC-motors is overheating. With this type of motor, heat is generated on the shaft of the motor since the rotor consists of coils. The rotor only has thermal conductivity with the leg. Other than that, heat must be radiated towards the stator. On top of that the leg can also conduct heat from the Lunar Surface into the motor. Brush-less DC motors have the coils mounted on the stator of the motor, which completely bypasses this problem.

Besides the thermal overheating, the brushes of a DC motor cause a lot more problems. Subject to the lunar environment, they will wear extensively over short period of time. This will cause particles of brush floating around in the motor or worse, the brushes reaching their end-of-life before the mission is over. Another risk is cold welding of the brushes and the commutator. This could happen during transport and cause the motors to not be able to start rotating.

Stepper Motor Stepper motors can be regarded as a class of BLDC motors that have been adapted to turn small increments instead of continuously. According to [51] there are roughly 4 types of stepper motors: reluctance, permanent magnet (PM), hybrid and piezoelectric stepper motors. All types use a combination of coils and PMs or just coils. The coils on the stator can have different windings. Uni-polar stepper motors have windings where the current will always travel in the same direction. Bi-polar stepper motors have windings where the current needs to be reversed to spin the motor in the other direction. This is why uni-polar motors can be driven by simple transistors while bi-polar motors require something that resembles an H-bridge to reverse the current.

The simplest type of stepper motor is a reluctance stepper motor. It has N pole pairs of coils on the stator and a steel rotor. The pole pairs generate a magnetic flux that is followed by the steel rotor. This type of stepper motor generally takes large steps and delivers larger torque levels.

The second and third type of stepper motor is a hybrid stepper motor. They use a magnetised rotor and coils as a stator. By magnetising different sequences of pole pairs, the magnet can be moved by certain increments. Because hybrid stepper motors always contain a permanent magnet (PM), they are also called PM stepper motors. Hybrid is the more correct term when torque is not only delivered due to the magnet reacting to the magnetic field of the pole pairs, but when reluctance torque is delivered as well due to a PM-iron combination. Hybrid stepper motors generally develop the highest torque to current ratio of all types of stepper motors. PMs are unfortunately influenced a lot by temperature and vibrations.

The last type of stepper motor is a piezoelectric stepper motor. This type of motor does not contain any magnetic materials. They are mostly used in areas where very high accuracy is required. This type of stepper motor is also called a piezoelectric motor and is discussed later.

The required nominal torque of 0.3 N m does not allow for the use of a stepper motor without using an additional gearbox. Because of this the angular position of the motor is not one-to-one related to the angular position of the leg. There is a relation between the rotation of the motor and that of the axle of the gearbox. However, a stepper motor is susceptible to slip. In contrast to a regular BLDC motor, the movement of a stepper motor can not be measured with internal sensors. A stepper motor can be told to make a step, but not make it and there is no way of knowing this without extra sensors. The slip of a stepper motor makes it unsuitable for use in Lunar Zebro.

Piezoelectric Motor Piezo-electric motors use piezo-materials to have the stator change shape slightly causing the rotor to move. This shape-changing is done by calculating the mechanical harmonics of the

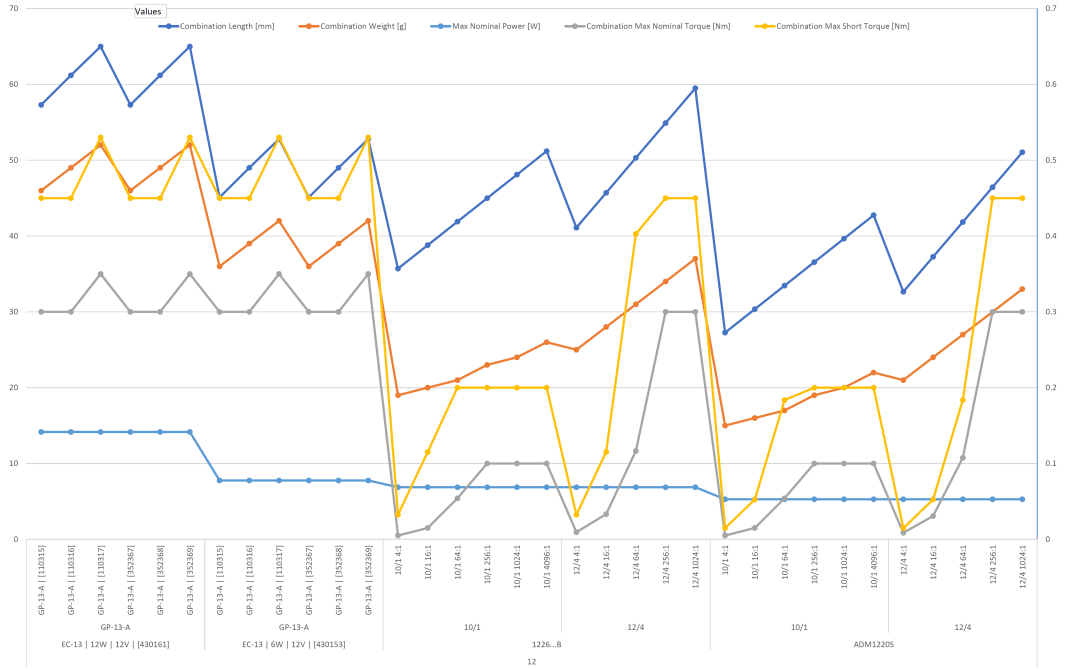


Figure A.3: Specifications of motor-gearbox combinations that are most relevant considering the mission requirements.

rotor and applying voltages with those harmonic frequencies. Without power a piezoelectric motor is locked, which is a very large advantage for Lunar Zebro because it can stay upright without power consumption.

On top of that, this type of motor has a very precise movement. Unfortunately, the control of this type of motor to gain the required accuracy is complicated. Additionally, the torque it can deliver is small. Because of the small torque and the fact that this type of motor is very uncommon in space missions, this motor type is not considered for Lunar Zebro.

Result Brush-less motors are chosen to drive Lunar Zebro because of their advantages over the other types of motors. The disadvantages of brush-less motors are deemed surmountable in favour of the advantages. This choice is supported by papers such as [32], where first stepper motors were preferred, but proven insufficient. Their suggestion to improve performance is the use of BLDC motors.

A.1.2. Motor Choice

To select the best motor for Lunar Zebro, a brute-force approach is taken. All brush-less motors developed by either Maxon Motors and Faulhaber are considered. They are matched with all possible gearboxes from the same companies. The results are plotted so a choice can be made according to the combined specs of all combination. Out of 289 possible combinations the right one is picked.

The most relevant combinations and their specifications are plotted in fig. A.3. From this graph it becomes clear both Maxon Motors (EC-13) and Faulhaber (1226...B) can deliver a suitable motor-gearbox combination. Eventually, the 1226-012-B motor in combination with a 256:1 12/4 gearbox from Faulhaber is chosen because of its lower mass and shorter lead time.

A.2. Motor Tests

Before the motors are used in Lunar Zebro, their compliance with all requirements shall be verified. The following test plan describes the procedures used to verify the requirements set for the motor and gearbox combination.

