Technische Universität München
TUM School of Engineering and Design

Specification of Distributed Computing for Small Satellite Control

Sebastian Rückerl

Vollständiger Abdruck der von der TUM School of Engineering and Design der Technischen Universität München zur Erlangung eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing. Christoph Holst

Prüfer*innen der Dissertation:

1. Prof. Dr. Ulrich Walter
2. Prof. Dr. Martin Schulz
3. Prof. Dr.-Ing. Carsten Trinitis

Die Dissertation wurde am 4. November 2022 bei der Technischen Universität München eingereicht und durch die TUM School of Engineering and Design am 2. Mai 2023 angenommen.

Acknowledgements

First, I would like to express my deepest gratitude to Prof. Walter, without whom this thesis would not be possible. Thank you for putting your trust in the work of your Ph.D. students and for providing the freedom and autonomy required to develop my ideas and approaches to the challenging topics of this thesis. At the same time, he provided great feedback and new ideas supporting the development of this thesis.

I am also extremely grateful to my mentor Carsten Trinitis. Thank you for your valuable second opinion on my ideas and your support with the supervision of various student theses.

This endeavor would not have been possible without the support of many people from different parts of TUM. I could not have finished this journey without Thomas Pöschl and Martin Losekamm. Thank you for the countless hours you spent explaining particle physics and radiation in space environments to me made. Without this support, this thesis, especially the radiation tests, would not have been possible. Furthermore, I am also thankful to all other Professors and Ph.D. students at TUM that supported me with the supervision of many student theses and provided valuable insight and new ideas regarding my research.

My special thanks also go to the LRT staff. Martin Rott and Uta Fellermair, thank you for always supporting me with any bureaucratic hurdles. Nicolas Appel, thank you for all the hours of drinking coffee. During this time we had many great (and many not-so-great) ideas that made me enjoy the time and provide the energy to tackle the various tasks required for this thesis and beyond. I would like to extend my thanks to all the other members of the LRT for a great time we had during the last years.

I would also like to thank the Munich Orbital Verification Experiment (MOVE) team and its former and current members. I started working on space-related topics only because of MOVE and Martin Langer. Finally, the MOVE students, especially the students of the current MOVE software team, had to endure various changes in the software framework developed throughout this thesis. Thank you for all the great ideas and patience with me breaking your software over and over again.

I am also very grateful to all the people proofreading this thesis. Especially to my father, Reinhard Rückerl, and my wife, Marion Rückerl, who read the entire thesis and provided valuable feedback.

Finally, I would also like to thank my family for their support. During the many years working on my Ph.D., they provided the required support and pressure to successfully finish. Special thanks to my wife who had to endure my mood during the last months of writing this thesis.

Abstract

The Technical University of Munich (TUM) will launch multiple satellite missions in the upcoming years. While all of these missions use CubeSats, they have diverging requirements from short time missions over long time missions with restrictive pointing requirements to reliable operation in the radiation environment of the South Atlantic Anomaly (SAA). Due to the tight schedule and the concurrent development of these missions, the reuse of components is essential for successful missions. We suggest a distributed system as a potential solution and expect such a system to provide the required flexibility and reusability.

While a wide range of small satellite on-board computers and data handling software frameworks exist in literature, only a few are actually available. For CubeSats, it is still common to use a centralized control setup. While advanced frameworks for distributed control and data handling exist, they target larger system with less restricted resources. None of these frameworks provides timing guarantees and enables the implementation of a distributed real-time control system.

The main research question is thus, if a distributed system is possible on a resource restricted CubeSat and if this distributed system improves flexibility and adaptability, and thus encourages the reuse between missions. At the same time, we also want to show that such a system can meet the timing requirements of a distributed real-time control system such as an attitude determination and control system (ADCS).

As previous frameworks for on-board software exist, we first select RODOS as basic operating system (OS). Using the available features of RODOS, we design a framework — called the DOSIS framework — that provides a simple, flexible, yet strongly type-checked interface for developers and contains the necessary features for distributed real-time control. An implementation of the framework and the RODOS ports for the STM32L4 and VA41620 microcontroller provide a basic prototype. A time synchronization test verifies the frameworks clock synchronization and the remaining timing error for timed sensor readout or actuator activation. A radiation test with protons additionally verifies that the VA41620 satisfies the requirements for operation in the SAA.

The DOSIS framework's implementation provides an interface for developers that simplifies the implementation of new software components. These components can be easily adapted and shared between missions and thus enable the reuse of major parts of the flight software. The time synchronization test verifies the real-time capabilities. The remaining timing uncertainty is below 2 ms — even with low-quality references for the local clock on each node — and thus sufficient even for ADCS applications. Finally, the radiation tests show that the VA41620 successfully corrects bit errors caused by proton irradiation. Thus, we expect this controller to operate in the SAA without software observable issues.

Within this thesis, we show that a distributed system for CubeSat on-board control is possible and can satisfy real-time requirements. Furthermore, a target platform exists that is well capable of reliable operation in the SAA. Thus, we find the DOSIS framework presented in this thesis as a viable option for future CubeSats at TUM.

Zusammenfassung

In den kommenden Jahren werden an der Technischen Universität München (TUM) mehrere Satellitenmissionen innerhalb kurzer Zeit stattfinden. Die Anforderungen dieser Missionen unterscheiden sich dabei deutlich: So gibt es Missionen mit nur kurzer Missionsdauer, eine Mission welche eine stabile Ausrichtung über längere Zeiträume benötigt, sowie eine Mission welche einen zuverlässigen Betrieb im Strahlungsmaximum der Südatlantischen Anomalie (SAA) erfordert. Aufgrund des straffen Zeitplans ist das Wiederverwenden von Einzelkomponenten sowie das Anpassen vorhandener Komponenten notwendig. Diese Arbeit stellt ein verteiltes System zur Steuerung von Kleinsatelliten als ein wiederverwendbares und anpassbares System und damit als Lösung für dieses Problem dar.

In der Literatur werden viele Softwarepakete für Satelliten erwähnt, jedoch ist nur ein kleiner Teil davon tatsächlich für Missionen an der TUM verfügbar. Des Weiteren nutzen die meisten CubeSats eine zentrale Komponente zur Steuerung des restlichen Satelliten. Verteilte Systeme, insbesondere solche die eine verteilte Regelung ermöglichen, finden sich dabei zumeist nur für größere und weniger ressourcenbeschränkte Systeme.

Ziel dieser Arbeit ist es zu zeigen, dass ein verteiltes System für CubeSats möglich ist und tatsächlich die Flexibilität bietet, um für zukünftige Missionen der TUM verwendet zu werden. Zusätzlich zeigt diese Arbeit, dass ein solches System in der Lage ist auch Echtzeitanforderungen eines Lagebestimmungs- und -reglungssystems (ADCS) erfüllen zu können.

Der Systementwurf beginnt mit der Wahl von RODOS als verwendetes Betriebssystem (OS). Anschließend wird das sogenannte DOSIS Framework mit allen notwendigen Funktionen für ein verteiltes Echtzeitkontrollsystem, insbesondere mit einfach zu nutzendem Interface sowie einer hinreichend genauen Zeitsynchronisation, entworfen sowie die Implementierung des Frameworks und der zugehörige Hardware-Prototyp vorgestellt. Die Funktionalität der Zeitsynchronisation sowie der zeitlich gesteuerten Abfrage von Sensordaten und Aktivierung der Aktuatoren wird anhand eines Prototypen aufgezeigt. Mit einem Strahlentest wird anschließend die Tauglichkeit des VA41620 Mikrocontroller für den Einsatz in der SAA untersucht. Insbesondere die Fähigkeit der Hardware Bitfehler direkt zu korrigieren wird dabei genauer analysiert.

Die Tests zur Zeitsynchronisation zeigen, dass die verbleibende Zeitunsicherheit für das Auslesen von Sensoren sowie das Ansteuern von Aktuatoren auch unter Verwendung eines ungenauen Taktgebers an jedem Knoten kleiner als 2 ms und damit für ein CubeSat ADCS ausreichend ist. Auch die Strahlentests konnten die angenommene Tauglichkeit der Hardware bestätigen. Obwohl einige Bitfehler durch Protonen verursacht wurden, konnte die Hardware diese wie erwartet beheben. Die Funktionalität des VA41620 auch unter extremen Bedingungen wird dadurch gezeigt und es wird erwartet, dass dieser Mikrocontroller auch in der SAA fehlerfrei betrieben werden kann.

In dieser Arbeit wird damit bestätigt, dass ein verteiltes System zum Steuern eines CubeSats eine vielversprechende Option darstellt und die Echtzeitanforderungen eines ADCS Systems erfüllen kann. Auch eine für CubeSats geeignete Hardware, welche das vorgeschlagene Framework unterstützt und den fehlerfreien Betrieb in der SAA ermöglicht, ist vorhanden. Das DOSIS Framework ist daher ein guter Kandidat als Framework für die Flugsoftware zukünftiger CubeSats der TUM.

Contents

Acronyms

ADC	Analog To Digital Converter
ADCS	Attitude Determination and Control System
Al	Aluminum
API	Application Programming Interface
C	Carbon
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-Rate
CAST	China Academy of Space Technology
CCSDS	Consultative Committee for Space Data Systems
CDH	Command and Data Handling
CMOS	Complementary Metal-Oxide-Semiconductor
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
Cu	Copper (from Latin Cuprum)
DAC	Digital To Analog Converter
DLR	German Aerospace Center (from German Deutsches Zentrum für Luft- und Raumfahrt)
DOSIS	Distributed Operating System Initiative for Satellites
ECSS	European Cooperation for Space Standardization
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIA	Electronic Industries Alliance
eMMC	Embedded Multimedia Card
EPS	Electrical Power System
ESA	European Space Agency
Fe	Iron (from Latin Ferrum)
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FR-4	Flame Retardant 4 (Fiber Glass Epoxy)
FRAM	Ferroelectric Random Access Memory
GNSS	Global Navigation Satellite System
GPIO	General Purpose Input/output
GPO	General Purpose Output
H	Hydrogen
He	Helium
I	Integral
IP	Internet Protocol

ISIS Innovative Solutions in Space
ISO International Organization for Standardization
ISS International Space Station
I²C Inter Integrated Circuit

LED Light Emitting Diode
LEO Low Earth Orbit
LET Linear Energy Transfer
LRSM Laboratory for Rapid Space Missions
LRT Institute of Astronautics (from German Lehrstuhl für Raumfahrttechnik)

MCU Micro Controller Unit
MISRA Motor Industry Software Reliability Association
MMU Memory Management Unit
MPSoC Multiprocessor System On a Chip
MRAM Magnetoresistive Random Access Memory

NASA National Aeronautics and Space Administration

O Oxygen
OBC On-Board Computer
OBDH On-Board Data Handling
ORIGINS DFG Cluster of Excellence ORIGINS
OS Operating System
OSI Open Systems Interconnection

P Proportional
PCB Printed Circuit Board
PDP Payload Data Processor
PI Proportional Integral
PSI Paul Scherrer Institute

RAM Random Access Memory
ROM Read-Only Memory
RS Recommended Standard
RTOS Real-Time Operating System

SAA South Atlantic Anomaly
SEE Single Event Effect
SI International System of Units (from French Système International d'Unités)
SiO_2 Silicon Dioxide
SML Space Missions Laboratory
SoC System On a Chip
SPENVIS Space Environment Information System
SPI Serial Peripheral Interface

TAI International Atomic Time (from French Temps Atomique International)
TCP Transmission Control Protocol
TIA Telecommunications Industry Association
TID Total Ionizing Dose
TM/TC Telemetry and Telecommand
TMR Triple Modular Redundancy
TUM Technical University of Munich

TüV Technical Inspection Association (from German Technischer Überwachungsverein)

UART Universal Asynchronous Receiver-Transmitter
UDP User Datagram Protocol
UHF Ultra High Frequency
UL Underwriters Laboratory
UML Unified Modeling Language
USB Universal Serial Bus
UUT Unit Under Test

VHF Very High Frequency

XML Extensible Markup Language
xTEDS Extended Transducer Data Sheet

Glossary

AFIS Antiproton Flux in Space (AFIS) is one of the LRSM CubeSat missions with an estimated launch date in 2024. The scientific goal of AFIS is the measurement of antiprotons within the SAA.

cFS NASA's core Flight System (cFS) is an open-source software framework for spacecraft. It is available online at https://github.com/nasa/cFS.

ComPol Compton Polarimeter (ComPol) is one of the LRSM CubeSat missions. The scientific goal of ComPol is the measurement of the polarization of X-rays originating from the Cygnus X-1 binary system.

COrDeT The Component Oriented Development Techniques (COrDeT) framework by P&P Software is a service-oriented software framework compatible with ESA standards. Documentation and a reference implementation is available online at https://www.pnp-software.com/cordetfw/.

CSP CubeSat Space Protocol (CSP) is a network-layer protocol for CubeSats. Documentation and a C-implementation is available online at www.libcsp.org.

DEDRA The DEbris Density Retrieval & Analyises (DEDRA) sensor is a detector for dust particles in space. It is developed at TUM as a modified and improved version of the MDC [160]. The DEDRA sensor is the main payload of the MOVE-III CubeSat.

F' F Prime (F') is a framework for spaceflight applications developed by NASA. F' is an open-source framework available at https://nasa.github.io/fprime/.

first-MOVE First CubeSat mission of the MOVE project. The 1 U CubeSat first-MOVE was launched into a LEO on November 21, 2013 [104].

fsfw The Flight Software Framework (fsfw) is a control software developed by the University of Stuttgart as part of the Flying Laptop Project. The source code for fsfw is available at https://egit.irs.uni-stuttgart.de/fsfw/fsfw.

GERICOS GEneRIC Onboard Software (GERICOS) is a framework for reusable software components for payload flight software developed by LESIA [169].

IOV-1 In-Orbit Verification Experiment 1 (IOV-1) is a LRSM mission for in-orbit verification of hardware developed for the AFIS and ComPol missions. IOV-1 is an ISS based experiment with an expected launch mid 2023.

MDC The Munich Dust Counter (MDC) is a dust particle detector developed at TUM for the MUSES-A mission [76].

MesoMag Mesosphere Magnetometry (MesoMag) is a proof-of-principle experiment on IOV-1 for a future LRSM mission measuring Earth's magnetic field.

MOVE The Munich Orbital Verification Experiment (MOVE) project is an educational project for satellite development at TUM. It is a joint effort between the LRT and the scientific workgroup of rocketry and spaceflight.

MOVE-BEYOND Development program within the MOVE project for a satellite bus as successor of MOVE-II

MOVE-II Second CubeSat mission of the MOVE project. MOVE-II was launched into a LEO on December 3, 2018 and is still in operation.

MOVE-IIb Third CubeSat mission of the MOVE project and sister-satellite of MOVE-II. MOVE-IIb was launched into a LEO on July 5, 2019.

MOVE-III Upcoming CubeSat mission of the MOVE project. MOVE-III's latest design is a 6 U CubeSat featuring the DEDRA sensor as its payload [160].

MQTT MQTT is a publish-subscribe, machine to machine network protocol. Additional information about this protocol is available online at https://mqtt.org/.

NMF Nanosat Mission Operations Framework (NMF) is a CCSDS compatible framework for small satellite software. It was developed by Graz University of Technology and is maintained by ESA. The source code and documentation is available at https://nanosat-mo-framework.github.io/.

OBC-NG On Board Computer — Next Generation (OBC-NG) is a framework for distributed data processing on high performance COTS hardware developed by DLR [118].

RODOS Realtime On-board Dependable Operating System (RODOS) is an open-source operating for small embedded devices developed by the Universit of Würzburg. RODOS is available online at https://gitlab.com/rodos/rodos.

RTEMS Real-Time Executive for Multiprocessor Systems (RTEMS) is an open-source RTOS available at https://www.rtems.org/.

ScOSA Scalable On-Board Computing for Space Avionics (ScOSA) is a satellite software project by DLR with the goal of developing a system based on the OBC-NG [218].

SPA Space Plug-and-Play Avionics (SPA) is a framework developed by the Air Force Research Laboratory that provides a standardized interface to interconnect satellite components on a software and hardware level [121].

VIDANA Visionary Data Management System for Nano-Satellites (VIDANA) is a task migration system for RODOS enabling state transfer and multiple hot-standby copies of tasks developed at the University of Würzburg [152, 229].