The motors that are going to be used for Lunar Zebro are selected using the properties listed in the datasheet. Verifying the compliance of a motor regarding the requirements set for Lunar Zebro comes down to verifying the values that are in the datasheet. The specific values that are of interest and that can be checked in the current time-frame are:

- No Load Speed

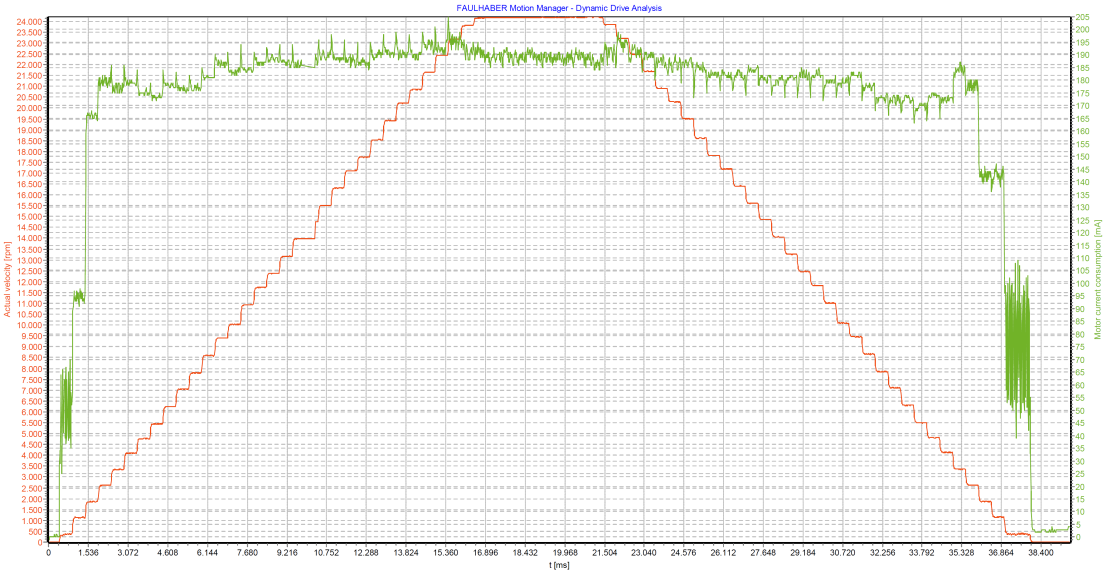


Figure A.4: Measured speed and current consumption during no-load test.

- No load Current
- Stall Torque
- Stall Current
- Temperature during different modes of operation
- Mass
- Dimensions
- Terminal Resistance Motor (phase-phase)

A.2.1. No-load Characteristics

The motors of Lunar Zebro receive an intermittent load torque. When the leg is touching the ground, the load is large. When the leg is not touching the ground, it can rotate freely. Because of this, it is assumed that in steady state, when the leg is spinning, the motor can be modelled as receiving no load.

The important properties that shall be verified are the *No-Load Speed* of the combination and the *No load Current* of the motor. The no-load speed of the motor listed in the datasheet is suspected to be too high as a gearbox with an unknown friction torque is directly attached to it. This friction torque can be estimated by measuring the no-load current and calculating backwards from there using the motor torque constant k_M listed in the datasheet. The motor torque constant will later on be verified with the stall torque test.

The no-load speed test is executed with the help of the Motion Manager software from Faulhaber and the MCBL3002 motor driver. The motor driver is put in a mode where the voltage of the PWM signal going into the motor can be set. This voltage is increased from zero by steps of around 0.37 V every 500 ms until the maximum voltage of 12V is reached. Inside the Motion Manager software, a plot is made of the speed of the motor which is measured by the internal hall effect sensors of the motor. Also the current going through the motor is measured. The result is shown in figs. A.4 and A.5. Exporting the data of the plots enables calculating the average no-load speed and current.

The maximum speed measured is 24203 RPM and the average speed at maximum voltage is 24172 RPM. The average current during no-load speed at maximum voltage is 189 mA. From this the combination of static and dynamic friction torque of the gearbox can be calculated as shown in eq. (A.5). In this equation C_0 represents the static friction torque of the motor and C_V the dynamic friction torque of the motor. At 24172 RPM the friction torque of the motor alone is 0.201 mN m. Calculating the friction torque of the gearbox at maximum voltage and no-load gives 0.578 mN m. This is almost three times as large, which makes sense since the gearbox has 4 planetary stages.

$$\begin{aligned}\text{Gearbox Friction Torque} = (I_{no-load} \cdot k_M) - C_0 - (C_V \cdot n_0) \mapsto \\ (0.189 \cdot 4.12) - 0.073 - (5.3 \cdot 10^{-6} \cdot 24172) = 0.578\,\text{mN m}\end{aligned} \tag{A.5}$$

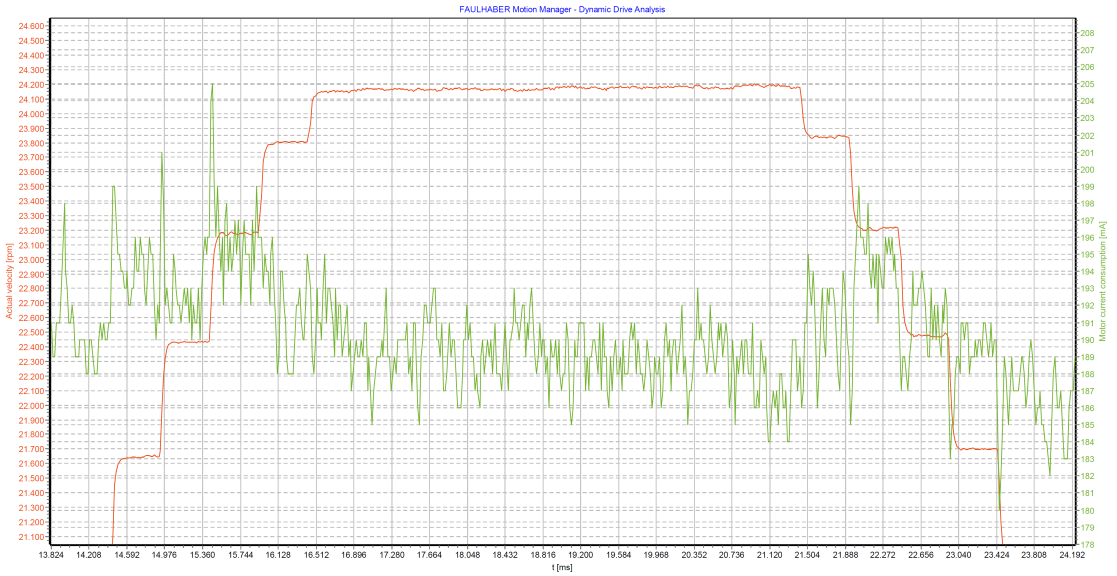


Figure A.5: close-up of the no-load speed and current consumption at maximum supply voltage.

From the data displayed in fig. A.4 the friction torque of the gearbox can be calculated for different speeds in the same way as before. The gearbox is specified by the supplier to only be used at a maximum of 5000 RPM on the in-going shaft. Therefore, the friction torque calculation is done for speeds up to 7000 RPM. A plot of this is shown in fig. A.6. It is clear that a negative friction torque for the gearbox at 0.18 V is not possible, however, it is very small and may be neglected. The calculations stay relevant.

A.2.2. Stall Characteristics

To test the stall characteristics of the motor, a static measurement method is developed. The motor is fitted with a 30 cm long beam. Weights can be attached to the beam at marked intervals. A weight is attached and the motor is instructed to rotate the beam to a horizontal position and hold that position. If the combination is able to lift and hold the beam, the weight is attached further away from the shaft. This is done until the combination is no longer able to lift the weight because the maximum current through the windings of the motor is reached. A photo of the test setup is shown in fig. A.7.

The results are shown in fig. A.8. A few observations are made. First, the motor stalls at 860 mN m. The motor is not able to hold the position of the beam at the 4 data points at a higher torque. Luckily, this stall torque is well beyond sufficient for Lunar Zebro.

Secondly, the maximum currents drawn show a linear dependency regarding the required torque. This is to be expected from an electric motor. The averaged current consumption does not have this linear dependency. It shows somewhat of a straight line indicating either the current measurement is influenced by the motor driver, or the motor driver performs poorly in the lower torque regions. Since the internal workings of the Faulhaber motor driver are not known, no conclusion can be drawn on this.

Third, from this data, the motor torque constant k_m can be calculated for the average power consumption and the maximum current consumption. For the average current $k_m = 2.80\,\mathrm{mN\,m\,A^{-1}}$ and for the maximum current $k_m = 1.48\,\mathrm{mN\,m\,A^{-1}}$. This is both significantly under the number specified in the datasheet: $k_m = 4.12\,\mathrm{mN\,m\,A^{-1}}$.

A.2.3. Temperature Measurements

When the motor does not receive a load, the leg spins freely. To have an indication about the temperature of the motor at no-load, the motor is continuously run at two different speeds for half an hour. The first speed is 5000 RPM. This speeds is estimated to be the average speed the motors will turn on the moon. With this speed it takes a leg around 3 s to complete one revolution. This is slow compared to Zebros walking on Earth, but seems a safe number for walking on the moon. The second speed is double the first speed, 10000 RPM. This test is run to put the motor under more stress and ensure safe operation. This is especially interesting considering the fact that the input speed of the gearbox is rated at a maximum of 5000 RPM.

The temperature is measured in two ways; the first being a thermal imaging camera and the second

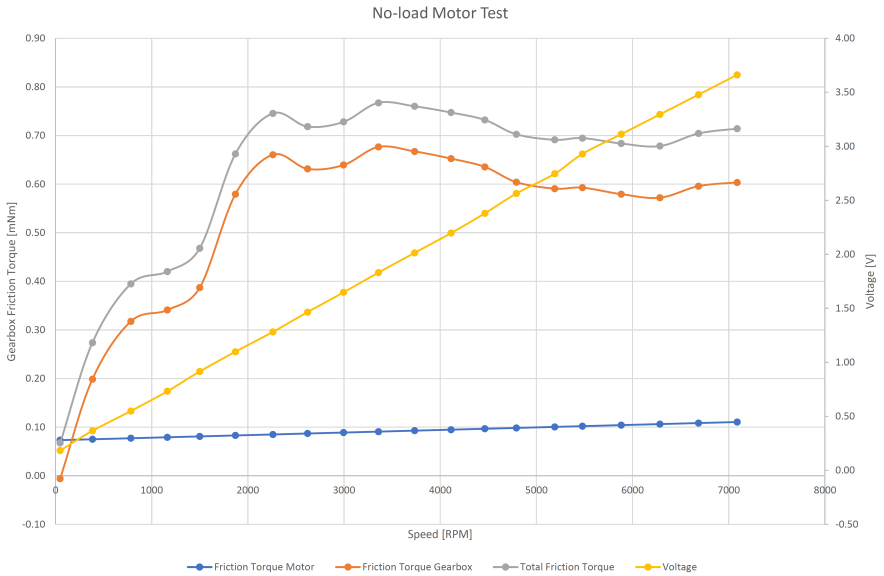


Figure A.6: Calculated friction torque from a no-load test.

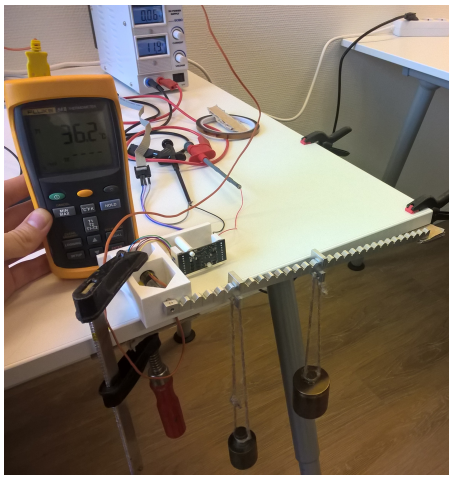


Figure A.7: Stall Torque Test Setup

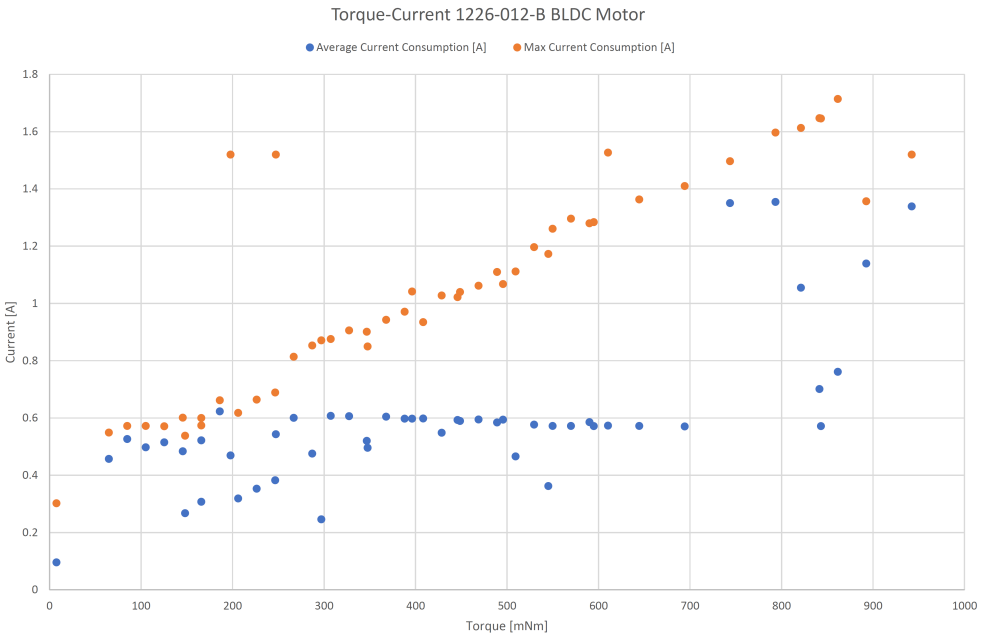


Figure A.8: Result of the stall torque test with averaged current and maximum drawn current.

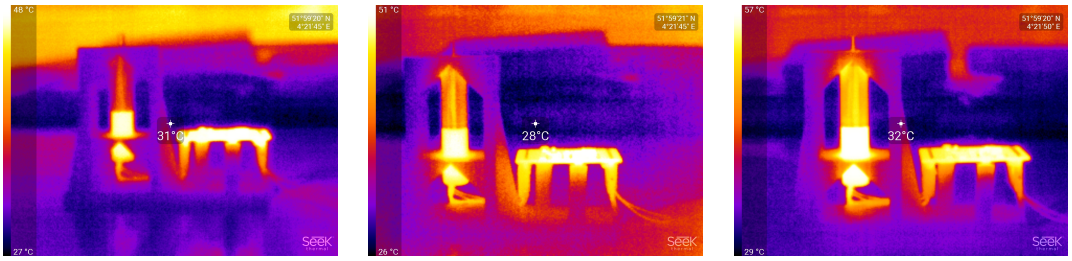


Figure A.9: Thermal images of the motor at 5000 or 10000 RPM.

a Fluke thermocouple. The first method is to find out where the motor or gearbox accumulates the most heat. The second is to obtain an accurate measurement of the temperature. The images of the thermal camera in fig. A.9 show how the motor heats up and the gearbox temperature rises when spinning at a higher speed. The actual temperatures are listed in table A.1.

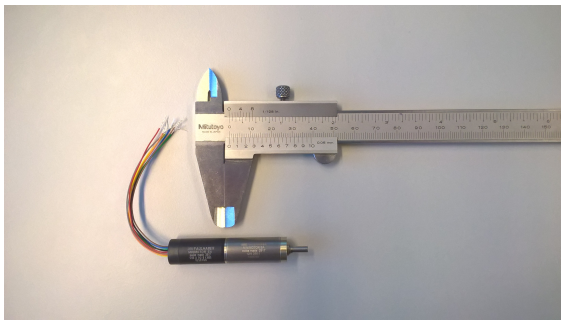
During the 10000RPM temperature measurement, an oscillation in the current consumption is observed with a frequency almost equal to the outgoing gearbox axis RPM. This indicates that one of the stages of the gearbox towards the outgoing shaft has created friction. Another possibility is a bearing at the outgoing shaft that is not running smoothly. This gives reason to believe Faulhaber has specified correctly that the gearbox can handle a maximum of 5000 RPM continuously or the outgoing axis has somehow sustained damage.

During the stall torque measurements, the temperature of the combination is closely monitored and never exceeds 50 °C. Because the load on the motor is very intermittent during this measurement, this temperature measurement can not be regarded as a definitive indication of what temperature the motor will reach during heavy operation. It is merely an indication.

Speed [RPM]	Final Temp. Motor [°C]	Final Temp. Gearbox [°C]
0	22.1	22.1
5000	33.5	31.5
10000	39.2	32.4

Table A.1: Temperature measurement

(a) Mass measurement



(b) Dimension measurement

Figure A.10: Motor measurements

Dimension	Measured Value [mm]	Within Tolerances [Yes/No]
Combination Mass	33.7	Yes
Combination Length	50.05	Yes
Diameter Motor	12.00	Yes
Diameter Gearbox	11.95	Yes
Diameter Gearbox Axle	3	Yes
Diameter D-shape of Gearbox Axle	2.75	Yes
Diameter Bushing Gearbox	6	Yes
Length D-shape on Gearbox Axle	6	Yes
Length Gearbox Axle	7.2	Yes
Length Axle to Bushing	8.9	Yes
Length Axle to Face Gearbox	9.85	Yes

Table A.2: Overview of the motor dimension measurements.