Symbols

Radiation Environment and Testing

$\sigma_{\text{bit-flip}}$	bit error cross section
ρ	volumetric mass density
d	shield depth
ϕ	particle fluence
Φ	particle fluence rate or particle flux
L	linear energy transfer
$N_{\text{bit-flip}}$	number of observed bit-flips
S_{memory}	size of monitored memory
D_{TID}	total ionizing dose

System Selection and Design

w_i	weight of selection criterion i

Time Synchronization

$b_{\text{CAN bus}}$	data rate on the CAN bus
$C_i(t)$	output of clock i at time t
c_i	correction term of clock i
$c_i(k)$	correction term of clock i in synchronization interval k
$d_i(t)$	drift of clock i at time t
$e_i(k)$	clock error at of clock i at the beginning of synchronization interval k
k	identifier of synchronization interval
K_i	integral gain koefficient
K_p	proportional gain koefficient
k_t	identifier of time synchronization interval at time t
$N_{\text{CAN frames}}$	number of CAN frames of a 1300 B RODOS message
o_i	offset of clock i
$S_{\text{CAN frame}}$	size of an extended CAN frame without stuff bits
s_i	skew of clock i
$S_{\text{stuff bits}}$	size of stuff bits in a CAN message
t	reference time
$T_{\text{CAN frame}}$	transmission time of an extended CAN frame excluding stuff bits
$T_{\text{CAN frame+}}$	transmission time of an extended CAN frame including stuff bits
t_k	time of reception of time update message k
$T_{\text{1300 B RODOS message}}$	transmission time of 1300 B RODOS message via CAN excluding stuff bits
$T_{\text{1300 B RODOS message+}}$	transmission time of 1300 B RODOS message via CAN including stuff bits
$t_{\text{rx},k}$	received reference time in time update message k after delay compensation
$T_{\text{stuff bits}}$	transmission time of stuff bits in a CAN message
$u_i(k)$	skew correction term of clock i in synchronization interval k

Bully Algorithm

I_{all}	set of used identifiers
i_l	unique identifier of currently elected leader
I_{live}	set of identifiers of live nodes
i_n	unique identifier of node n
$T_{\text{await coordinator}}$	timeout to wait before detecting a missing coordinator message
T_{election}	timeout after sending an election message
T_{fail}	timeout after which the election failed
T_{ok}	timeout after sending an ok message
T_{p}	processing time
T_{query}	timeout after sending a query message
T_{resp}	upper bound of response time
$T_{\text{send ok}}$	timeout before sending ok message
T_{trx}	bully message transmission time

Chapter 1

Introduction

1.1 Motivation

CubeSats are small satellites that were first proposed and standardized by Jordi Puig-Suari and Bob Twiggs in 1999 [213]. The *CubeSat Design Specification* [213] defines CubeSats in terms of basic units (U). A 1 U CubeSat has a size of approximately 10 cm × 10 cm × 10 cm and a mass of up to 2 kg [213]. This standard also defines larger CubeSats of up to 12 U by stacking multiple basic units. While CubeSats initially targeted educational purposes [219], they have evolved to a wide field of users and applications including remote sensing, technology demonstration, and science [226]. With the variety of applications, the number of active CubeSats is increasing steadily [208, 226]. This trend is still continuing and more CubeSats than ever will be launched in the upcoming years [101].

The Munich Orbital Verification Experiment (MOVE) program at the Institute of Astronautics (LRT) at Technical University of Munich (TUM) is an educational program to build, launch, and operate CubeSats with students [52, 107]. The 1 U CubeSats first-MOVE, MOVE-II, and MOVE-IIb were launched as part of this program in 2013 [104], 2018, and 2019 [183] respectively. Recently, design and development of the MOVE program's next satellite MOVE-III has begun. At the same time, the DFG cluster of excellence ORIGINS (ORIGINS) Laboratory for Rapid Space Missions (LRSM) initiated the concurrent development of multiple scientific CubeSat missions. The combination of those missions drives the need for a simple, reusable and adaptable platform to bundle efforts and reduce the workload to implement several CubeSat missions concurrently.

1.1.1 Future MOVE Missions

MOVE-III, the next CubeSat of the MOVE project will host the DEbris Density Retrieval & Analysis (DEDRA) payload [159, 160], which counts dust particles to improve space debris models [160]. MOVE-III will be a 6 U CubeSat with three copies of the DEDRA sensor [160]. The details of this mission are still undefined, but initial development and pre-studies have started. The student team uses precursor missions on stratospheric balloons to verify their initial development efforts [112].

This approach reflects some lessons learned of previous missions. From first-MOVE, we know that requirements, especially software requirements, are imprecise and incomplete in early phases of the development [106]. This was noticed similarly during the development of MOVE-II where major parts of the software were only designed and implemented once they were actually required for integrated tests. Langer [105] found that early integration and integration testing as early as possible is critical for a successful mission.

1.1.2 ORIGINS LRSM Missions

The ORIGINS LRSM [62] will launch multiple concurrent CubeSat missions with short timelines in the upcoming years. Currently, two missions are part of the LRSM and scheduled for launch in the upcoming years: The Antiproton Flux in Space (AFIS) mission measuring the antiproton flux in the Van Allen belt, specifically within the South Atlantic Anomaly (SAA) [170], and the Compton Polarimeter

(ComPol) mission measuring the polarization of X-ray originating from Cygnus X-1 [137]. Each of these missions has a set of unique requirements, e.g., AFIS requires a lot of electrical power and reliable operation within the SAA [170]. ComPol, on the other hand, requires precise pointing to Cygnus X-1 [62]. A precursor on the International Space Station (ISS) will demonstrate the technology and verify the scientific instruments in-orbit. Even though the requirements differ, these missions will share major parts of the software and hardware to reduce the workload of the relatively small LRSM team.

1.2 Problem Statement

For future MOVE missions, we see the need for a system that can be used for MOVE-III and the stratospheric balloons. These early balloon missions already drive the requirements for usable end-to-end solutions and provide a platform for early testing of larger parts of the MOVE-III satellite. While the flexibility required for reuse between balloon and satellite missions requires some effort, it also enables late adaptations to changed requirements of the satellite. While this flexibility is important, the changing student team also requires a system that is easy to understand and use after only a short learning phase. For those reasons we see a strong requirement for a system that consists of smaller and independent building blocks and can be assembled to a larger system for real missions. At the same time, the smaller parts enable the independent testing on precursor missions.

Similar to MOVE-III, the LRSM requires a flexible framework to enable a simplified reuse and adaptation to different scenarios. A framework that is easy to understand further reduces the mental load of the LRSM team members and thus enables successful missions with short timelines. At the same time, an increased adaptability to different scenarios, including the increased reliability required for successful operation within the SAA, enables the reuse of the suggested framework between different LRSM and the MOVE-III missions.

The currently available MOVE-II platform is not flexible enough and cannot be adapted easily to these upcoming missions. Development of an independent system for each mission is not an option due to the limited manpower. Therefore, we see the need for a flexible, adaptable, and easy to understand framework for future CubeSats at TUM.

We propose a distributed system based on a network of nodes with an adequate software support as a solution to this problem. We suppose that:

Hypothesis 1 *A system based on a network of nodes can improve the flexibility and adaptability of the overall system and allows simple extensions to meet different mission criteria.*

Specifically, we suppose that this distributed system enables sharing of components between different missions and even between CubeSats, stratospheric balloons, and other precursor missions. A distributed system enables the independent development and simple adaptation to the various scenarios and, with proper interface design, simplifies development and reduces the potential for design and implementation faults due to mismatched interfaces.

In the context of a real-time sensitive system, it is not only important to provide flexibility, but also to provide a suitable level of timing of actions distributed over several nodes. This is especially important for control systems, e.g., an attitude determination and control system (ADCS). We suppose that the proposed system can meet those requirements, specifically that:

Hypothesis 2 *Such a system can meet the timing requirements to implement timing sensitive systems such as a distributed attitude determination and control system.*

Overall, we suppose that such a distributed system can meet all the requirements for the development of MOVE-III, the LRSM missions, and all precursors thereof on stratospheric balloons and the ISS.

1.3 State of the Art

This chapter presents an overview over CubeSat projects at the LRT, available commercial off-the-shelf (COTS) on-board computers (OBCs), and currently available frameworks for on-board software development suitable for CubeSats. Finally, it presents some ideas originating from on-board computing within the Space Shuttle and so-called data field systems.

1.3.1 Satellites at the Institute of Astronautics

To this day, three CubeSats were built and launched at the LRT as part of the MOVE project: first-MOVE, MOVE-II, and MOVE-IIb.

First-MOVE

Development of the first satellite of the LRT called first-MOVE began in 2006 [52]. First-MOVE was a 1 U CubeSat [52] according to the previous version of the *CubeSat Design Specification* [212]. First-MOVE was a technology demonstrator for the in-house developed hardware with a focus on education of students on satellite development and operation [52]. Its scientific objective was the characterization of solar cells in cooperation with an industrial partner [52]. As secondary payload, a camera on-board first-MOVE should gather pictures of earth [52]. Figure 1.1 depicts first-MOVE in its deployed configuration in the laboratory of LRT.

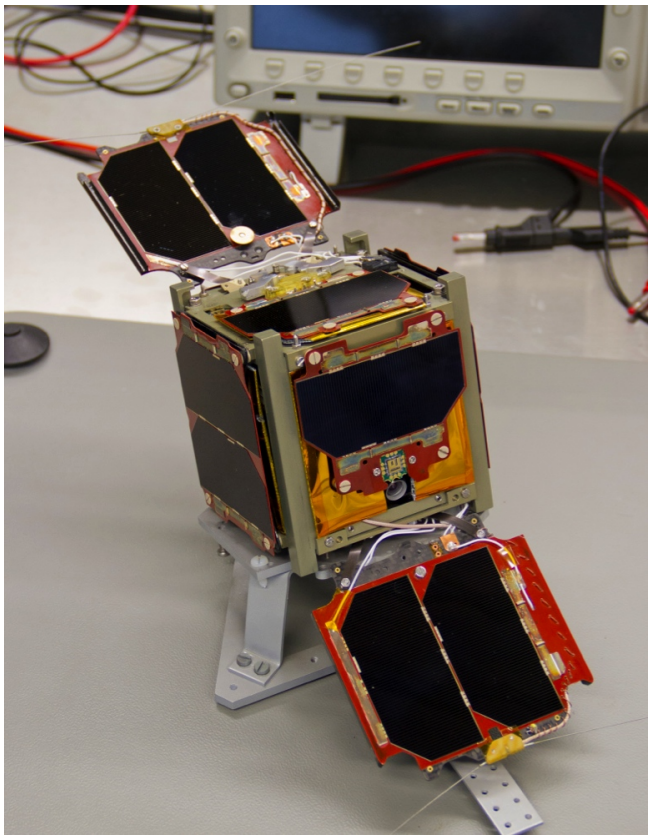


Figure 1.1: First-MOVE in its deployed configuration at the LRT laboratory. Image source: [106]

Most subsystems of first-MOVE were built in-house at the LRT [52]. The only exceptions are a ClydeSpace EPS and battery and an Innovative Solutions in Space (ISIS) UHF/VHF transceiver [52]. The on-board data handling (OBDH) system's hard- and software was also developed in-house, mostly by a single person [106]. An Atmel SAM9260 [49] containing an ARM9 core running at up to 180 MHz was the OBDH system's core [52]. A magnetoresistive random access memory (MRAM) stored the boot image to reduce the likelihood of bit-errors [52]. Additional flash memory extended this storage and contained redundant copies of the boot image for error correction and software updates [52].

A compare and update unit could repair bit flips in the boot image [52]. To mitigate temporary failures a circuit breaker could perform a hard reset of the OBDH [52]. It could be triggered by a latch-up detection, a hardware watchdog or the hard command unit connected to the transceivers [52]. Redundant implementation of the reset mechanisms using radiation hardened components aimed to reduce the risk of a malfunction in this system critical part [52]. No detailed documentation on how all these mechanisms should work is available, as no documentation was created due to missing time and manpower [106]. Figure 1.2 depicts the printed circuit board (PCB) stack of first-MOVE containing all these components.

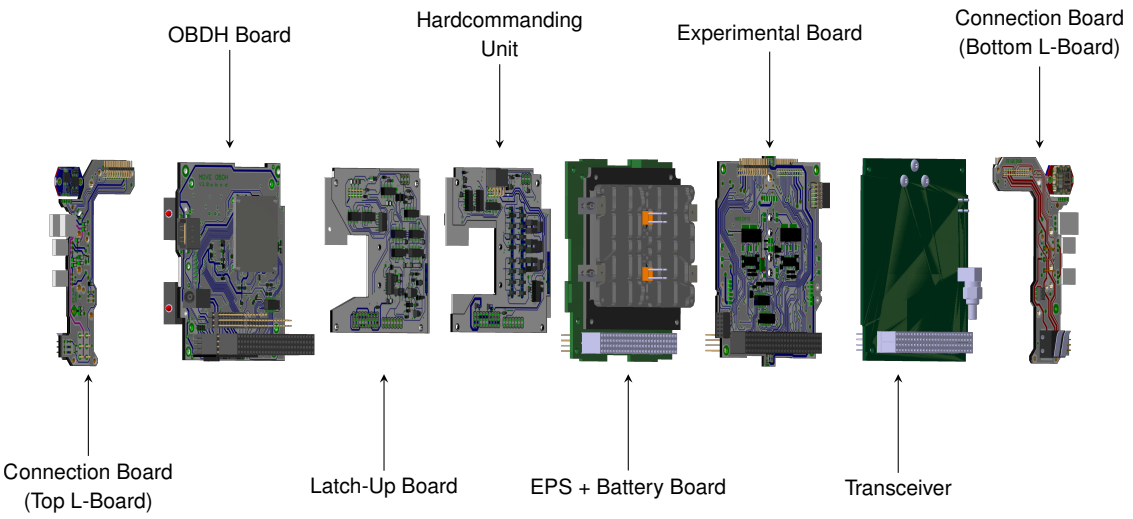


Figure 1.2: PCB stack of first-MOVE. Image source: [104]

First-MOVE was launched in November 2013 and operational in orbit for approximately four weeks [104]. At this point, the operators observed a major anomaly that degraded the satellite to a state where two-way communication was no longer possible [104]. Langer et al. assume that a failure of the OBDH is most likely the reason for this degradation [104].

MOVE-II

Development of MOVE-II, the second CubeSat at the LRT began in 2015 [107]. MOVE-II is a 1 U CubeSat with deployable solar cells and antennas [107]. 4-junction solar cells are the payload of MOVE-II. The scientific goal of MOVE-II is the observation of performance and degradation over time of this novel type of solar cell [186].

Figure 1.3 depicts the stack with the main subsystems of MOVE-II according to the *MOVE-II System Documentation* [108]. The top panel combines the payload solar cells, a sun sensor, and a magnetic coil of the ADCS integrated into the PCB. It connects to the main PC/104 stack via an adapter board. The command and data handling (CDH) board, which also contains the global navigation satellite system (GNSS) receiver [183], is the first board in this stack, followed by the battery and the electrical power system (EPS) PCB. The ADCS main panel is the core of the ADCS. The UHF/VHF transceiver [182] is the always active telemetry and telecommand (TM/TC) transceiver of MOVE-II [107]. The S band transceiver [182] is available for high data rate communication [107] and only enabled on demand.

Not shown in figure 1.3 are the UHF/VHF and S band antennas mounted below the PC/104 stack and the four side panels, one on each side of the stack. Each side panel contains ADCS sensors and actuators, two smaller 4 cm × 4 cm solar cells, and a deployable flap panel with two 4 cm × 8 cm solar cells [107].