A.2.4. Measuring Mass, Dimensions and Terminal Resistance

The motors are supplied with CAD files and a datasheet containing technical drawings. To verify these two documents, the motors are measured using a scale and caliper. See fig. A.10. The scale has an accuracy of 0.1 g and the caliper can measure with an accuracy of 0.05 mm.

The mass of the motor-gearbox combination is measured to be 33.7 g. This is well within the 34 g listed in the datasheet.

Not all dimensions given for motor and gearbox are equally important or can be accurately measured using a caliper. For example, because it is a motor and gearbox combination, the separate lengths of the motor and gearbox can not accurately be measured, but only the combination length. Luckily, this is also the dimension that is most important to the project. The most important dimensions that are measured are listed in table A.2. The dimensions given by the manufacturer are shown in figs. A.11 and A.12.

The terminal resistance is measured with a HP 34401A multimeter. Two measurements are done for redundancy purposes, one with two wires and one with four wires. The probes are attached to the connectors that have been crimped to the wires. This way the resistance that is measured includes the resistance created by the connector to measure the total resistance and not only that of the windings. The datasheet specifies a phase to phase resistance of 5.45 Ω. The result of the measurements is show in table A.3.

A.2.5. Results

The no-load speed of the motor is roughly 3000 RPM lower than specified in the datasheet. This difference is explained by the gearbox that is connected to the motor. Regardless of this fact, the motor is plenty fast for Lunar Zebro.

The no-load current is measured to be 189 mA. This is more than three times as high as in the datasheet. It is not sure of all of this increase should be attributed to the friction torque added by the gearbox. Unfortunately, from these tests there is no way of knowing.

The stall torque of the motor is much higher than needed for Lunar Zebro. It allows for a safety factor

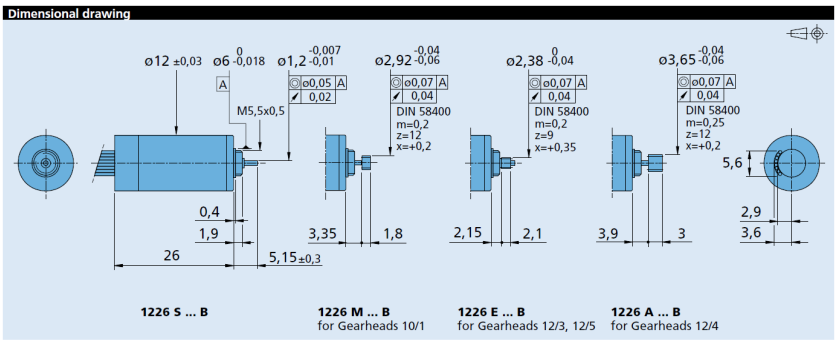


Figure A.11: Dimensional drawing of the motor taken from the Faulhaber datasheet.

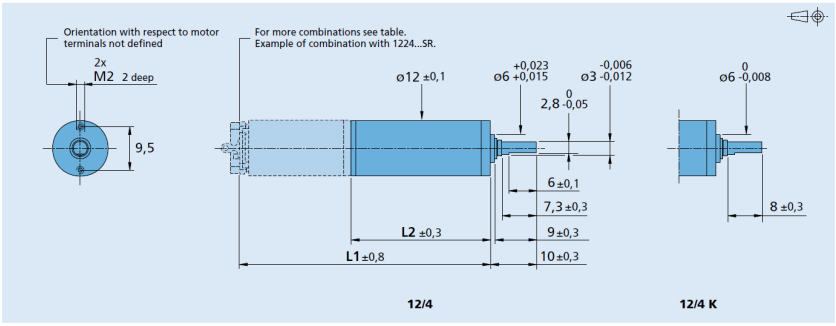


Figure A.12: Dimensional drawing of the 12/4 gearbox taken from the Faulhaber datasheet.

Measured Phases	Measure method [2w/4w]	Value [Ω]
phase a to b	4w	5.360
phase a to b	2w	5.465
phase a to c	4w	5.262
phase a to c	2w	5.401
phase b to c	4w	5.396
phase b to c	2w	5.544

Table A.3: Phase Resistance Measurements

of almost three. The stall torque is limited by the maximum current through the coils of the motor. The stall current is thus equal to the maximum allowed current which is specified to be 1.7 A by Faulhaber's Motion Manager Studio.

From the start it is clear the motor becomes very hot. However, it never passes 50 °C in all tests that are performed. The maximum temperature of the motor is listed to be 100 °C, which gives a safety factor of two.

The mass of the motor and the dimensions are complying with the wishes of them being 'as small as possible'.

The terminal resistance is well within the specification on the datasheet. The 2w measurements are very close to the listed value, where as the 4w measurements are well below. The 4w measurements are more accurate and these numbers lead to the conclusion that the datasheet is correct.

B

Leg Design for Lunar Zebro

Zebro is well known for it's characteristic leg shape. Designing a new leg shape is out of the scope of this project, but the leg of DeciZebro shall be adapted to fit the lunar environment. To successfully adapt DeciZebro's legs, it is necessary to become familiar with the properties of the c-shaped leg.

The c-shaped leg is advantages when climbing objects as described in [40], where c-shaped legs are researched for climbing stairs. The main reasons why this shape is convenient are the high climbing height and rolling contact point. Because of the rolling contact point, the hip (motor axis) of Zebro will move in a relatively straight line when climbing an object. This is shown in fig. B.1. Little movement in the hip is desirable for energy efficient walking. According to [40], Zebro can save 37% of power when climbing objects compared to previous non-round leg designs.

A disadvantage of the c-shaped leg is that the moving contact point results in uncontrolled accelerations of Zebro when it climbs objects of unknown shape. If the leg touches the object it wants to climb close to the hip, the rolling contact will start moving to the toe of the leg With a speed proportional to the rotational speed of the leg and the radius of the leg. If the rotational speed of the leg is not changed during a step, the hip will have different accelerations in different directions. When subject to the moon's weaker gravity field, this could result in an unwanted catapulting effect launching the robot away from the surface of the moon.

Apart from the c-shape, there are other important aspects of the leg that need to be designed. These are: (i) connection of the leg to the motor shaft, (ii) placement of a magnet on the leg, (iii) width of the leg, (iv) tread of the leg.

B.1. Motor Shaft Connection

The connection to the motor shaft shall be able to handle the required torque. Motors and gearboxes are delivered with a certain output shaft dimension and shape. The outgoing axle of the motors of Lunar Zebro have a D-shaped axle as shown in fig. A.12. The manufacturer is not able to change this axle for the amount of motors that are ordered for this project.

The connection from the leg to the shaft is made by making a d-shaped hole in the leg. However, there is still an axis of freedom along the shaft and the leg can come off. To fix the leg to the shaft, multiple

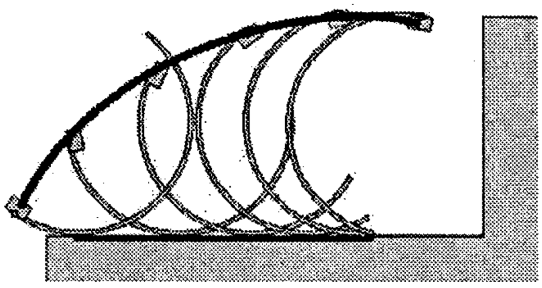


Figure B.1: A Half-Circle leg shown in stop motion. The black line is a trace of the hip motor shaft position. The straight line contains the leg contact points. [40]

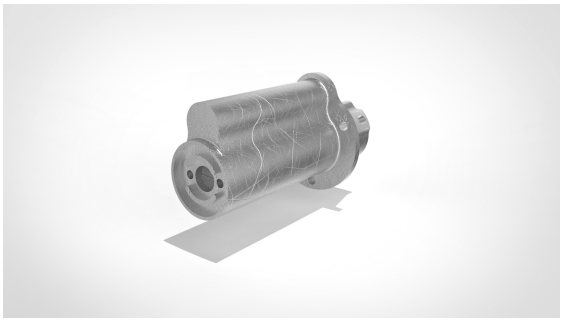


Figure B.2: Motor bushing of Lunar Zebro. Front side view where the motor shaft will protrude the bush. Above the motor there is a hollow shaft where a Hall-effect sensor can be inserted. The opening of this shaft is not visible.

options are considered. To bind the leg and the shaft in the most permanent way, a press fit is considered. However, a press fit is not recommended because high axial stresses will be present on the gearbox shaft during assembly. To avoid this disadvantage, the d-shaped hole could be made slightly larger and the shaft and leg can be glued together. This method is deemed insufficient regarding the extreme temperature requirements and possible outgassing of the glue.