Command and Data Handling (CDH) According to [183], a SAMA5D2 [141] processor running at 400 MHz is the core of the CDH system. Furthermore, 512 MB of random access memory (RAM) and 512 MB of NAND flash storage provide the main memory [183]. Additionally, 32 kB of FRAM memory provide storage for critical values and a space vault with 1.5 GB of flash storage and two SD cards

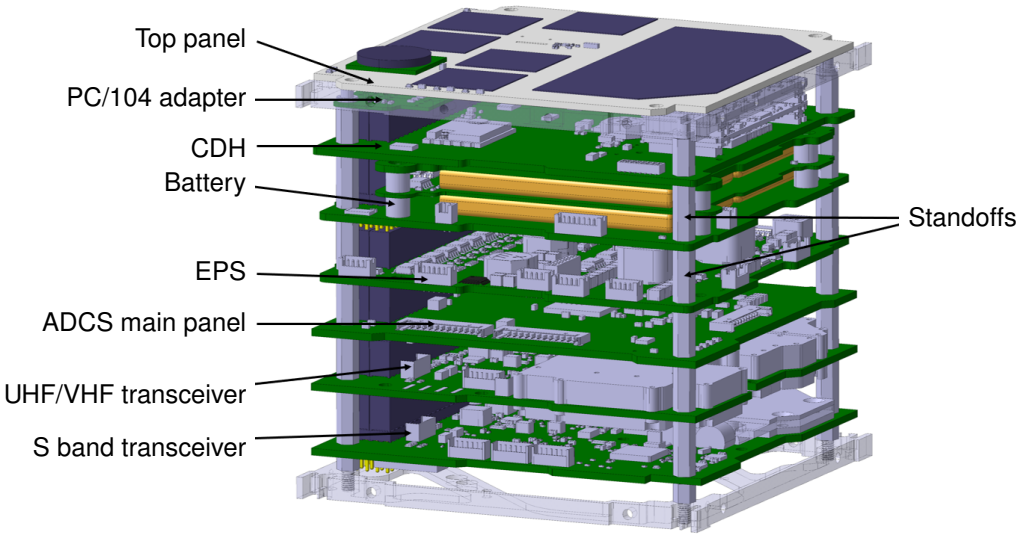


Figure 1.3: PC/104 stack of MOVE-II. Image source: [108].

provides additional long-time storage for less critical data [183]. An external controller provides a watchdog capable of resetting the entire CDH [183].

The authors of [183] also present the boot process of the MOVE-II CDH: During startup the bootloader randomly selects one of three Linux kernels. If booting a certain kernel fails, a watchdog automatically resets the system. The bootloader repeats this process until it finds a working kernel. Afterward, the startup code selects one of three system images based on a round-robin scheme. This selection only uses images previously reaching an uptime of more than 10 min.

A MOVE-II system software image consists of a collection of daemons [114]. Each daemon provides a special service, in most cases the functionality of one subsystem [114]. Systemd[1] is used to manage these daemons, D-Bus[2] is used for communication between different daemons where needed [183]. A special daemon called *HORST* monitors the health of all other subsystems and triggers automated state changes based on this information [183]. Figure 1.4 provides an overview of these daemons.

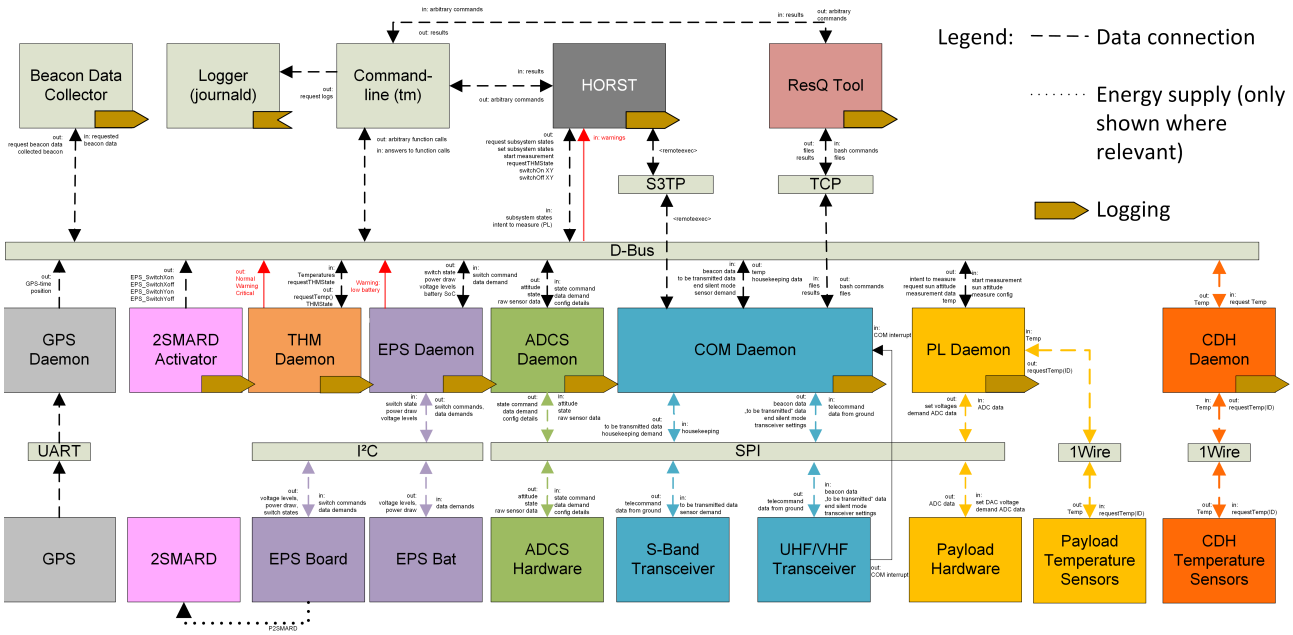


Figure 1.4: MOVE-II software architecture. Image source: own publication [183].

[1] A system and service manager for Linux available at https://systemd.io/.

[2] A message bus system for inter-process communication. Additional information available online at https://www.freedesktop.org/wiki/Software/dbus/.

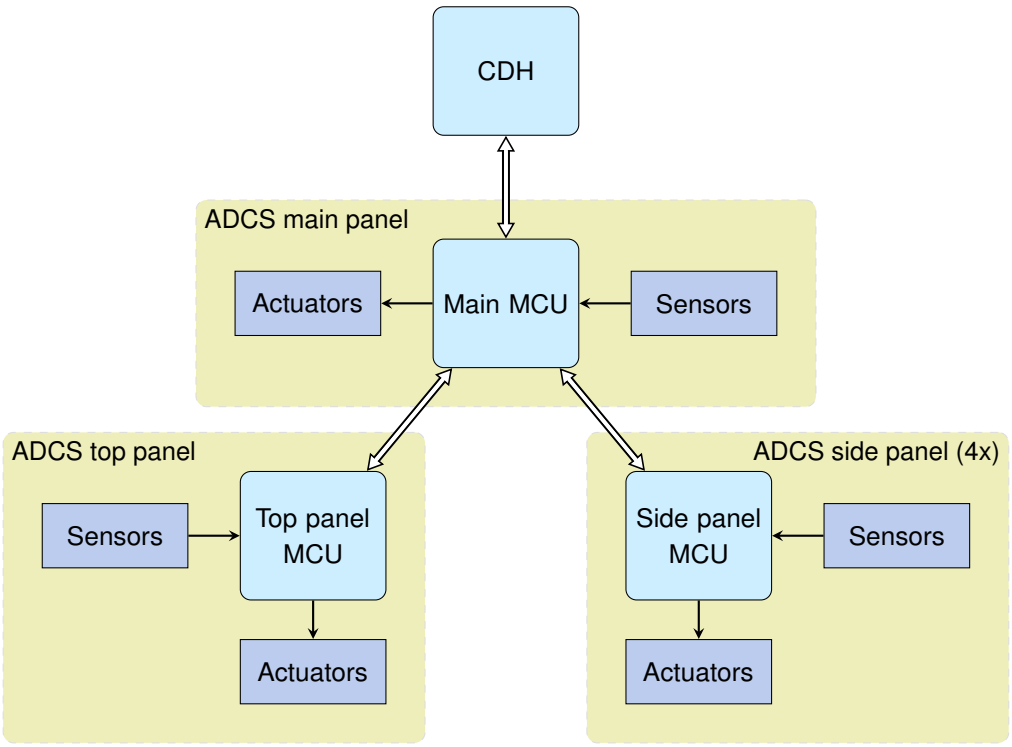


Figure 1.5: MOVE-II ADCS architecture according to [139].

According to Lill, Messmann, and Langer [114], the MOVE-II software was developed using agile methods. The authors state that individual components were developed independently based on an initial system decomposition. According to them, early system level integration and testing assured functionality beginning with the minimum viable product. This way problems with hardware and software could be detected early and were fixed without budget or schedule issues [114].

Attitude Determination and Control System (ADCS) Messmann et al. [139] present the features of the magnetorquer based MOVE-II ADCS: Within this system, a microcontroller on the ADCS main panel is responsible for the control algorithm. Additional microcontrollers on each side panel and the top panel provide the interface to the magnetic field and sun sensors, the gyroscope, and the magnetic coils including the current-sensor feedback. Software components called producers read the sensor values from the hardware and provide it to the main microcontroller. The main microcontroller estimates the current attitude of the satellite based on this input, orbital elements from a two line element and the current time. Based on this input, the control algorithm generates an output based on the estimated attitude and forwards this information to software components called consumers. These consumers in turn control the individual magnetorquers on the different ADCS panels.

Figure 1.5 depicts a summary of the ADCS architecture as described in [139]. According to the authors of [139], the system has a tree like organization where the main microcontroller is responsible for all calculations. All other controllers act as simple input/output nodes for various sensor and actuator signals. Hardware-in-the-loop and software-in-the-loop verification of the flight software did show good results for the overall ADCS system [93].

Launch and Operation On December 3, 2018, MOVE-II was launched and first results were presented in [183]: Initially, regular operation of MOVE-II was delayed due to issues with the ground station. In this phase, MOVE-II had a slowly increasing angular velocity of up to $527\,^{\circ}\,s^{-1}$. Due to the flexible Linux software and the possibility to upload new scripts, the operators could establish an actuation pattern that effectively decreased the angular velocity and stabilized MOVE-II at a residual spin rate of less than $10\,^{\circ}\,s^{-1}$. According to [183], the root cause of the fast spinning is a current loop of the solar panel wiring. Additional issues with the power budget increase the difficulty of the MOVE-II

operation [183].

A twin satellite called MOVE-IIb was launched on July 5, 2019 [93]. Compared to MOVE-II, MOVE-IIb has an adapted solar panel wiring and updated software [183]. Due to a reduced output signal power of MOVE-IIb no regular operation could be performed so far.

1.3.2 Commercially Available CubeSat On-Board Computers

Several integrated COTS OBCs solutions for CubeSats are available today. Table 1.1 lists an overview of these solutions, table A.1 presents a more extensive list in the appendix. Most of them rely on a 32 bit ARM Cortex M core in combination with up to 1 MB of RAM [1, 16, 17, 61, 156, 204]. The major exceptions are the low-power CubeSat Kit modules by Pumpkin Space Systems. These modules use 8 bit or 16 bit microcontrollers combined with less than 100 kB of RAM [171–177]. Powerful multiprocessor system on a chip (MPSoC) with multiple ARM cores combined with field programmable gate array (FPGA) gates available for mission developers are used for computational intense applications [1, 79, 98, 162, 203]. Up to 64 GB RAM is available on these powerful platforms [162]. Only a few platforms use cores designed specifically for space missions, e.g., the LEON3-FT [2, 3, 79].

Table 1.1: Summary of COTS OBCs for CubeSats.

Core \ Memory	FRAM/MRAM	Flash	Both	Unknown
8 bit MCU	-	-	-	PPM-B1 [176]
16 bit PIC	-	PPM-D1 [171] PPM-D2 [175] PPM-E1 [177]	-	-
16 bit MSP430	Eddie The Computer [205]	-	-	PPM-A1–PPM-A3 [172–174]
32 bit ARM M4	-	-	n-ART OB-COMMS [16]	n-ART SMC [17]
32 bit ARM M7	-	EnduroSat OBC [61]	3C2 [156] OBC-P3 [204]	-
other 32 bit ARM	-	-	iOBC [80]	-
other 32 bit	-	-	NanoMind A3200 [91]	-
32 bit LEON3-FT	-	-	-	SIRIUS OBC/TCM [2, 3]
FPGA or similar	KRYTEN-M3 [1]	CSP [203] Proton 200k Lite [202]	CFC-400 [79] Antelope OBC [98]	TELOS OBC [162]

In most cases, memory for program code is part of the microcontroller. A few MB of additional external code memory exist on some OBC modules. Either ferroelectric random access memory (FRAM) or MRAM provides high reliability storage on some platforms. Flash memory provides high density storage for less critical data. On most COTS OBCs SD-cards provide additional mass memory.

Most COTS OBCs do not provide any kind of hardware redundancy and rely on a watchdog, which is sometimes combined with a second boot image stored in the same memory. In a few cases, duplicate SD-card slots exist, which can provide redundancy regarding mass memory storage. The Space Inventor OBC-P3 is an exception and provides fully redundant copies of all hardware components [204].

1.3.3 On-Board Software Frameworks

A number of software frameworks for on-board software developments are available today. This section gives a summary of the known frameworks in alphabetical order.

CAST Reference Architecture

This paragraph is a summary of the China Academy of Space Technology (CAST) reference architecture as presented in *CAST Flight Software as a CCSDS Onboard Reference Architecture* [36]: The CAST reference architecture aims to standardize the on-board software and networking, increase reuse of software, and replace manual programming of on-board software by assembling a software based on an architecture and smaller software components. The CAST flight software implements several Consultative Committee for Space Data Systems (CCSDS) standards, especially for information transfer on lower layers. The CAST reference architecture consists of four layers:

- A hardware layer containing the physical hardware, e.g., the central processing unit (CPU), memory, or external interfaces;
- an operating system (OS) layer providing a real-time kernel, hardware abstraction and a uniform application programming interface (API);
- a middleware layer providing networking, data transfer, and commonly used basic support services;
- and an application layer containing the various management services of the system.

The authors of [36] emphasize the improvements in standardization, flexibility, and scalability due to the simplified reuse and replacement of individual components. They also state that using the CAST reference architecture could improve the reliability by task migration, system reconfiguration, and the reduced complexity of system verification due to reuse of components.

Source code or extended documentation of the CAST reference architecture is not publicly available [143] and was not available to the author of this thesis. A publication as a CCSDS orange book (*CAST Flight Software as a CCSDS Onboard Reference Architecture* [36]) is pending [231].

NASA cFS

National Aeronautics and Space Administration (NASA) core Flight System (cFS) is an open source[3] on-board software framework providing abstractions and applications reusable between missions [133]. The cFS consists of three layers:

- A platform abstraction layer providing an abstraction of the used OS and a platform support package containing the software required to combine the cFS core, the OS, and the hardware [132]. Linux, Real-Time Executive for Multiprocessor Systems (RTEMS), and VxWorks are supported OSs with platform support packages available for Linux based development, ColdFire, RAD750, LEON3 and the GomSpace NanoMind CubeSat OBC [133].
- An executive services layer containing the core and an API to access these service [133]. The core provides essential services for messaging, alerts, runtime configuration, and startup/managing of other services [133].
- And an application layer consisting of threaded applications and shared libraries [133]. A number of applications for general TM/TC tasks are available [132].

An active community provides additional utilities and applications [133]. cFS is successfully used on the Dellingr CubeSat, even though porting it to the CubeSat hardware was a challenge [50].

Although cFS provides an inter-task message routing mechanism, no use in distributed systems is mentioned.

[3] Available at https://github.com/nasa/cFS.

COrDeT On-Board Software

Three layers assemble the Component Oriented Development Techniques (COrDeT) on-board software reference architecture: A component layer, an interaction layer and an execution platform [181]. The component layer contains the functional and sequential behavior of the software system where each component implements the software for a single concern [181]. The interaction layer consists of containers and connectors: Containers surround these components and take care of concurrency, real-time, and reconfiguration tasks; connectors implement the communication between different containers [6]. The execution platform provides low level abstractions, monitoring and control services, domain-neutral services such as tasking support, and on-board interface services [181]. This decoupling on on-board interface services with strict functional and timing requirements from other components allowed independent development of the various parts of the overall system [6]. The COrDeT development process uses various modeling tools to declare and define the on-board software [181].

The C2 implementation of COrDeT by P&P Software GmbH is publicly available via the vendor website[4] or a public GitHub repository[5]. This implementation does not provide a communication middleware for any specific hardware [163]. Instead, it assumes that a message-passing middleware with certain properties exists [163]. Neither user manual [163] nor framework definition [164] mention the used real-time OS or any other hardware specific issues. Therefore, the author assumes that these have to be provided by the user of the framework.

Although [181] mentions the physical distribution across nodes, no further analysis of the use of COrDeT in a distributed system could be found.

CubedOS

CubedOS is a CubeSat software framework entirely written in SPARK [25]. As Brandon et al. [25] states, the advantage is the static verification of the software. Yet he also mentions the possibility to use bindings for regular Ada or C code with the intent to use external libraries. CubedOS provides direct messaging as well as publish-subscribe message-passing among different modules [25]. It can be executed on any system with Ada support such as Linux or VxWorks [25]. Current effort to increase the feature set of CubedOS is ongoing and distributed processing is one of the next steps [26]. CubedOS is publicly available[6] with unknown license.

F Prime (F')

F Prime (F') is an open-source[7] framework for small spacecraft developed by NASA's Jet Propulsion Laboratory [22]. The following paragraph is a summary of F' according to Bocchino et al. [22]: According to the authors, F' aims to simplify reuse of flight software over multiple missions and speed up development and testing. The main features of F' are its modular architecture, a complexity level matching the small satellites, a development ecosystem, and the wide range of supported processors. Splitting the architecture into three main parts increases the reusability. *Components* are the first concept and are similar to classes in object-oriented languages, which define data and methods to manipulate said data. Ports define interfaces of components to interact with one another and are strictly typed. Topologies define an assembly of components interconnected via the respective ports. A set of support tools provide the ability to model a system and generate large parts of the required code based on this model; only the internal functionality of a component has to be implemented manually. The current open-source version[7] provides a Linux and macOS port.

The F' community on GitHub[8] provides additional ports for FreeRTOS, VxWorks, Arduino, and some others. Additional documentation of F' is available via the project's website[9].

[4] https://www.pnp-software.com/cordetfw/download.html
[5] https://github.com/pnp-software/cordetfw
[6] Available at https://github.com/cubesatlab/cubedos.
[7] Available at https://github.com/nasa/fprime.
[8] https://github.com/fprime-community
[9] https://nasa.github.io/fprime

Although most systems using F' are monolithic systems, the F' User's Guide[10] suggests a hub pattern to distribute an F' application over several nodes [31]. While this enables distributed systems, it is not directly supported. Especially the necessity to define individual ports for every message traversing a shared medium as suggested in [31] increases the required effort if more components want to communicate with one another on this medium.

GEneRIC Onboard Software (GERICOS)

The GEneRIC Onboard Software (GERICOS) framework as presented by Plasson et al. [169] aims at increasing the development speed of payload software. They present three main parts of GERICOS [169]:

- the core acting as an abstraction of the used real-time OS,
- the blocks providing generic solutions for software components required in most payload instruments,
- and the drivers acting as generic interface to the periphery not yet covered by the used OS.

GERICOS is implemented in the C++ subset as defined in *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program* [33] and supports the ThreadX, RTEMS, and FreeRTOS operating systems [169]. A Unified Modeling Language (UML) based documentation fully documents the design of GERICOS [169]. Plasson et al. [169] mention an Extensible Markup Language (XML) dialect that can be used to automatically generate some parts of the code. Although [169] mentions this feature, the authors also mention that it is not fully implemented. The current state of these features is unknown.

Although GERICOS was initially developed for a LEON based platform of the Radio Plasma Waves onboard the European Space Agency (ESA) solar orbiter [169], it has been successfully ported to CubeSat hardware of the PicSat mission [110]. Plasson et al. [169] list porting the system to a LEON3-FT multicore system as one of the challenges for future developments of GERICOS. Therefore, the author of this thesis assumes that it is not suitable for distributed systems. Currently, GERICOS is not available as open-source software and no technical documentation is publicly available.

KubOS

KubOS is an open-source[11] on-board software framework developed by the Kubos Corporation[12]. According to KubOS online documentation [100], it is a collection of microservices running on a custom Linux distribution. The documentation furthermore states that Internet Protocol (IP) is used as baseline for all internal communication. Although Linux is not a real-time OS, the KubOS documentation states that it is still sufficient for most applications and response times as low as 30 μs are possible.

The requirement for the custom Linux distribution restricts the usage of KubOS on some platforms. Especially for 16 bit micro controller units (MCUs) or MCUs without memory management unit (MMU) the Linux support is very limited [234].

Nanosat Mission Operations Framework (NMF)

This paragraph is a summary of Nanosat Mission Operations Framework (NMF) as introduced by Coelho, Koudelka, and Merri [46]. NMF aims at increasing the portability of on-board software and embedding it in a larger framework that also includes ground station software components. In contrast to cFS, NMF does not target embedded systems but systems with suitable features to run a regular OS. The framework is split into four layers:

- a transport layer to transfer messages of higher layers between different nodes,

[10] The F' User's Guide is available online at https://nasa.github.io/fprime/UsersGuide/guide.html.
[11] Available at https://github.com/kubos/kubos.
[12] https://www.kubos.com/

- a message abstraction layer connecting the transport layer to a specific NMF implementation and ensuring communication between components on different implementations of NMF,
- a service layer providing a number of CCSDS services, platform management services, and software management services,
- and an application layer containing the mission specific logic implemented in individual applications.

Every application implements a certain functionality and can be individually installed, executed, and updated [45]. Dependencies between different applications are possible, thus a specific service can be split up into a chain of smaller individual services [45]. The software management and update components provide a way to update individual applications, the entire OS, or even the FPGA configuration [47]. ESA's OPS-SAT mission makes use of these management features [47].

An open-source Java implementation as presented in [44] is publicly available[13]. Coelho, Koudelka, and Merri [45] suggest the use of NMF in a distributed application spread over multiple satellites in formation flying scenarios. Using NMF for embedded distributed systems is not feasible due to the requirement for a rather powerful controller capable of running a regular OS [46].

On Board Computer — Next Generation (OBC-NG) and Scalable On-Board Computing for Space Avionics (ScOSA)

Lüdtke et al. [118] present a distributed approach for the On Board Computer — Next Generation (OBC-NG). According to them the main goals of OBC-NG are the increased performance, high reliability due to autonomous reconfiguration, and commanded in-orbit reconfiguration. An example setup of an OBC-NG system can be seen in figure 1.6. Each node of the OBC-NG consists of an ARM based processing unit [20] with an optional high performance coprocessor on processing nodes or periphery on input nodes [118]. A SpaceWire network provides the communication capability within the OBC-NG and is used to synchronize the local time on all nodes [20]. Linux and Realtime On-board Dependable Operating System (RODOS) are supported OSs of the OBC-NG [118]. A tasking framework on top of the OS enables the cooperation of different modules and provides the necessary software infrastructure [167]. The OBC-NG middleware provides a reconfiguration and checkpoint service capable of redistributing tasks on a node failure or as requested via a ground command [118]. An evaluation of the recovery mechanism on a network of Zynq based nodes confirmed the redistribution time of less than 5 s for non-real-time tasks and less than 100 ms for real-time tasks [167].

The tasking framework of OBC-NG was used on the Eu:CROPIS satellite of the German Aerospace Center (DLR) [126]

The OBC-NG development continues within Scalable On-Board Computing for Space Avionics (ScOSA) [218]. Treudler et al. [218] present that the on-board computing system within ScOSA is split into redundant high-reliability nodes based on a LEON3 system on a chip (SoC) and COTS Xilinx Zynq based computing nodes. According to them, SpaceWire interconnections provide the reliable communication within the system and each node uses the Linux or RTEMS OS. They demonstrated the usability of ScOSA for earth observation, attitude control, robotic servicing, and autonomous navigation tasks. The ScOSA flight experiment project aims to demonstrate the ScOSA system within the DLR CompactSat mission [119].

Although the OBC-NG and ScOSA systems provide a platform for distributed on-board computing, it is in its current state not suitable for CubeSats due to the power consumption of the suggested hardware. The source code of the software is not publicly available.

Space Plug-and-Play Avionics (SPA)

Lyke et al. [124] present Space Plug-and-Play Avionics (SPA) as a framework that provides an interface standard to achieve a behavior similar to common Universal Serial Bus (USB) devices; using a device should be possible without the need of additional drivers. For this purpose, each device can identify

[13] Available at https://github.com/esa/nanosat-mo-framework.

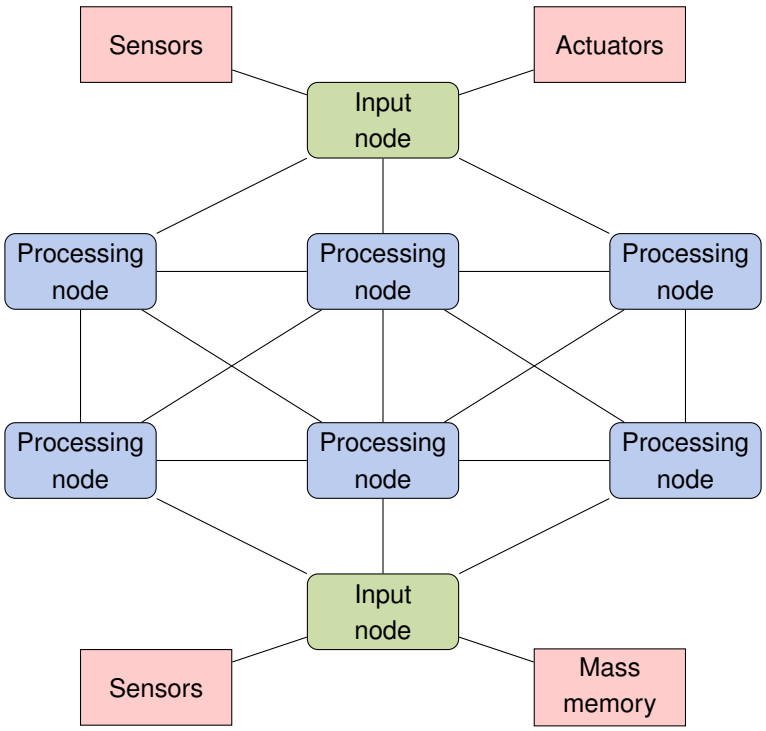


Figure 1.6: Example setup of an OBC-NG system according to [118].

itself and provide the interface definition with an extended Transducer Data Sheet (xTEDS) [109]. The communication within the first generation of SPA was implemented using a modified USB that provides sufficient power for space hardware [123]. Later on, support for SpaceWire [122], inter integrated circuit (I^2C) [225], and optical communication [67] were added. Sensor interface modules simplify development of SPA enabled devices, as they can act as interface adapter between the SPA network and individual sensor modules [121]. SPA was used on various satellites [28, 68, 102, 103]. A CubeSat compatible variant only using I^2C and USB interfaces is also available [92, 134]. SPA did show its capability in rapidly assembling a satellite system from individual components [39].

Although SPA is usable on CubeSats [92, 134] and provides a universal interface to connect components and autonomously configure these components in a distributed manner [121], SPA is mainly used as a network of sensors connected to a central OBC. No SPA implementation is available to the author of this thesis, although [40] presents an implementation of SPA.

Flight Software Framework (fsfw)

In [18], Bätz introduces the open-source[14] Flight Software Framework (fsfw) developed at the University of Stuttgart. This paragraph presents fsfw according to this description by Bätz: fsfw is a component based framework, where each of these component has well-defined interface accessible by other components. The core acts as a connector for different components and as a high-level interface to hardware and OS functionalities. A separate platform abstraction layer provides an interface to the hardware, which is also used by the core. Initially, development of fsfw was based on the LEON3-FT and the RTEMS real-time OS [18]. Additional support for Linux, FreeRTOS, and a generic hosted abstraction was added later on [154].

Although the fsfw provides some features required for a distributed system, [18] explicitly mentions that the distribution of periodic data, simplified support to call the interface of other modules, and the support for distributed computing is currently not available.

[14] Available at https://egit.irs.uni-stuttgart.de/fsfw/fsfw.

1.3.4 Space Shuttle Avionics

One of the first high reliability systems implemented partially in software is the Space Shuttle avionics system. According to *Space Shuttle Avionics System* [73], this system consists of four redundant units and an additional payload unit connected to each other and the periphery via communication buses. It also introduces these 28 physical communication buses grouped into seven categories, which interconnect the nodes and connect them to the periphery. According to that documentation, all critical sensors and actuators are redundantly available and interconnected in a way that assures that no single component or communication bus failure is critical for the overall operation. Soft checkpoints synchronize all nodes in redundant operation aided by discrete output signals on each node. [34].

A system of four redundant units can reliably detect up to two concurrent node failures [34]. The comparison of special data words and bus timeouts indicate a node failure in this mode of operation [198]. These tests use redundant buses, where each node controls a separate bus and only listens on all other buses [73]. Dedicated hardware implements the final voting step to detect and disable failed nodes [198]. This hardware automatically removes a node from the redundant set if the input from two other nodes indicates failure of this specific node [198].

This cooperative voting is no longer possible in a degraded system and only two available units. In this case, only limited failure detection and recovery is available [198]. If these nodes disagree, a self-test can sometimes still decide which node failed [198].

The Space Shuttle avionics system provides an outstanding reliability and can recover from many faults; A failing node does not affect nominal system operation. While this level of reliability may be desired, it has a high cost regarding the system's complexity, a requirement for special hardware, and a large amount of physical bus interconnections. The Space Shuttle system cannot be directly transferred to CubeSats as they do not generally require this level of reliability and are constrained regarding available hardware and physical space required for the interconnections.

1.3.5 Data Field Systems

Mori [153] presents a distributed system based on a so-called data field. Data is broadcasted into this data field identified with a content code [153]. Subsystems must be able to manage themselves and operate even without the other subsystems available [153]. Yashiro, Takahashi, and Fujiwara [235] extends this to a system based on COTS OBC modules connected via a redundant Controller Area Network (CAN) bus. Every data point from input nodes or the OBC nodes themselves is broadcasted on the CAN and can be processed by the OBC modules on demand [236]. This network schedules messages based on cycles, which also act as a synchronization point for the redundant copies of the OBC [236]. Majority voting and an output function assure that only a single command for an actuator is published [235]. After a failure nodes perform autonomous recovery, synchronize themselves to the cycles and regain correctness after collecting a suitable amount of data at the CAN bus [235]. According to [153], this approach of a distributed system simplifies maintenance, adaptation, and future extensions of the system. Although bread board level system tests were successful [235, 236], no further development of the system could be found by the author of this thesis.

1.4 Gap Analysis

As previously mentioned in section 1.2, the MOVE-III and LRSM CubeSat missions require a flexible, adaptable, and yet easy to understand platform and software framework. The MOVE-II platform is not flexible enough to provide all the required functionalities and can thus not be used without major modifications. Instead, we propose a solution based on a network of nodes and a software framework to enable simplified use of such a distributed system (see hypothesis 1). This system should be capable of distributed control and enable the development of complex and time critical distributed subsystems, such as an ADCS (see hypothesis 2).

First, a framework for a distributed control in MOVE-III and the LRSM CubeSats should be open-source or otherwise available at TUM, including a usable implementation and suitable documentation. Additionally, such a framework should be suitable for smaller MCUs with limited memory and potentially

Table 1.2: Features of available frameworks.

	CAST	cFS	COrDeT	CubedOS	F'	GERICOS	KubOS	NMF	OBC-NG/ScOSA	SPA	fsfw
open-source	✗	✓	✓[i]	✓[ii]	✓	✗	✓	✓	✗	✗	✓
documentation	✗	✓	✓	✓	✓	✗	✓	✓	✗	✗[iii]	✓
small MCUs	?	✓[iv]	?	✓[v]	✓	✓	✗	✗	✗	✓	✓
modular	?	✓	✓	✓	✓	✓	✓	✓	?	✓	✓
interfaces	?	✓[vi]	✓[vi]	?	✓	?	✓	✓	?	✓	✓[vii]
distributed	✓	✓[viii]	✓[ix]	?[x]	✓[xi]	✗	✓[xii]	✓[xiii]	✓	✓[xiv]	✗
synchronization	?	?	✗	?	✗	?	?	?	✓	✓	✗

[i] Only high level code available, not bound to any OS [ii] License unknown [iii] Although some standards exist [iv] With some limitations [v] VxWorks as base OS available for small MCUs [vi] Generic messages only [vii] No simple interface to access other components [viii] Inter-task message routing is available, but no use in distributed systems documented [ix] No message-passing is implemented [x] Distributed processing is one of the next steps [xi] Rather complicated setup to send a large amount of different messages over a shared network [xii] IP based communication enables distributed applications [xiii] Coelho [44] suggests a distributed use over several satellites, but does not mention distributed applications in a single satellite [xiv] Distributes individual subsystems, but no distributed control

without a dedicated MMU; otherwise the system might exceed the limited resources, especially regarding electrical power, on a CubeSat. This also potentially allows choosing from a wide range of COTS hardware platforms as presented in section 1.3.2. To enable the required flexibility and reusability of components, basic building blocks should be available that can be configured and orchestrated into a specific system. This modularity enables the reuse of components in different missions and reduces the required manpower. A user-friendly interface to these modules furthermore reduces the mental load of developers and potentially shortens the required timeline of a mission. Distributed control algorithms require a well established communication scheme between different software components within a single node and also within different nodes of the distributed network of nodes. Thus, the system must enable this distribution over several nodes within its design. Finally, a synchronization mechanism enables real-time sensitive applications within a distributed system. This mechanism is thus an essential part of such a framework.