A mechanical clamping connection can be realised in different ways. A bolt can be screwed through the leg onto the axle. However, the contact surface between the axle and the frontal side of the bolt is small and it is unsure if this will result enough friction force. An adaptation of this clamping design is proposed by Stellar Space Industries, where a bolt is used to decrease the diameter of the hole and create a clamping force. Now the the contact surface is as large as the motor shaft itself. A disadvantage of this solution is that there is a open space at the top of the leg where lunar regolith can find its way to the motor's bearing. This disadvantage is deemed surmountable in another iteration of the design.

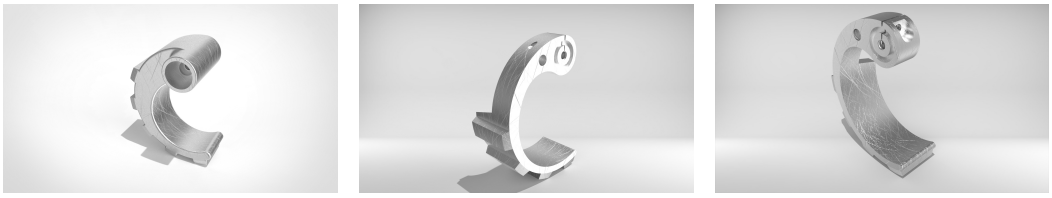
B.1.1. Dust Seal

Sealing the motor shaft and bearings from lunar regolith is in close relation with attaching the motor shaft to the leg. The ideal dust seal would be a 'wireless' torque conveyor. This can actually exist as a magnetic coupling where two disks with magnets are attached on the in- and outside of the body of Lunar Zebro. The disk on the inside of the rover is driven by the motor and the disk on the outside is attached to the leg and follows the movement of the inside disk. This ensures there are no holes in the body of Lunar Zebro and no lunar regolith can get in. Unfortunately, this coupling requires the motors to be on the inside of the body entirely, which creates a violation of the maximum volume of Lunar Zebro. Additionally, it is unclear whether or not this coupling will be able to transfer the necessary torque. The magnets can hypothetically be chosen such that the maximum torque the coupling can transfer, equals the maximum torque of the motor. This would protect the motor against any over-torque situations. Stronger magnets, however, increase the friction force between the disks and the body. The motion of the disk on the outside of the rover will also be hindered by lunar regolith over time.

Following from the volume constraints, it is decided to have as much of the length of the motor as possible outside of the rover, contained in a motor bushing. The middle motor bushings are the constraining factor in this design. They extend almost the full length of the motor outside of the rover. To seal the motor bearings as much as possible from the lunar dust, having part of the leg extend over the motor bushing would be ideal. This is shown in fig. B.3a. Dust must now travel horizontally between the leg and motor bushing to reach the bearing. This solution is not possible since a hall effect sensor is positioned above the motor bushing in close proximity of the leg. The motor bushing is shown in fig. B.2. In the design of the motor bush the opposite is done, the motor bush extends slightly in the leg. It resembles a very crude labyrinth seal.

A labyrinth seal is often used in rovers to protect gearboxes from dust. It consists of a certain amount of grooves inside of a bushing. More grooves result a better dust seal, but also require more precise machining and operation. Especially during operation, an elaborate labyrinth seal is expected to be unsuitable for Lunar Zebro. This is due to the intermittent radial and rotational loads inferred by the leg that cause elastic deformations of the motor shaft.

The result is a partial dust seal of two grooves. One groove on the motor bush shown in fig. B.2 and one on the leg shown in fig. B.3c. This final design does not yet incorporate a mitigation for dust getting into the opening of the leg as mentioned earlier.



(a) Early version Aldrin tread type with dust seal extending over motor bushing.

(b) Backside view of final Aldrin tread type leg.

(c) Frontside view of final Aldrin tread type leg.

Figure B.3: Prototypes of Aldrin tread legs.

B.2. Determining Absolute Leg Position - Magnet Placement

Placing a magnet on the leg is part of a solution to the requirement of having to know the absolute radial position of the leg. The absolute position of each leg shall be known in order to properly execute a step. Determining this absolute position can be done in different ways, as will be discussed.

Current Designs Currently, KiloZebro uses analogue Hall-effect sensors and a magnet mounted on the leg to position its legs. During a calibration phase, the leg is spun until its magnet passes the Hall-effect sensors on the body of KiloZebro. In this way the leg position is known at that time. After calibration, the legs are controlled using a relative quadrature pulse encoder mounted on the shaft of each motor. The quadrature pulse encoder can not provide absolute information about the leg position. On top of that the gearbox between the motor and the leg eliminates the direct connection between the leg and the motor shaft.

DeciZebro uses the same mechanism with one Hall-effect sensor inside the body and a magnet on each leg. Additionally, there is an absolute encoder present on the same shaft as the leg. The absolute encoder is made in-house and consists of a disk with small slits and an optical switch.

B.2.1. Optional Designs

There are many design options to fulfil the requirement of knowing the absolute position of a leg. All have their advantages and disadvantages.

Hall-effect Sensor Based Leg Positioning Applying the same method used in KiloZebro for Lunar Zebro would provide a cheap and simple solution. However, this requires a calibration phase in which the legs spin until the calibration point is reached. This has disadvantages since it can result in unwanted movement of Lunar Zebro. This method also requires a space graded relative encoder at the axle of the motor. If, however, a BLDC motor is used, it is possible to use the internal Hall-effect sensors of the motor to obtain information about the position of the motor shaft.

Absolute Encoder An absolute shaft encoder can be used to measure the position of the leg at any time. To be able to do this, the encoder is to be mounted on the axle that connects the leg with the motor. This could be the preferred solution for the problem since it has no obvious disadvantages like requiring a calibration cycle. Unfortunately, these type of encoders do not exist for Lunar Zebro's size, motors and environmental requirements. On top of that, if a coupling between the gearbox shaft and the leg is used, the absolute shaft encoder can provide false information given the possibility of slip.

Current Consumption Based Calibration The leg position can also be calibrated using the current consumption. When the leg comes in contact with the surface, the current drawn by the motor will increase and the position of the leg is more or less known. This method requires that Lunar Zebro is lying on its belly and the surface needs to be flat in order to provide an equal absolute reference point for each leg. A single irregularity in the surface can offset the calibration, resulting in inefficient walking.

B.2.2. Magnet Placement

Due to its simplicity, the Hall-effect sensor approach is taken. To maximise its performance, the magnet and Hall-effect sensor need to be placed as far away from the motor shaft as possible. This minimises the error when the magnet triggers the Hall-effect sensor. Again for simplicity, digital Hall-effect sensors are chosen. Figures B.3b and B.3c show a hole where the magnet will be inserted on the leg. It is shown on the left of the hole for the motor shaft.

Property	Lunar Soil
Average Particle size	60 µm up to 80 µm
Average Bulk density	1.45 g cm^{-3} up to 1.55 g cm^{-3}
Average Void Ratio	1 up to 1.14
Average Porosity	50 % up to 54 %
Average Relative density	65 %

Table B.1: Lunar soil characteristics.

B.3. Leg Width

Determining the width of Lunar Zebro's legs starts with understanding the interaction between the leg and the soil it is walking on. There is a lot of literature describing wheel-soil interaction on earth. There is also some literature describing wheel-soil interaction on the moon. The latter is interesting for Lunar Zebro.

B.3.1. Leg-soil Interaction

Because of the c-shape of the leg, the first part of a step is comparable to a wheel when it comes to soil interaction. Unfortunately, when the moving contact point comes closer to the toe of the leg, the leg stops behaving as a wheel and starts acting more and more as a single contact point.