Although some frameworks for on-board software exist (see section 1.3.3), none of those provides the required properties for distributed control on CubeSats. Table 1.2 provides an overview over these frameworks regarding their support for the mentioned properties and features. The CAST reference architecture, COrDeT, GERICOS, ScOSA, and SPA are not publicly available and thus cannot be used for the future TUM CubeSats. CFS, CubedOS, KubOS, NMF, and ScOSA do not support small MCUs and have thus a limited usability on CubeSats. Furthermore, only the CAST reference architecture and ScOSA explicitly support distributed applications, but as previously mentioned both are unavailable. Similarly, only ScOSA and SPA explicitly support the synchronization of nodes. Again, these are not publicly available.

Therefore, we identify two major research gaps: First, distributed satellite bus control for resource constrained hardware on CubeSats is not available. Second, no reliable time synchronization between nodes in such a system is available.

Research Gap 1 *Distributed satellite bus control for a resource constrained CubeSat platform.*

As previously mentioned, no framework provides advanced features for distributed systems on constrained hardware, i.e., such hardware without MMU and only limited available RAM and program memory. As such hardware is common in COTS CubeSat OBCs (see section 1.3.2), a framework for distributed control on such platforms is essential to demonstrate the use of a distributed system for

CubeSat bus control applications. Additionally, no demonstration of the increasing flexibility and adaptability of a distributed CubeSat control system is currently available (hypothesis 1).

Research Gap 2 *Reliable time synchronization on a resource constrained platform.*

Besides the general distributed system for CubeSats, a complex control application requires an advanced time synchronization mechanism. Only once all nodes share a common time model, a synchronized execution of time critical applications is possible. Currently, this is not supported in any of the previously mentioned frameworks suitable for CubeSat typical resource constrained hardware.

1.5 Scope of this Thesis

This section introduces the objectives of this thesis based on the previously presented problem statement (section 1.2) and gap analysis (section 1.4). The main goal is the demonstration of a distributed system suitable for CubeSats as suggested in hypothesis 1. As specific missions of the LRSM as well as the MOVE-III satellites should use the proposed system, this includes not only a theoretical demonstration, but instead requires a hardware and software demonstrations that directly enables mission development. Therefore, we split the main goal into three distinct objectives:

Objective 1 (Framework Design and Implementation) *Design and implement a framework for distributed control on CubeSats suitable for small MCUs with or without dedicated MMU.*

Objective 2 (Demonstrate ADCS Capability) *Demonstrate that real-time requirements can be met and thus an ADCS system typical for a CubeSat is feasible based on the proposed framework.*

Objective 3 (Demonstrate on Target Hardware) *Demonstrate that real hardware exists that is suitable for use in space and can be used as a platform for the provided framework.*

Objective 1 is the enabler for distributed systems on CubeSats. As currently no such framework exists and the lack thereof is one of the identified research gaps (see research gap 1), this objective aids in advancing the state of the art regarding CubeSat control systems. Afterward, objective 2 demonstrates the usability for real-time sensitive applications. Although the objective focuses on an ADCS system, it indirectly demonstrates the limitations of the proposed system. Therefore, this objective not only satisfies the needs of ADCS designers, but aids general mission designers and provides insight into system boundaries and the implementation of real-time control loops based on the previously proposed framework. Finally, the demonstration on appropriate target hardware assures that it is really usable for the target environment, where especially AFIS requires safe operation in high radiation environments of the SAA. While hardware evolves over time, the LRSM missions as well as MOVE-III require an available hardware platform to advance their development.

While these objectives include the demonstration of the required framework capabilities to build a real mission, they do not include the implementation of user applications.

1.5.1 Approach

The general focus of this thesis is the overall functionality and usability of the proposed framework for distributed control on a CubeSat. The clear goal is a framework that can be used on LRSM missions and the MOVE-III satellite. Therefore, the suggested solution is not necessarily optimal, but instead aims at a simple and yet flexible solution that can meet the requirements of those missions. This way, we hope that future mission designers and developers can still use the framework, play with its features, and evolve the framework with their additions and improvements.

To achieve this goal, we will first have a closer look at the LRSM missions AFIS and ComPol as well as the MOVE-III mission. As the AFIS mission requires operation in the increased radiation of the SAA, we will also have a look at the radiation environment in low earth orbit (LEO) and specifically the SAA. Based on this information, we will better understand the needs of these missions and derive design guidelines and, if applicable, performance requirements.

The final design of the distributed satellite control framework happens in three steps: An initial analysis of existing frameworks and suitable OSs provides the basic knowledge and foundation of this design. A selection of an existing framework as base of the own design avoids reinventing larger parts and enables the focus on the previously presented objectives of this thesis. Additionally, we select a general network topology and technology for the distributed system. A rough description of the overall framework initiates the second step. The following detailed description includes all parts of the framework's core, the synchronization of nodes, and some additional features providing an extended design space for future mission's designers. The last part of the framework's description provides insight into the implementation. This part includes the reference hardware platform, the implementation of the framework's core, and the suggested usage of the framework. Unit testing and software interface tests provide a reasonable confidence in the implementation and assure proper behavior of the framework's core.

At this point, the description of the framework is complete, and thus objective 1 is fulfilled. The remaining objective 2 and objective 3 are demonstrated in subsequent tests using the suggested reference hardware. The first test focuses on the critical time synchronization of all nodes and the timed execution of tasks. They verify the nominal case and analyze worst-case scenarios. Thus, they provide insight into the limitations of this aspect of the framework. The second test focuses on the radiation tolerance of the selected MCU. It verifies the MCU's radiation tolerance as specified by the vendor and closes a gap in the available data. This test assures that a real platform exists that satisfies the needs of the AFIS mission, specifically the operation within the SAA.

1.6 Structure of this Thesis

The first part of this thesis introduces the general problem and the relevant background information. The current chapter starts with the general problem, followed by an analysis of the related work and the identification of the relevant research gap. Chapter 2 introduces the LRSM and the MOVE-III missions, identifies the limiting factors based on the known profiles of said missions, and finally defines design guidelines for later system design.

The second part presents the design and implementation of the suggested solution. Chapter 3 shows the proposed system starting with a selection of a baseline architecture and framework or OS. Afterward, it introduces the proposed framework in detail. The following chapter 4 highlights the implementation of the framework including its current limitations and presents the prototype hardware.

The third part presents two tests conducted to verify the framework and the used hardware platform. First, chapter 5 demonstrates the time synchronization behavior and the accuracy and precision of timed execution as required for an ADCS system. Afterward, chapter 6 demonstrates that the selected MCU is indeed capable of operating in the harsh space environment.

The last part discusses and summarizes the achievements of this thesis. Chapter 7 discusses the results with a comparison of achievements to defined goals of this thesis, presents the simplifications and the use of the presented work as part of the LRSM missions, and provides insight into current limitations and future extensions. Finally, Chapter 8 concludes this work with a summary and an outlook into future efforts.

Chapter 2

Background and Design Goals

2.1 ORIGINS LRSM Missions

The LRSM is part of the excellence cluster ORIGINS[1] at TUM. Within this interdisciplinary research network, the LRSM supports scientists using CubeSats and small satellites for scientific missions [62].

The ORIGINS LRSM CubeSats and related missions will use the suggested on-board control system. This section introduces the AFIS and ComPol missions and the ISS based technology demonstrator In-Orbit Verification Experiment 1 (IOV-1).

2.1.1 AFIS

The AFIS detector measures the flux of antiprotons trapped in Earth's magnetic field [170]. AFIS focuses on the inner Van Allen belt and specifically the SAA [170]. While previous measurements detected antiprotons with a kinetic energy above 60 MeV [5], AFIS targets the lower energetic antiprotons in the range of 25 MeV to 100 MeV [170].

The active core of AFIS has a volume of about 8 cm × 8 cm × 8 cm and consists of 1024 scintillating fibers aligned as 32 layers of 32 fibers each [117]. The entire sensor including the readout electronics fits into a cube of 10 cm × 10 cm × 10 cm [116] and thus into 1 U of a CubeSat. The sensor, including the readout and processing electronics, consumes about 20 W to 35 W of electrical power [115, 116, 170]. Processing of the raw sensor readout classifies the events and particles based on the deposited energy along the particle's track within the active volume of the sensor [116]. The omnidirectional [115, 117] AFIS sensor has a worst-case angular resolution for detected particles of 6° [116] and average-case angular resolution of approximately 2° [117]. Losekamm et al. [116, 117] estimate the relative energy resolution to about 1 % within the target energy range.

An evaluation of the sensor within the ISS [117] is currently ongoing.

Impact on CubeSat Platform

Although the missions final orbit or concept of operations is not fixed yet, we can derive some basic aspects affecting the CubeSat design. Due to the sensor size of approximately 10 cm × 10 cm × 10 cm, the AFIS CubeSat including the sensor and the required satellite platform will be a 3 U CubeSat. Three main limitations for the design space of this CubeSat exist: the large amount of data that needs processing and downlink capabilities; the power consumption of more than 20 W; and the operation within the radiation intense SAA. Other aspects, e.g., pointing of the CubeSat into a specific direction, are less important. The sensor and subsequent data analysis require only a rough attitude knowledge.

A consequence of a large amount of data is the requirement for a communication system with a high data-rate downlink, which in turn most likely requires directional antennas and ground station pointing during overpasses. Similarly, the high power consumption requires an extended solar array, which must be pointed towards the sun. Therefore, the ground station and sun pointing modes are the

[1] Additional information about ORIGINS and the research efforts is available online at https://www.origins-cluster.de/.

main drivers of any ADCS requirements. Both systems require an accuracy in the magnitude of a few degrees only.

Operation within the SAA requires a certain level of radiation tolerance. Rare unexpected resets, missing some measurements, or the corruption of a few measurements within this region may be tolerable. Nevertheless, the overall operation and specifically the firmware and critical configuration set must not be affected by the operation in the SAA.

The high computational effort required for data processing is another critical aspect for the design of the AFIS CubeSat platform. A separate high-power Payload Data Processor (PDP) replacing the FPGA otherwise included in the AFIS readout electronics is developed within the LRSM for this purpose. This PDP as such is not subject of this thesis.

2.1.2 ComPol

The following description of ComPol is a summary of ports of the Master's thesis of Meier [137]. ComPol's goal is the observation of spectrum and polarization of hard X-ray emission of the black hole binary system Cygnus X-1 for a prolonged time. The ComPol detector itself is based on Compton scattering. A first detector layer scatters the X-ray photons, a second layer absorbs these photons. Both layers detect the position of scattering or absorption of the photon and the photon's energy. Based on these measurements, Compton-scattered photons can be separated from other events. A statistical analysis of many Compton-scattered photons and specifically the scattering direction of these photons provides information about the polarization of those photons. A collimator with a preliminary inner diameter of 15 mm and length of 10 cm limits the opening angle of the instrument and reduces the expected noise. The ComPol detector fits into the volume of about 1 U and the current mission design assumes a 3 U overall CubeSat.

Impact on CubeSat Platform

The observation of the stellar object Cygnus X-1 requires a precise pointing of the satellite. Analysis of the Compton-scattered photons requires a precise knowledge of the CubeSat's pointing, as the incoming angle of photons is one of the basic parameters. Due to the narrow opening angle, the instrument requires an accurate pointing for any measurement. Additionally, the statistical analysis requires a long-time stable and precise pointing to avoid additional measurement uncertainties. Currently, we estimate the requirement for the ADCS accuracy and precision in the range of 0.1°. To meet this requirement in a distributed system, such an ADCS requires a precise and accurate distributed time knowledge and reliable sensor readout and actuation at predetermined points in time. At the time of the writing of this thesis, the LRSM team favors usage of an externally developed ADCS over an in-house development for various reasons. Even with an externally developed ADCS, ComPol requires a good time synchronization for reliably and timely exchange of pointing information between the instrument and the ADCS component.

ComPol requires less electrical power and generates fewer measurement data than AFIS. Thus, it does not change the limits of the design space in this regard.

2.1.3 IOV-1

IOV-1 is an ISS based technology demonstrator for the LRSM systems. It will be mounted to the ArgUS multi-payload adapter on the Bartolomeo platform outside the ISS[2]. The main objective of IOV-1 is the verification of the AFIS and ComPol payloads and the LRSM in-house developed CubeSat bus components. IOV-1 furthermore includes an additional payload as proof of principle for a potential future mission measuring the Earth's magnetic field. The long-time goal of this mission called Mesosphere Magnetometry (MesoMag) is the measurement of Earth's magnetic field based on periodic excitation of sodium atoms at 92 km above ground as demonstrated in [30] from a CubeSat based instrument. Within IOV-1, the MesoMag prototype performs a measurement of background noise at 589 nm.

[2]More information on ArgUS and the Bartolomeo platform is available in [48] and https://www.airbus.com/en/products-services/space/in-space-infrastructure/bartolomeo.

The launch of IOV-1 is currently scheduled for mid 2023.

Preliminary Setup

As previously mentioned, IOV-1 contains a prototype of ComPol, AFIS, MesoMag and the CubeSat bus CDH. Individual smaller boxes contain the hardware of each prototype and are interconnected via a main backplane.

The ComPol prototype contains a scaled-down version of detector without collimator. It will demonstrate the operation of the ComPol sensor and meanwhile measure the X-ray background radiation. The AFIS prototype contains a smaller version of the final instrument with only a quarter of the scintillating fibers. During in-orbit operation, the updated AFIS sensor will be verified within the in-orbit thermal vacuum environment and measure the radiation background outside the ISS. The ComPol and AFIS prototypes will use the PDP for data processing. This way, they also verify the interface towards and the capabilities of the PDP in an environment similar to the final CubeSat missions.

The MesoMag prototype contains a 589 nm imager in combination with a visible light reference imager. It gains information about the detector principle and measures the background noise. The visible imager provides a reference to identify potential sources of locally increased background, which is expected above certain regions, e.g., above densely populated areas.

The CubeSat bus prototype contains two CDH units. It verifies the usability of the components in-orbit. At the same time, the CubeSat bus prototype verifies the interface of the CDH towards the payloads, more specifically towards the individual PDPs. Therefore, this prototype can verify the CDH system, its satellite control capabilities, and parts of the distributed features within a reduced setup of only two nodes.

An additional box contains the interface to the ISS, including the electrical power converters and an NVIDIA Jetson-based flight computer connected to the ISS-provided Ethernet. This box does not only act as an interface adapter, but also provides direct access to the debug ports of all prototypes. This way, not only the regular operation but also additional housekeeping data can be extracted. The debug interface furthermore enables detailed debug output, in-orbit updates of all components, and the resolution of potential software issues due to prior undetected behavior as a consequence of the space environment.

Figure 2.1 depicts this setup of the IOV-1 mission. The MesoMag prototype will be pointed towards nadir, the ComPol prototype towards zenith.

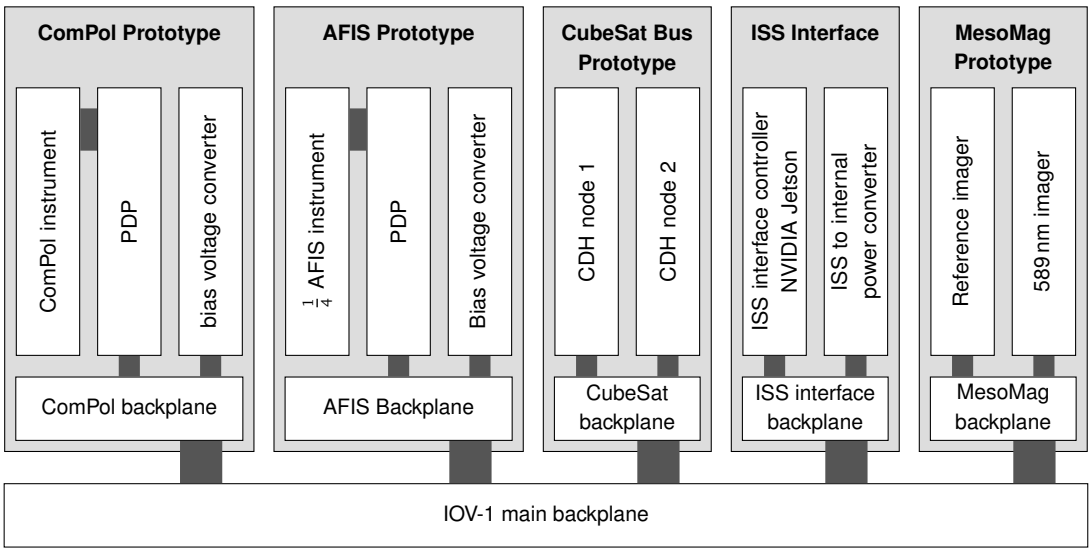


Figure 2.1: Overview of the components of IOV-1 with prototypes separated to individual boxes. The overall setup will be mounted to the ArgUS platform that the MesoMag prototype points to nadir and the ComPol prototype to zenith. Not shown are the openings in the boxes of MesoMag and ComPol for their respective optics.

Impact on CubeSat Platform

In comparison to AFIS and ComPol, IOV-1 has relatively relaxed requirements for the CubeSat platform. It mainly targets the verification of all components; the CDH is the main component of the CubeSat platform for this verification. To enable this testing, the CubeSat platform needs to be flexible enough to allow independent testing and reuse of these components for AFIS, ComPol, and IOV-1. Additionally, the CDH has to handle its redundant units and, in contrast to the later CubeSat missions, interface with multiple PDPs.

2.2 MOVE-III

MOVE-III is the fourth satellite of the MOVE series at the LRT. Similar to its predecessors, it is a student developed satellite, and it is currently in its design phase. While most details about the mission are yet to be defined, the DEDRA sensor will be the scientific payload targeting the detection of small particles in LEO [159].

2.2.1 DEDRA Sensor

DEDRA is part of a collaborative effort to improve space debris models [27]. The sensor itself is a modified and improved version of the Munich Dust Counter (MDC) [160, 161].

The MDC is an impact-ionization-detector that measures mass and velocity of dust particles [76]. This sensor was first used in a highly elliptical orbit with an apogee of up to $1.53 \cdot 10^6$ km on the HITEN (MUSES-A) satellite [77]. Furthermore, the MDC was also part of BREM-SAT [95] and the Mars orbiter NOZOMI [188].

The DEDRA sensor on MOVE-III has, compared to the MDC, a smaller opening of only approximately $8\,\text{cm} \times 8\,\text{cm}$ and fits into 1 U of a CubeSat [160]. Compared to the MDC, an improved version of the DEDRA sensor uses an extended setup for better estimation of the direction of incoming particles [160]. The improved version is slightly larger than 1 U of a CubeSat [160] and estimated to fit into a volume of 1.5 U [161]. DEDRA targets the measurement of particles with a mass of $1 \cdot 10^{-15}$ kg to $1 \cdot 10^{-10}$ kg and a speed of up to $30\,\text{km}\,\text{s}^{-1}$ [160].

2.2.2 CubeSat Platform

MOVE-III including the DEDRA sensor will be a 6 U CubeSat based on the MOVE-BEYOND development [160, 161]. It will contain three DEDRA sensors, where two use the basic version fitting into 1 U and one uses the improved version with better detection of the particle's direction [160, 161]. Similar to the learnings of first-MOVE [106], precise requirements for and details about the satellite bus used for MOVE-III are still unknown.

Due to the student project character, the precursor missions on stratospheric balloons [112] and the unavailable requirements, MOVE-III needs a flexible system enabling early testing of first prototypes and the possibility for adaptations at a late stage of the development.

2.3 Radiation Environment in Space

This section provides an introduction to the radiation environment in a LEO orbit relevant for the previously introduced missions and this environment's effects on a satellite. Specifically, it introduces the high energetic particles and their effects on embedded electronics.

2.3.1 Radiation Sources

Three main sources of energetic particles are most relevant for LEO orbits: Galactic cosmic rays from outside the solar system, particles emitted from the Sun, and particles trapped in the geomagnetic field [145].

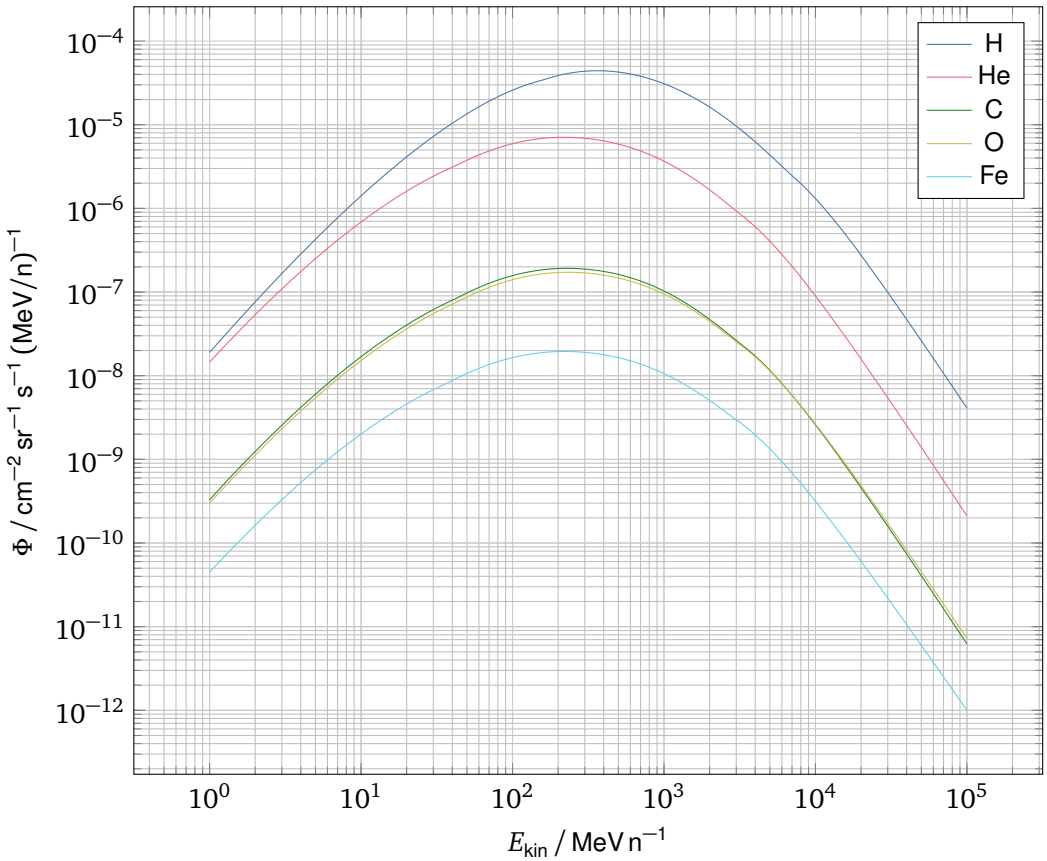


Figure 2.2: Differential flux of galactic cosmic rays in a 600 km polar orbit during a solar minimum. Shown is the flux of hydrogen (H), helium (He), carbon (C), oxygen (O), and iron (Fe) nuclei. Kinetic energy normalized to the respective number of nuclei. Data generated with SPENVIS (https://www.spenvis.oma.be/) using the ISO 15390 model.

Galactic Cosmic Rays

Galactic cosmic rays are particles at very high kinetic energy of up to 10^{21} eV [145]. Most of those particles are hydrogen (H) or helium (He) nuclei and only about 1 % are nuclei of heavier elements [140, 196]. Due to the increased rate of energy loss of heavier nuclei, nuclei up to iron (Fe) contribute significantly to the rate of transferred energy [140]. The Lunar environment provides a good estimate for the galactic cosmic rays outside Earth's protective magnetic field. The main contribution of H (31 %), Fe (16 %), and He (9.4 %) to the total dose of galactic cosmic rays in this environment [179] demonstrates that heavier nuclei cannot be neglected. The Sun's activity modulates the absolute flux of galactic cosmic nuclei; a maximum in the Sun's activity reduces the flux of galactic cosmic nuclei [13]. Overall, the flux is relatively low with only a few particles per cm^2 s [57]. Additionally, Earth's magnetic field deflects parts of these particles [57, 145].

Figure 2.2 depicts the galactic cosmic ray flux of selected elements in a circular 600 km polar orbit.

Solar Energetic Particles

The so-called solar wind consists of a large amount of particles, mainly protons and electrons [190], at a speed of up to 800 km s^{-1} and thus a relatively low kinetic energy of up to a few keV [178]. Additionally, solar events may generate bursts of high energetic particles with a kinetic energy of up to several GeV [57, 144, 178]. Similar to the galactic cosmic rays, these bursts consist mainly of electrons, He nuclei, and H nuclei but also contain larger ions [54, 178, 190]. These ions, especially those from elements with a larger core charge, are only partially ionized, i.e., they may contain remaining electrons [54]. The largest proton events have a flux of 10^2 to 10^4 protons per cm^2 s sr with a kinetic energy of more than 10 MeV [144]. Those extreme events have a duration in the order of a few seconds

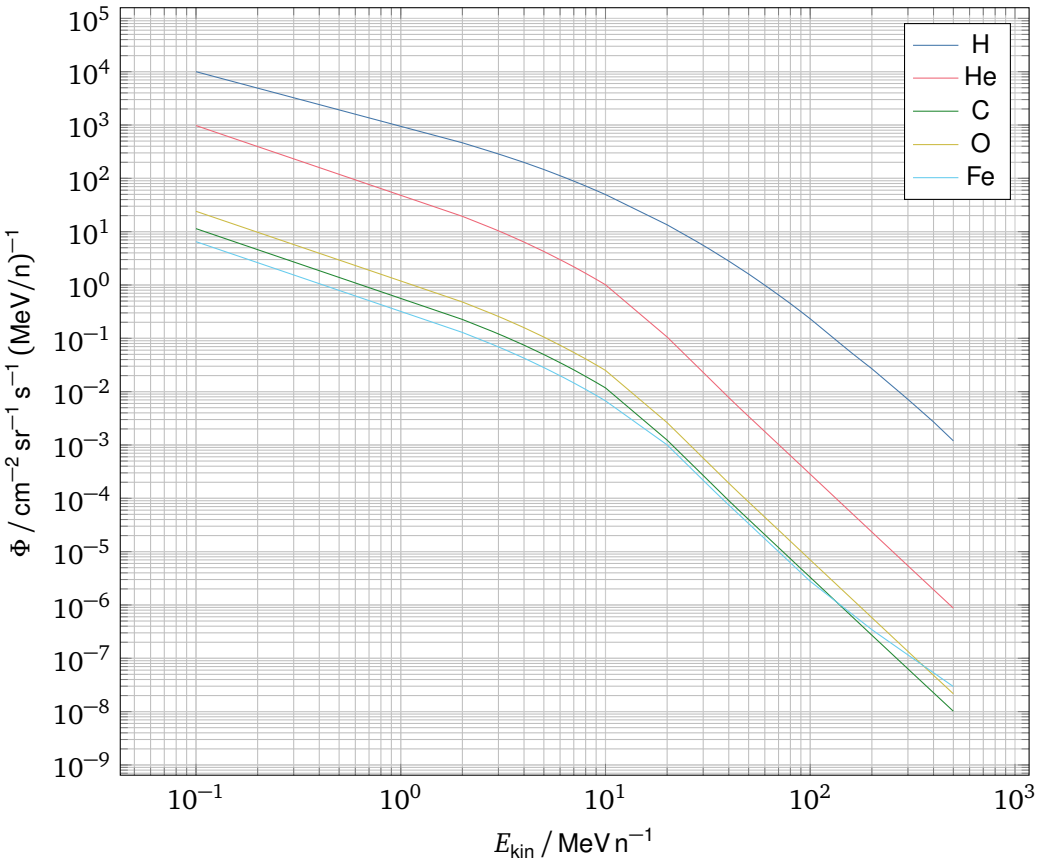


Figure 2.3: Worst week average differential flux of solar cosmic rays in a 600 km polar orbit. Shown is the flux of H, He, C, O, and Fe nuclei. Kinetic energy normalized to the respective number of nuclei. Data generated with SPENVIS (https://www.spenvis.oma.be/) using the CREME-96 (worst week) model.

only [42, 144]. While these events also emit neutrons at high kinetic energies [42, 144], most neutrons decay rapidly and do not reach Earth [57]. Again, Earth's magnetic field deflects parts of the charged particles and thus provides protection especially against those with lower kinetic energy [144].

The solar activity and thus the flux and kinetic energy of solar cosmic rays vary in the long term with solar cycle (scale of years) and in the short term with solar jets (scale of hours) [178]. Figure 2.3 depicts the average flux of particles originating from solar cosmic rays of a single worst week.

Trapped Particles

The third and due to the high flux one of the major sources of radiation in LEO are charged particles trapped in Earth's magnetic field. These particles travel in spiraling lines along the magnetic field lines and bounce off the magnetic mirror due to the field gradient near the magnetic poles [75].

Van Allen Belts The so-called Van Allen belts — named after James Van Allen who first observed them in 1958 [221] — are belts of trapped particles around Earth's magnetic equator [222]. The inner Van Allen belts consist mostly of trapped protons and electrons [222] with a kinetic energy of up to a few MeV and a few hundred MeV respectively [57, 145]. It affects high altitude LEOs and extends up to a height of about 2.5 Earth's radii[3] [145]. The outer Van Allen belt extends to about 10 Earth's radii and mainly consists of trapped electrons [145].

South Atlantic Anomaly In principle, satellites in low altitude LEO are not directly affected by the Van Allen belts [145]. Only due to an asymmetric magnetic field and the lower field strength above the

[3] About 9500 km above Earth's surface.

southern Atlantic Ocean [41], the lower Van Allen belt can reach down to about 200 km above Earth's surface in this region. Due to this anomaly called the South Atlantic Anomaly (SAA), satellites in a LEO may still pass through the Van Allen belt in this region and are thus affected by trapped protons and electrons.

Figure 2.4 depicts the flux of protons with a kinetic energy above 20 MeV to visualize the SAA. According to this data, we expect a flux of about $3 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ of such protons for the peak of a SAA pass. Figure 2.5 depicts the average flux of protons and electrons in a polar LEO orbit. Although a major part of this radiation originates from the SAA, other regions also contribute to this average.