The shape of the leg and the soil properties combined create the soil-leg interactions that determines how effective the Zebro moves. For conventional lunar Rovers with wheels, the soil-wheel interaction is well understood and described in [28]. However, the force applied by the c-shaped leg on the soil during walking is not as equally distributed as with a rolling wheel. The relationship between walking speed, the soil packing fraction and the leg's rotational speed are described in [35]. An important conclusion in this research is that very small changes of less than a per cent in soil packing fraction can greatly influence the walking speed of the robot. Unfortunately, these kinds of conclusions are not very helpful for designing for an environment of which the properties are not well known.

In [28, ch. 2], the shearing of the surface when a wheel drives over it is described. The shear surface appears before the wheel is actually present and is altered from a 45° angle to a horizontal position by the wheel traversing over it. From the c-shaped leg it is expected that when the contact point moves to the toe of the leg, the soil will shear an extra time at a 45° angle. The intuition behind the idea that the soil will shear one extra time is as follows. When a leg sinks into the soil closer to the hip, it is clear that it can be regarded as a wheel since an certain arc-length of the leg is in contact with the soil. The mass induced by Lunar Zebro is distributed over this arc length. Further along in the step, when the contact point gets closer to the toe, the mass is distributed over a increasingly smaller area until the contact point becomes a line at the toe of the leg. This will induce the extra shearing.

According to [27, ch. 9], the average **bulk density** of the lunar soil is 1.45 up to 1.55 g/c^{3}m. ("The bulk density, ρ, of soil is defined as the mass of the material contained within a given volume, usually expressed in grams per cubic centimetre.") The average **porosity** is 50 up to 54 %. ("The porosity, n, is defined as the volume of void space between the particles divided by the total volume.") The average **void ratio** is 1 up to 1.14. ("... which is equal to the volume of void space between the particles divided by the volume of the "solid" particles (again, including the sub granular porosity).") The **relative density** is estimated to be 65 % (medium dense to dense) in the layer up to 15 cm deep and 90 % (very dense) from a depth of 30 cm on-wards. Relative density is a measure of density relative to the minimum and maximum density of the material in question. If a substance cannot be stacked more dense without breaking molecules, the relative density is 100 %. However, the properties of the lunar soil differ on each location. The average lunar soil properties as just described are shown in table B.1.

When it comes to moving around on the moon, [27] says the following: "From the experience of the Apollo and Lunokhod missions, we now know that almost any vehicle with round wheels will perform satisfactorily on the lunar surface, provided the ground contact pressure is no greater than about 7–10 kPa." Additionally it also mentions the pressure from the Lunar Module's foot-pads during the Apollo 11 mission to be 5 kPa.

Two instances where even this pressure was too high are also described in [27]. The Apollo 15 LRV got stuck once, but due to its low weight on the moon's surface, the astronauts could easily move it by hand. Another example is the Lunokhod 2 that sank into the lunar regolith for more than 20 cm at the toe of a crater. From this it is concluded that having the contact pressure on the safe side (lower) increases the chances of Lunar Zebro being able to operate anywhere on the lunar surface.

During the research phase for the Apollo missions, Cinder Field in Arizona was used as an analogue training area for astronauts. It is regarded as a direct analogue of the Moon's surface and was used for

testing rovers. Due to volcanic activity in this region around 1064 A.D., the soil is a mixture of basalt and clay. From research website [45], it can be concluded that volcanic terrains are the best terrestrial analogues to the lunar soil.

B.3.2. Leg Width Calculation

Determining the width of the leg is all about knowing the sinkage of the leg. If the leg sinks into the lunar regolith as far as its own diameter, Lunar Zebro will resemble more of a swimming robot than a walking one. To prevent this, the leg must have a certain width, that ensures the mass of Lunar Zebro is spread over an area, such that the pressure on the lunar regolith stays well below 7 kPa cm^{-1} as described in appendix B.3.1.

Due to the mass and volume restrictions from outside this project, the body will have a maximum length of around 20 cm and height of 5 cm. Because of this, the legs are required to have a diameter of 5.3 cm. The goal is to keep the sinkage of the leg at a minimum, but with 1 cm being the maximum allowable sinkage, the footprint length becomes 4.15 cm.

There are roughly two ways of calculating the sinkage of the legs, or any structure. Equation (B.1) formulated by [6] describes the sinkage of a wheel. This equation is referenced by [27] along with the recommended values for k_c (modulus of soil deformation, 0.14), k_ϕ (frictional modulus of soil deformation, 0.82) and n (1) to calculate k (soil consistency). The force (W) that is applied by the mass of Lunar Zebro is calculated to be the entire mass of Lunar Zebro in the gravity field of the moon. On top of that it is assumed that there should be a maximum sinkage of 1 cm. With all this data, the necessary width of the leg can be calculated. This width is 0.55 cm.

$$
\begin{aligned}
\text{WheelSinkage } z\ [cm] &= \left(\frac{W\ [N]}{A\ [cm^2] \cdot k\ [kPa/cm]}\right)^{\frac{1}{n}} \\
&\text{Where :} \\
W\ [N] &= m\ [kg] \cdot g\ [m/s^2] \\
k\ [kPa/cm^{n+2}] &= \frac{k_c\ [N/(cm)^{n+1}]}{\text{LegWidth } [cm]} + k_\phi\ [N/(cm)^{n+2}] \\
A\ [cm^2] &= \text{FootprintLength } [cm] \cdot \text{LegWidth } [cm] \\
\text{FootprintLength } [cm] &= \left(2 \cdot r\ [cm] \cdot \sin\left(\arccos\left(\frac{r\ [cm] - z\ [cm]}{r\ [cm]}\right)\right)\right)
\end{aligned}
\tag{B.1}
$$

The width of Lunar Zebro's legs can also be calculated using static allowable bearing pressure. A disadvantage of the static allowable bearing pressure is that it is proven only with boot prints of astronauts rather than something resembling the legs of Lunar Zebro. The static allowable bearing pressure is describe in eq. (B.2). The allowable bearing pressure q_all is described as a constant k times the maximum acceptable settlement d_{acc}, in this case 1 cm [27, p.519].

If q_{all} is set equal to $\frac{W}{A}$, again the required width of the leg can be calculated. One assumption must be made, the value of k, the modulus of subgrade reaction. In [27] it is mentioned that for the Lunar Rover Vehicle, k should stay below 7 kPa cm^{-1}. However, to reach a higher confidence interval of 95 % -the lunar regolith can differ from place to place- [27] recommends a value for k of 2 kPa cm^{-1}. Using these numbers, the width is calculated as shown in eq. (B.3) to be 0.84 cm or 2.94 cm with the higher confidence interval.

$$
q_{all}\ [kPa] = k\ [kPa/cm] \cdot d_{acc}\ [cm] \tag{B.2}
$$

$$
\text{LegWidth } [m] = \frac{W\ [N]}{k\ [kPa/cm] \cdot d_{acc}\ [cm] \cdot \text{FootprintLength } [m]} \tag{B.3}
$$

The wheel-sinkage method and the static allowable bearing pressure method with low confidence interval agree quite well on the leg width. The static allowable bearing pressure with high confidence interval results a more than three times higher leg width. This is in agreement with the observation made in [27] directly after the wheel-sinkage method, where the critical condition soft soil is discussed. A recommendation is done to take soft soils into account when designing future lunar surface vehicles. This is exactly what is done with the static allowable bearing pressure method with higher confidence interval. Unfortunately, this results a very wide leg of almost 3 cm. The volume budget for the mission does not allow legs of this width. It is recommended to try to get as close to the width of 2.94 cm as possible. Currently, a width of 2 cm is realised.

B.4. Leg Tread

There is extensive research concerning wheel-soil interactions. For terrestrial vehicles [6] is often referenced. Inspired by this research, [28] conducted research on the same topic but for the lunar environment. This literature can shed a light on the pull (or traction) of wheels with different types of tread.

Unfortunately, time does not permit an analytic study of the influence of different types of tread on the traction of the leg. A practical approach is taken by creating four different types of tread and having [49] test these in suitable conditions. The treads are shown in figs. B.3 and B.4.