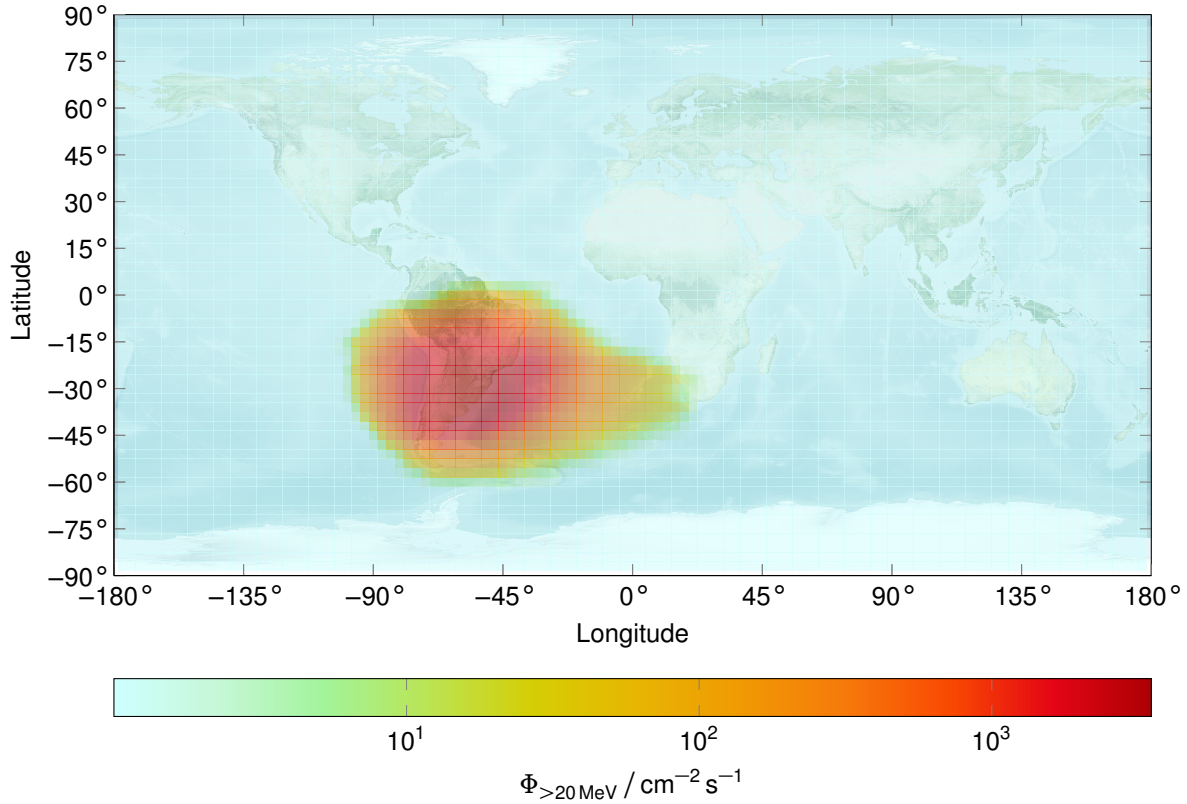


Figure 2.4: World map of trapped proton flux for kinetic energy above 20 MeV on a 600 km polar orbit during a solar minimum. Data generated with SPENVIS (https://www.spenvis.oma.be/) using the AP-8 model. Background image credit: NASA Earth Observatory map by Joshua Stevens using data from NASA's MODIS Land Cover, the Shuttle Radar Topography Mission (SRTM), the General Bathymetric Chart of the Oceans (GEBCO), and Natural Earth boundaries (https://visibleearth.nasa.gov/images/147190/explorer-base-map/147191l).

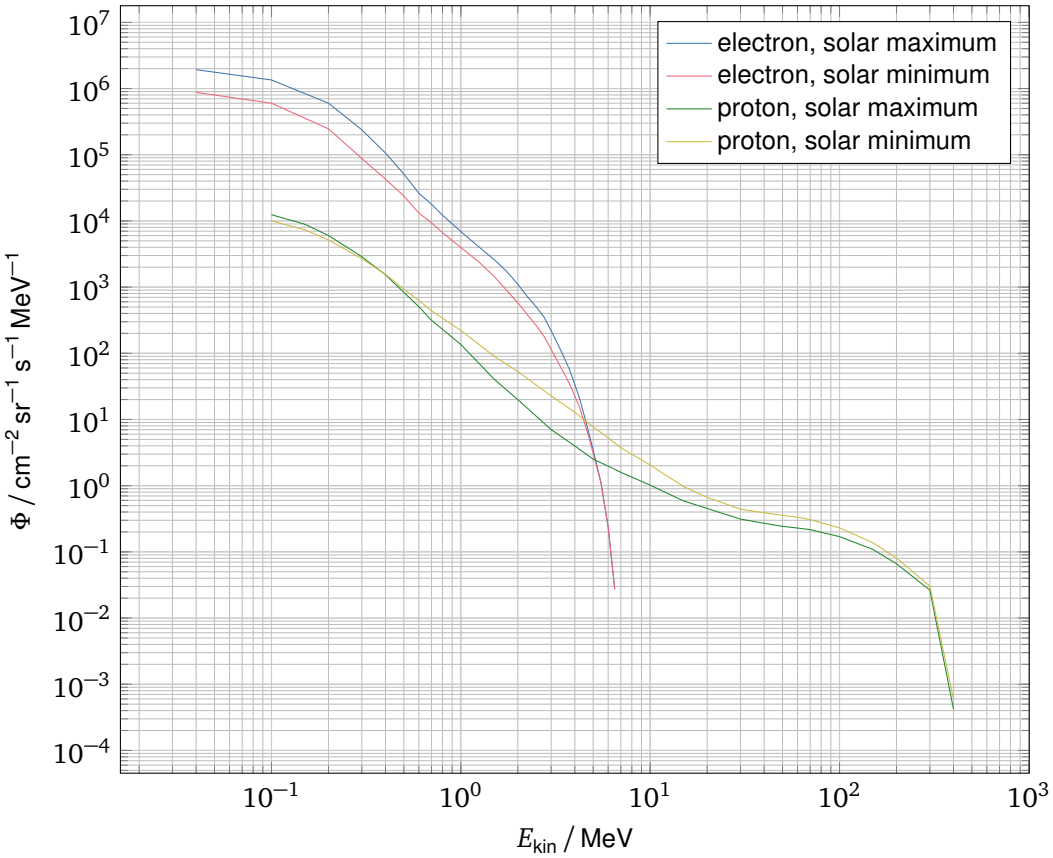


Figure 2.5: Differential flux of trapped electrons and protons in a 600 km polar orbit. Data generated with SPENVIS (https://www.spenvis.oma.be/) using the AP-8/AE-8 model.

2.3.2 Effects on Electronics

Radiation affects the functionality of semiconductors as used in modern electronics. This can be separated into three types of effect: displacement damage, single event effect (SEE) as the result of a single ionizing impact, and total ionizing dose (TID) effects as a cumulative effect over the entire lifetime of a component [56, 223]. Out of these types, SEE and TID effects have the largest influence on commonly used complementary metal-oxide-semiconductor (CMOS) electronics [58].

Total Ionizing Dose

The total ionizing dose (TID) is a measure for deposited energy E per unit mass m of a specific target material [56]:

$$D_{\mathrm{TID}} = \frac{\mathrm{d}E}{\mathrm{d}m} \tag{2.1}$$

Although Gy is the standard unit of the International System of Units (SI), TID is mostly given in rad in literature, where $1\,\mathrm{rad} = 0.01\,\mathrm{Gy}$ [58].

An increasing TID leads to gradual degradation of electronic devices [168]. Transistors and integrated electronics such as CMOS devices are most sensitive to TID related degradation [56]. Typically, CMOS devices suffer from decreased switching speed, increased leaking current, and a reduced threshold voltage and driving current [58]. Depending on the used technology, the TID tolerance is in the range of 10^1 Gy to 10^4 Gy [56]. Some COTS MCUs where successfully tested at a TID of 100 Gy to 500 Gy [32, 53, 197]. Still, a careful selection of those components is necessary as the TID tolerance of COTS in general varies in a wide range of 10^1 Gy to 10^4 Gy [125, 197].

As the TID directly affects the switching characteristics of the hardware, e.g., of an MCU, no mitigation in software is possible. Special hardware tolerating the expected TID must be used or the TID must be reduced to an acceptable level using appropriate shielding.

Table 2.1: Density of material used for shielding estimation.

Material	Density ρ / g cm^{-3}	Reference	Normalized Density ρ/ρ_{Al}
Al	2.70	[74, p. 4-44]	1.00
FR-4	1.92	[210][i]	0.71
Cu	8.96	[74, p. 4-59]	3.32
SiO_2	2.20 to 2.65	[74, p. 4-84]	0.81 to 0.98
Packaging	2.20	estimate[ii]	0.81

[i] Used as estimate, as final material and vendor are unknown. [ii] Lowest SiO_2 density used, as it is the main component and specific details about the final material are unknown.

Shielding The shield depth specified in either mm aluminum (Al) equivalent or g cm^{-2} — the latter is normalized to the shielding material's density — is used to estimate the effects on the radiation environment. Although the effects of different materials slightly differ, we estimate the Al equivalent depth based on the density of the target materials for a first approximation:

$$d_{\text{Al equiv.}} = d_{\text{shield}} \cdot \frac{\rho_{\text{shield}}}{\rho_{\text{Al}}} \tag{2.2}$$

We estimate the TID within a CubeSat based on a simplified 1 U CubeSat model. This CubeSat consists only of six regular PCBs covering the sides of the CubeSat. Each PCB is assumed as a 1.55 mm FR-4 board with 2 full layers of 35 μm of copper (Cu). Additionally, we assume the packaging of the chip to surround the target chip's die with 0.5 mm of packaging plastic. We estimate the packaging plastic's density — which consists mainly of silicon dioxide (SiO_2) filler (70 % to 90 %), epoxy resin, hardener resin, and some additives in smaller amounts [94] — to 2.2 g cm^{-3}.

$$\begin{aligned} d_{\text{chip in CubeSat; Al equiv.}} &= d_{\text{FR-4}} \cdot \frac{\rho_{\text{FR-4}}}{\rho_{\text{Al}}} + d_{\text{Cu}} \cdot \frac{\rho_{\text{Cu}}}{\rho_{\text{Al}}} + d_{\text{packaging}} \cdot \frac{\rho_{\text{packaging}}}{\rho_{\text{Al}}} \\ &\approx 1.7\,\text{mm} \end{aligned} \tag{2.3}$$

Table 2.1 lists the densities used for this calculation. The calculated Al equivalent shield depth is a worst-case estimate. A real CubeSat will contain additional PCBs, including the carrier of the chip itself, and most likely is covered in solar cells. Thus depending on the direction, the actual Al equivalent shield depth will be larger than the estimated 1.7 mm, which further reduces the expected TID.

Figure 2.6 depicts the expected TID after one year in a 600 km polar orbit for different shield depths. The horizontal line indicates the previously calculated minimum shield depth. Based on these values calculated with the Space Environment Information System (SPENVIS)[4], we expect a TID for a chip within the CubeSat of less than 60 Gy per year.

Single Event Effects

This section summarizes the information about SEEs from *Single Event Effects in Aerospace* [168], unless specifically stated otherwise.

As previously mentioned, a number of high energetic particles from various sources reach all components within a spacecraft. These particles also cross electronic components and interact with the component's fabric. Energetic particles generate electron-hole pairs along their path in semiconductors and deposit their kinetic energy this way. Nuclear reactions of these particles, especially of protons, may additionally deposit parts of the kinetic energy and generate secondary particles. Within the electronic components, especially within semiconductors, the deposited energy leads to various unwanted SEEs. In contrast to the previously introduced TID effects, a SEE is the result of a single particle hitting a device and thus has a localized effect.

[4] https://www.spenvis.oma.be/

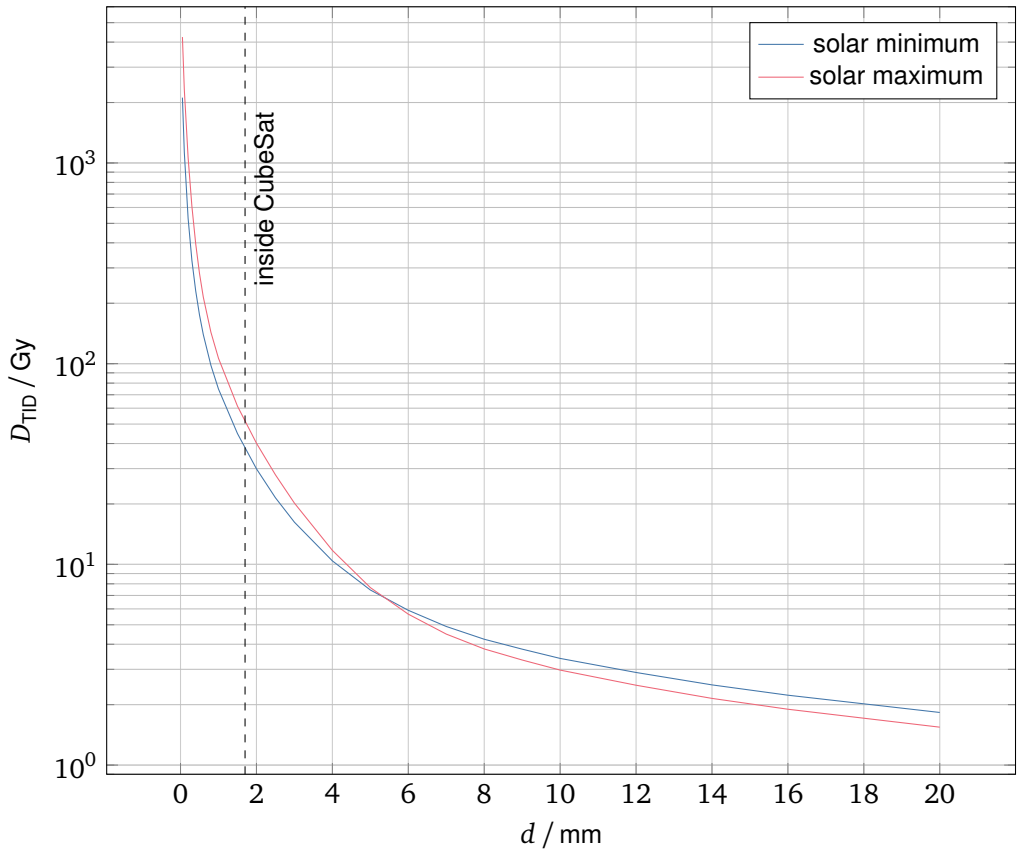


Figure 2.6: D_{TID} in SiO_2 due to trapped particles and solar protons after 1 year in a polar 600 km LEO for different Al equivalent shield depths d. The vertical line indicates the Al equivalent shied depth for a chip inside a CubeSat surrounded by regular PCBs. Data generated with SPENVIS (https://www.spenvis.oma.be/) with SHIELDOSE-2 as suggested in [58]. Proton and electron fluxes generated with AP-8 and AE-8 models respectively.

In digital electronics, such a SEE may lead to a single event upset and corrupt bits of information. Such a single event upset may be transient, permanent, or static: A transient effect only affects a local signal that is not yet stored in any latch or register. They may be overridden by the surrounding logic prior to the next clock edge and thus remain hidden. A static effect, on the other hand, changes the content of a latch or register, and is observable for an extended time. The observable effect ranges from a single changed memory bit, over multiple simultaneously changed bits, to the temporary interruption of (parts of) the device's functionality. Static errors can be corrected by resetting the affected memory location or power cycling the device, and they do not permanently affect the functionality. Permanent errors on the other hand permanently damage the device. An example for such an error is a single event latchup in a CMOS device that short circuits a signal and enables a high current flow. This current flow in turn may generate excessive heat and thus thermally damage the device.

Linear energy transfer (LET) is used to analyze the ionizing effect of a specific impact; it states the deposited energy per distance traveled within the target component and is derived from the stopping power of the target material. The LET is an average value and depends on the particle properties and target material. It is typically specified in units of either $\text{MeV}\,\text{mm}^{-1}$ or $\text{MeV}\,\text{cm}^2\,\text{g}^{-1}$, where the latter is a normalization with respect to the target material's density. Note that the LET is not physically accurate as it assumes a constant stopping power of the target material, even though it is not constant along the path. A particle deposits most of its kinetic energy just before it is entirely stopped. Figure 2.7 depicts the stopping power of Si for electrons, protons, and He nuclei. This simplification of the effect may be acceptable as the targets are relatively thin, e.g., a few hundred μm for a Si based digital circuit. We will use the symbol L to reference to the LET in formulas and figures.