Box The first tread type is called a box tread and is shown on the left in fig. B.4. The rationale behind this design is that soil in and under the boxes becomes more compact compared to open treads where soil can escape from the pressure to the sides of the leg. Due to the irregular shape of the grains in lunar regolith, it is thought that compacting these grains makes them cut into each other and increase mutual friction. Compacting the soil is at the same time the greatest weakness of this design. Although the boxes are shaped trapezoidal to decrease the likeliness of soil being stuck in the boxes, it is still feared the boxes will fill up with soil. This can be due to the soil being statically charged and sticking to the leg, or the compaction being so efficient that the sharp grains eat their way into the material of the leg and stay there.

Dune The dune type of tread is based on the tread found on tires used on beach buggies that race through the dunes. These buggies manoeuvre through very fine and loose soil which is the closest comparison to lunar regolith found in nature on earth. The tread is shown in the middle of fig. B.4. On these buggies the front wheels often have a different tread to increase the ability to steer. Lunar Zebro's legs only need traction in one direction and thus this tread suffices for all legs.

Nanokhod The Nanokhod tread is based on the Nanokhod rover developed by ESA/ESTEC [34]. This rover is build to function on Mercury with temperatures down to −180 °C. Unfortunately, the Bepi-Colombo mission that would take the rover was altered to not include a lander and rover. The rover is proposed as suitable for the dark side of the moon as well.

The rover is based on a Russian predecessor and is basically a tank. This tank type of tread can also be applied to the legs of Lunar Zebro for obvious reasons and is shown in fig. B.4 on the left.

Aldrin The last tread type is modelled after the boots of Apollo astronauts, hence the name. The tread type is shown in figs. B.3b and B.3c. It is an elongated version of the Nanokhod type of tread. Confidence that the tread type can create enough traction is build on the fact that the design of the boots did not change much over the course of the Apollo missions.

B.4.1. Final Design

All designs are manufactured using 3D printing techniques and are ready to be tested. Unfortunately, only the Aldrin tread type is tested in [49]. In this test the Aldrin tread design performs satisfactory with a leg width of 2 cm. Because of a lack of other test, this type of tread is chosen for the final design that is shown in figs. B.3b and B.3c.

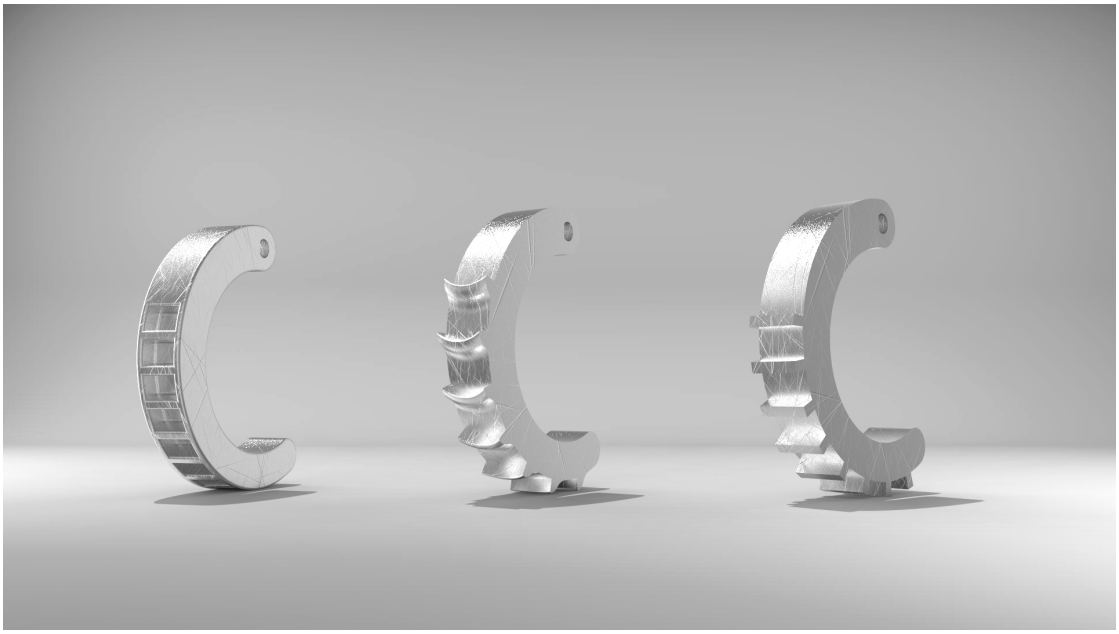


Figure B.4: From left to right, Box, Dune and Nanokhod tread.

Bibliography

[1] Spaceflight 101. Chinese Yutu Moon Rover Pronounced Dead after Record-Setting Mission, 2016. URL http://spaceflight101.com/change/chinese-yutu-moon-rover-pronounced-dead/. [Online; accessed 21-June-2018].

[2] Marnix P. Abrahams, Daley L. Adrichem, Daniël J. Booms, Piet N.T. De Vaere, and Alex F. Oudsen. Designing a Robust Robot Leg Module. Technical report, TU Delft, 2016.

[3] J. H. Adams, A. F. Barghouty, M. H. Mendenhall, R. A. Reed, B. D. Sierawski, K. M. Warren, J. W. Watts, and R. A. Weller. CREME: The 2011 Revision of the Cosmic Ray Effects on Micro-Electronics Code. *IEEE Transactions on Nuclear Science*, 59(6):3141–3147, Dec 2012. ISSN 0018-9499. doi: 10.1109/TNS.2012.2218831.

[4] National Aeronautics and Space Administration. NASA-GB-8719.13 - NASA Software Safety Guidebook. Technical report, National Aeronautics and Space Administration, March 2004.

[5] Alex Allain. Lambda Functions in C++11 - The Definitive Guide, 2017. URL https://www.cprogramming.com/c++11/c++11-lambda-closures.html. [Online; accessed 27-July-2018].

[6] Mieczyslaw G. Bekker. Introduction to Terrain-Vehicle Systems. Part I: The Terrain. Part II: The Vehicle. Technical report, Michigan University Ann Arbor, 1969.

[7] S. Bourdarie and M. Xapsos. The Near-Earth Space Radiation Environment. *IEEE Transactions on Nuclear Science*, 55(4):1810–1832, Aug 2008. ISSN 0018-9499. doi: 10.1109/TNS.2008.2001409.

[8] Hyperion Technologies B.V. CP400.85, 2018. URL https://hyperiontechnologies.nl/products/cp400-85-processing-platform/. [Online; accessed 01-August-2018].

[9] Verum B.V. Verifying models, 2018. URL https://www.verum.com/supportitem/verifying-models/. [Online; accessed 18-August-2018].

[10] Verum Software Tools B.V. About Dezyne, 2017. URL https://www.verum.com/dezyne/. [Online; accessed 02-March-2018].

[11] Verum Software Tools B.V. Dezyne External Tutorial, 2017. URL https://www.verum.com/supportitem/how-to-use-external/. [Online; accessed 25-July-2018].

[12] Verum Software Tools B.V. Glossary, 2017. URL https://www.verum.com/supportitem/glossary/. [Online; accessed 04-April-2018].

[13] Verum Software Tools B.V. Code Integration Tutorial, 2017. URL https://www.verum.com/supportitem/implementing-and-integrating-native-components/. [Online; accessed 30-July-2018].

[14] Verum Software Tools B.V. Dezyne Syntax, 2017. URL https://www.verum.com/supportitem/identifiers/. [Online; accessed 18-July-2018].

[15] Verum Software Tools B.V. Thread-Safe Shell, 2017. URL https://www.verum.com/supportitem/thread-safe-shell/. [Online; accessed 27-July-2018].

[16] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36742-7.

[17] Jurriaan de Groot. Swarm Behaviour for the Zebro Robot. Master's thesis, Delft University of Technology, November 2017.

[18] P.N.T. De Vaere, M.P. Abrahams, D.L. Adrichem, D.J. Booms, and A.F. Oudsen. Designing a Robust Robot Leg Module. Technical report, Delft University of Technology, June 2016.

[19] Zebro Team TU Delft. Roots, 2018. URL http://zebro.org/roots. [Online; accessed 16-May-2018].

[20] Technische Universiteit Eindhoven. mCRL2 Analysing System Behaviour, 2018. URL http://www.mcrl2.org/web/user_manual/index.html. [Online; accessed 02-March-2018].

[21] Rien Elling, Bas Andeweg, Jaap de Jong, and Christine Swankhuisen. *Rapportagetechniek*. Noordhoff Uitgevers, third edition edition, 2005.

[22] D. Falguere, D. Boscher, T. Nuns, S. Duzellier, S. Bourdarie, R. Ecoffet, S. Barde, J. Cueto, C. Alonzo, and C. Hoffman. In-Flight Observations of the Radiation Environment and its Effects on Devices in the SAC-C Polar Orbit. *IEEE Transactions on Nuclear Science*, 49(6):2782–2787, Dec 2002. ISSN 0018-9499. doi: 10.1109/TNS.2002.805380.

[23] R. Feldt, R. Torkar, E. Ahmad, and B. Raza. Challenges with Software Verification and Validation Activities in the Space Industry. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 225–234, April 2010. doi: 10.1109/ICST.2010.37.

[24] European Cooperation for Space Standardization. ECSS-E-ST-10-06C - Space Engineering - Technical Requirements Specification. Technical Report Third Issue, European Space Agency, March 2009.

[25] European Cooperation for Space Standardization. ECSS-E-HB-40-A - Space Engineering - Software engineering handbook. Technical Report First Issue, European Space Agency, December 2013.

[26] C. M. Fuchs, T. P. Stefanov, N. M. Murillo, and A. Plaat. Bringing Fault-Tolerant GigaHertz-Computing to Space: A Multi-stage Software-Side Fault-Tolerance Approach for Miniaturized Spacecraft. In *2017 IEEE 26th Asian Test Symposium (ATS)*, pages 100–107, Nov 2017. doi: 10.1109/ATS.2017.30.

[27] Grant Heiken, David Vaniman, and Bevan M. French. *Lunar Sourcebook: A User's Guide to the Moon*. CUP Archive, 1991.

[28] H.J. Hovland. Mechanics of Wheel-Soil Interaction. Technical report, NASA, 1973.

[29] Bart Kersbergen, Gabriel A.D. Lopes, Ton J.J. van den Boom, Bart de Schutter, and Robert Babuška. Optimal Gait Switching for Legged Locomotion. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'11)*, pages 2729–2734. IEEE, Sept. 2011.

[30] Hans Keur and Roel Mouris. Zebro Drive System. Technical report, TU Delft, 2015.

[31] Laurens D. Kinkelaar. Adaptive Gait Switching Control Structure using Max-Plus Systems in Legged Locomotion. Master's thesis, Delft University of Technology, May 2018.

[32] S. Klinkner, C.G.-Y. Lee, C. Wagner, M. Lengowski, H.-P. Röser, and P. Bourlier. Experiences Gained from the Thermal Vacuum Tests of the Microrover Nanokhod. In *10th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, November 2008.

[33] NASA Jet Propulsion Laboratory. Discovery Guide: Mars Rover Curiosity, 2018. URL https://www.jpl.nasa.gov/education/marsrover.cfm#7. [Online; accessed 21-June-2018].

[34] C.G.-Y. Lee, S. Klinkner, W. Hlawatsch, A.-M. Schreyer, H.-P. Röser, J. Schaefer, A. Schiele, and J. Romstedt. Mercury Nanokhod Rover - Hardware Realisation and Testing. In *9th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, November 2006.

[35] Chen Li, Paul B. Umbanhowar, Haldun Komsuoglu, Daniel E. Koditschek, and Daniel I. Goldman. Sensitive Dependence of the Motion of a Legged Robot on Granular Media. *Proceedings of the National Academy of Sciences*, 106(9):3029–3034, 2009.

[36] Gabriel A.D. Lopes, Bart Kersbergen, Ton J.J. van den Boom, Bart de Schutter, and Robert Babuska. Modeling and Control of Legged Locomotion via Switching Max-Plus Models. *IEEE Transactions on Robotics*, 30(3):652–665, 2014.

[37] Gabriel A.D. Lopes, Robert Babuška, Bart de Schutter, and Ton J.J. van den Boom. Switching Max-Plus Models for Legged Locomotion. In *Proceedings of the 2009 IEEE International Conference on Robotics and Biomimetics (ROBIO 2009)*, pages 221–226. IEEE, Dec. 2009.

[38] Jeffrey B. Miog. Design of the Locomotion Sub-System for Lunar Zebro. Master's thesis, Delft University of Technology, August 2018.

[39] Inc. Modicon. Modicon Modbus Protocol Reference Guide, 1996. URL http://modbus.org/docs/PI_MBUS_300.pdf. [Online; accessed 09-March-2018].

[40] E.Z. Moore, D. Campbell, F. Grimminger, and M. Buehler. Reliable Stair Climbing in the Simple Hexapod RHex. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 3, pages 2222–2227. IEEE, 2002.

[41] Indian Space Research Organisation. GSLV-F10/Chandrayaan-2 Mission, 2017. URL https://www.isro.gov.in/gslv-f10-chandrayaan-2-mission. [Online; accessed 23-July-2018].

[42] Mattijs Otten. DeciZebro: the Design of a Modular Bio-Inspired Robotic Swarming Platform. Master's thesis, Delft University of Technology, July 2017.

[43] Science Magazine Pallava Bagla. India Plans Tricky and Unprecedented Landing near Moon's South Pole, 2018. URL http://www.sciencemag.org/news/2018/01/india-plans-tricky-and-unprecedented-landing-near-moon-s-south-pole. [Online; accessed 23-July-2018].

[44] Jiantao Pan. Software Reliability, 1999. URL https://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/. [Online; accessed 05-August-2018].

[45] Louisa Preston, Monica Grady, and Simeon Barber. The Catalogue of Planetary Analogues. Technical report, The Planetary and Space Sciences Research Institute, The Open University, UK, 2012.

[46] LCC Quantum Leaps. Modern Embedded Systems Programming, 2018. URL https://www.state-machine.com/. [Online; accessed 24-July-2018].

[47] Floris Rouwen. Motor Control For KiloZebro. Document can be requested from the Zebro Project., June 2017.

[48] Esterel Technologies S.A. Scade Suite, 2014. URL http://www.esterel-technologies.com/products/scade-suite/. [Online; accessed 16-August-2018].

[49] K. Sharma. Adaptation Study of Zebro as Nano Rover for Lunar Exploration and Demonstration of Locomotion on Simulated Lunar Surface. Master's thesis, Delft University of Technology, January 2018.

[50] William Suriana. Turning of a Legged Robot via a Switching Max-Plus Linear System: Simulation and Implementation. Master's thesis, Delft University of Technology, October 2017.

[51] Hamid A. Toliyat and Gerald B. Kliman. *Handbook of Electric Motors*, volume 120. CRC press, 2004.

[52] Volkswagen. Schoner Rijden Dankzij BlueMotion Technologies, 2018. URL https://www.volkswagen.nl/over-volkswagen/duurzaamheid/schoner-rijden. [Online; accessed 14-March-2018].

Acronyms

ADC
Analog to Digital Converter 7

BLDC
Brush-Less Direct Current 29, 43, 44, 55

BMS
Battery Management System 5

COTS
Commercial Off-The-Shelf 7, 8

CRC
Cyclic Redundancy Check 11, 27, 32

CSP
Communicating Sequential Processes

ECSS
European Cooperation for Space Standardisation 2, 8, 35

EEMCS
Electrical Engineering, Mathematics and Computer Sciences 5

ESA
European Space Agency iii, 7–9, 35

FDR
Failures-Divergences Refinement

FIFO
First-In-First-Out 33, 34

FPGA
Field-Programmable Gate Array 7

ISRO
Indian Space Research Organisation 1

JPL
Jet Propulsion Lab 6

LC
Locomotion Controller iii, 3–6, 8, 10, 17–20, 22, 23, 25–27, 29, 32, 33, 36, 37

LSS
Locomotion Sub-System iii, 1–3, 5–15, 17–22, 26, 29, 31–33, 35–37, 39

mCRL2
Micro Common Representation Language 2 8

NASA
National Aeronautics and Space Administration 2, 6

OBC
On Board Computer 6, 7, 9–11, 13, 17–20, 22–27, 31–33, 36, 37

PNS
Peripheral Nervous System 4

SEE
Single Event Effects 7

TID
Total Ionising Dose 7

Zebro
ZEs-Benige RObot iii, 1, 3–5

List of Figures

List of Tables