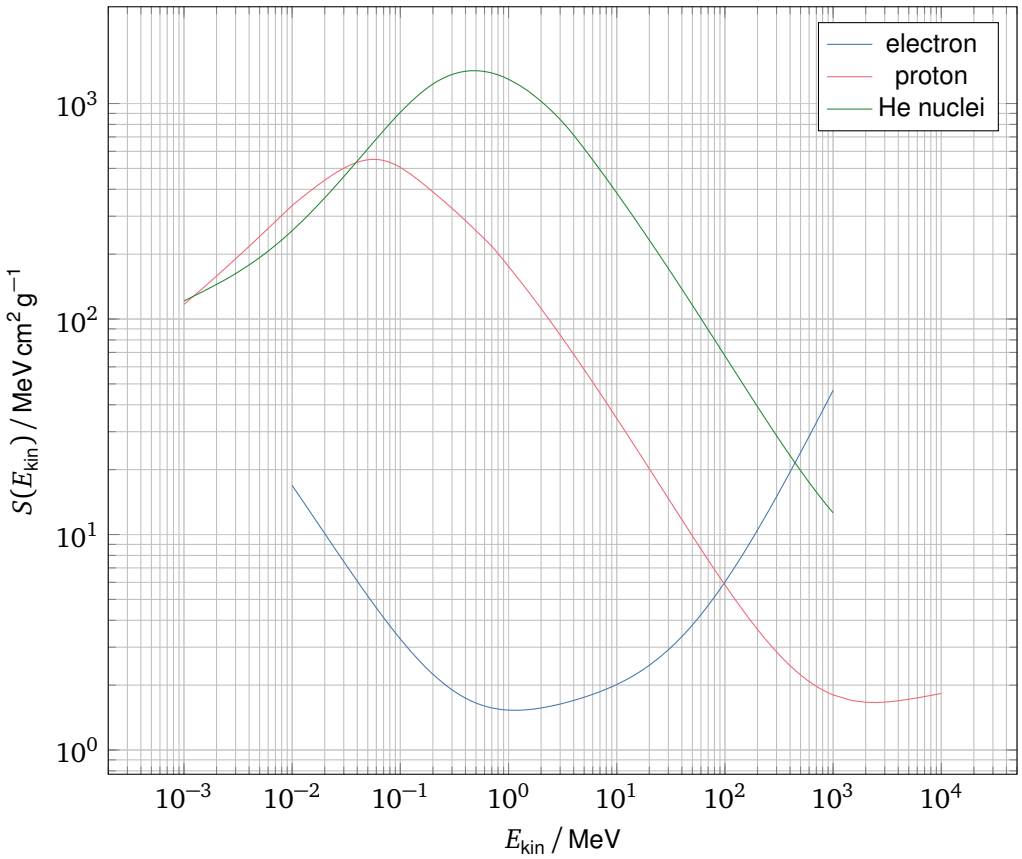


Figure 2.7: Stopping power of SI for electrons, protons and He nuclei. Data from https://physics.nist.gov/PhysRefData/Star/Text/contents.html [194].

Shielding Shielding can significantly change the rate of expected SEEs. Not all particles presented in section 2.3.1 will actually hit and potentially generate a SEE. Instead, these particles have to first pass the shield and therefore already deposit parts of their kinetic energy. Particles with a low kinetic energy deposit their entire kinetic energy within the shielding and never reach the sensitive electronics.

Figure 2.8 depicts the range of electrons, protons, and He nuclei-particles in Al according to [194]. A shield of 1.7 mm Al, which is equivalent to a shield depth of 0.46 g cm^{-2}, already stops protons with a kinetic energy below 20 MeV.

2.3.3 Summary

For a CubeSat in LEO, we expect a radiation environment mainly consisting of protons and electrons combined with a few heavier particles. The PCBs mounted on the outside of a CubeSat act as a shield for parts of this radiation and stop particles with a kinetic energy below 20 MeV n^{-1} from reaching sensitive electronics. The flux of protons above $1 \cdot 10^5$ MeV is below one proton per year and cm^2.

Another important aspect of the radiation environment are the Van Allen belts and the increased flux within the SAA. An expected proton flux in the maximum of the SAA for protons with a kinetic energy above 20 MeV of approximately $3 \cdot 10^3$ cm^{-2} s^{-1} may severely influence the safe operation within this region. Other CubeSats already reported an increased bit error rate within this region [29]. As AFIS specifically requires reliable operation within the SAA, the bit errors due to SEEs must be mitigated.

The expected TID of approximately 60 Gy per year for electronics inside a CubeSat may also affect the reliable operation of electronics. While we can neglect TID effects on short-term missions, a mission with a duration of more than one year requires a careful selection of components or additional shielding.

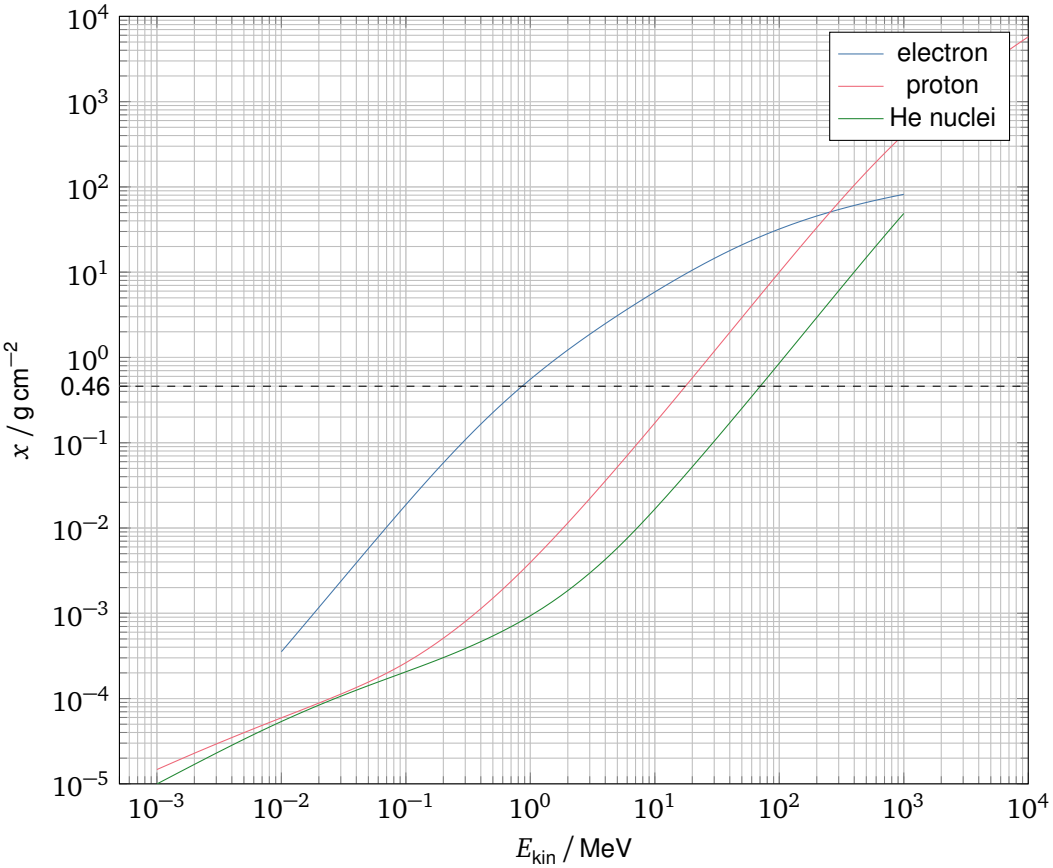


Figure 2.8: Range of electrons, protons, and He nuclei in Al based on the continuous-slowing-down approximation. Data taken from https://physics.nist.gov/PhysRefData/Star/Text/contents.html [194]. An Al shield with a depth of 1.7 mm entirely stops all particles below the horizontal line at 0.46 g cm^{-2}.

2.4 Design Goals

As previously presented, the different missions' requirements diverge. For example, the available power and communication bandwidth are the limits for AFIS, but the ADCS capabilities and timely knowledge of the pointing are the driving factor for ComPol. A system perfectly suited for one of these missions might not be a good candidate for the others.

Design goals provide a way to compare different scenarios or solutions and influence design decisions. For this thesis they act as a rough guideline to enable the design of a solution, although detailed requirements are yet to be defined. The presented criteria do not have a fixed precedence and are presented in alphabetical order.

Design Goal 1 (Component Availability) *The used software and hardware components should be available for CubeSat missions at TUM.*

The availability of components is an important design goal for the proposed system. It is important to only select components that can be used within the LRSM and MOVE projects. Therefore, COTS should be used instead of highly specialized parts. If no such component exists, building it in-house is preferred over one-of-a-kind products of external companies without the possibility to easily get a reliable supply. All software components should be either licensed by TUM or available as open-source software. In any case it should be possible to use the software for academic and scientific purposes.

Design Goal 2 (Expandability and Flexibility) *The proposed framework should be expandable and adaptable to the needs of future LRSM missions.*

Expandability and flexibility are important factors enabling a reuse of the OBC for various not yet defined missions. It also accounts for the fact that most requirements of the upcoming LRSM and MOVE

missions are not yet final. To compensate this lack of knowledge, the proposed system must provide the flexibility to add new components or replace existing components of the overall system at a later stage of the development.

Design Goal 3 (Power Consumption) *The OBC should not consume significantly more power than other CubeSat OBCs.*

The available electrical power on a CubeSat is limited. Therefore, it is crucial to reduce the power consumption of the OBC. A reduced power consumption relaxes the design constraints on other subsystems and thus potentially enables longer scientific operation.

Design Goal 4 (Reliability) *The system should allow reliability measures either embedded into the framework, or at least allow the users of the system to define those measures on top of the proposed framework.*

The harsh space environment increases the demand for reliability compared to most other use cases. Not only is the environment hostile, but it is also unfeasible to perform manual maintenance once the satellite is in orbit. Therefore, it is important to reduce the potential of an unresponsive system. The system should be kept as simple as possible to reduce the potential for hardware or software bugs. The used components should account for the harsh environment and reduce the risk of a mission loss as far as possible.

Design Goal 5 (System Load) *The proposed solution should not introduce a large overhead that generates a high load at the target platform.*

The system load is a good measure for the efficiency of the resource utilization. Reducing the load while not harming the functionality of the system as such also aids in satisfying other design goals. A system with only minimal load in its nominal state still provides a potential for future expansion and adaptation to new missions. Additionally, a reduced system load can benefit the overall power consumption and responsiveness of a system.

Chapter 3

System Design

The system design provides the critical information for a later implementation of the suggested system. It is based on the knowledge about the AFIS and ComPol mission specific requirements and the learnings from previous CubeSats at the LRT. In combination with the implementation presented in chapter 4 this chapter fulfills objective 1 as it provides the entire design of the suggested framework for distributed control in CubeSats. As previously stated in section 1.5.1, we aim at a simple solution that meets the requirements presented earlier and has the potential for future adaptations and extension. This way, the suggested system can evolve in the future and adapt to the needs of upcoming missions. To enable future changes to the framework, this chapter provides insight into the selection of the basic setup and a detailed description of the suggested framework with a focus on the interconnection of the various components and the intended usage.

This chapter starts with the selection of a framework or OS in section 3.1 that already provides many of the features required for the suggested system. Afterward, section 3.2 selects a topology and interface standard for the interconnection of nodes within the distributed system. This selection of the RODOS OS and an interconnection of all nodes with a CAN bus network provides the basic setup for the suggested system.

Section 3.3 provides an overview of the Distributed Operating System Initiative for Satellites (DOSIS) framework proposed in this thesis. Afterward, section 3.4 and section 3.5 present the basic building blocks of this framework called *Components* and *Modules*. Section 3.6 details the communication abstraction within DOSIS, which is based on the selected CAN bus and the capabilities provided by RODOS. The introduction of the core of DOSIS ends with section 3.7, which presents used to logically connect different parts of the system.

Objective 2 demands timing guarantees to enable a distributed real-time control. Therefore, the system requires a time synchronization, which section 3.8 presents in detail. This chapter ends with an introduction to fault tolerance mechanisms, which designers of a specific mission may use on-demand, in section 3.9.

3.1 Baseline Framework

Publicly available frameworks and OSs can provide parts of the basic infrastructure required to implement a framework as wanted by objective 1. In particular, they can provide a suitable hardware abstraction and a message-passing mechanism to enable basic communication. Real-time scheduling and the management of concurrent threads are other important aspects that should not be re-invented within this thesis. Although this is mostly required for the implementation, it also has some effects on the design.

We use a multi-criteria decision analysis based on the method presented by Zangemeister [237] for this selection process.. This method compares a final score of the individual solutions to select the best match. The final score of each solution is a weighted sum of the scores for the different criteria.

3.1.1 Exclusion Criteria

Exclusion criteria prohibit the use of a certain solution. Thus, we exclude any framework or OS meeting one of the following criteria from the decision process.

Exclusion Criterion 1 (Availability) *Neither the source nor a binary distribution of the framework or OS is available.*

Any framework or OS that is only mentioned in literature but not available as source code or in a binary distribution suitable for linking the software cannot be used to build an executable software. Without an executable software it is not possible to satisfy objective 3 (Demonstrate on Target Hardware). Additionally, this would not allow the use of the framework in any real application and thus no benefit from such a framework could be expected.

Exclusion Criterion 2 (Suitability for small MCUs) *The framework or OS is not suitable for embedded low-power MCUs.*

Smaller embedded MCUs with a focus on low-power rather than high performance are common in CubeSat platforms. Such MCUs often do not have a dedicated MMU or are limited regarding the available code memory and RAM, which does not exceeding a few MB for most of these MCUs. Especially for distributed applications based on multiple nodes, the power consumption of each individual node is critical. Thus, it should be possible to run software based on the proposed framework even on small nodes without MMU and less than 1 MB of code storage and RAM. Using a high-power platform would limit the possible use of the suggested framework to larger CubeSats with relaxed power constraints. Therefore, we exclude frameworks or OSs designed for larger MCUs or other high-power platforms from the selection.

3.1.2 Selection Criteria

Three groups of criteria provide the baseline of the selection process for the remaining candidates: criteria regarding availability and support (criteria 1 to 3), criteria regarding portability (criteria 4 to 5), and criteria regarding features (criteria 6 to 8).

Criterion 1 (Open-source) *The source code of the framework or OS is publicly available and uses an open-source license.*

To develop a framework based on a specific OS or other framework in an embedded application it is critical to have access to the sources thereof. Not only the availability of the software, but also the license of the software is important to maximize the possible future use cases of the framework developed within this thesis. Less restrictive licenses enable a wide range of possible future use cases for institutional and commercial applications in the future. If the source code is not available at all, the corresponding framework or OS is excluded from the selection process by exclusion criterion 1.

Criterion 2 (Documentation) *The framework's or OS's documentation is complete and meet basic quality criteria.*

Documentation provides the insight into a framework or OS for a better understanding of the functionality and proper use of its features. A complete documentation contains at least some information on how to get started using the framework, a documentation of the software interfaces, and background information on the intended use of the features of the framework or OS.

Criterion 3 (Support) *The maintainers and/or the community still actively support the framework or OS.*

Questions not answered by the documentation might still be relevant during development. In such cases an active community or maintainer can provide additional information. Additionally, active maintainers and community members assure that the software is kept up-to-date.

Criterion 4 (Ports) *Ports for target devices, especially target devices with a small MCUs, are available.*

A software port provides the necessary adaptations required for a specific target platform. Compatibility of a software with a wide range of target platforms increases the flexibility to use said software for various applications. Ports for small MCUs potentially without MMU show the use of the framework or OS in embedded systems, which many CubeSats use for their on-board control system. An available port to a desktop system such as Linux simplifies development.

Criterion 5 (Porting) *It is simple to port the framework or OS to a new target platform.*

A low effort of porting the framework or OS to a new target platform simplifies the use in new systems. This will decrease the time needed to adapt to a new hardware generation or a changed platform due to external constraints. We estimate the effort of porting the framework based on the structure of the software and the available documentation specifically targeting the process of porting the system.

Criterion 6 (Multi-threading) *The framework or OS supports multi-threading.*

Multi-threading enables the executing distributed applications within a single node. Supporting functionality for thread synchronization simplifies the application development. Examples for support functionality are semaphores and mutexes, protected regions or atomic execution.

Criterion 7 (Real-time) *A scheduler is available and supports real-time scheduling capabilities.*

A scheduler assigns execution times to different tasks or threads. For control loops in time critical systems the real-time capability of a scheduler is of high importance. Support for simplified activation at a certain time in the future or in a certain interval simplifies development of timed user applications. This is required not only for time critical control loops (e.g., for ADCS) but also for low priority tasks like telemetry gathering or regular self checks of the software.

Criterion 8 (Message-passing) *The framework or OS includes a message-passing mechanism, preferably a publisher-subscriber based message-passing.*

Support of message-passing is required for a distributed application. It enables the communication between different threads on a single node or between different nodes. For the framework developed within this thesis a simple message-passing scheme is beneficial. Especially publisher-subscriber based messaging allowing multiple receivers of a single message can simplify the higher level messaging.

Value Scale and Weight We evaluate all criteria and assign a score on a scale from 0 to 1 where 0 is the lowest and 1 the highest possible score. A preference analysis provides the relative weight of each criterion. Table 3.1 presents the resulting weight w_i for each criterion. Appendix B.2 presents details on the pairwise comparison and the calculation of the weights.

3.1.3 Available Frameworks

Section 1.3.3 already presents available frameworks for on-board computing on small satellites. First, we check the exclusion criteria for these frameworks. Afterward, we rate the remaining candidates according to the previously presented criteria.

Exclusion Table 3.2 presents the exclusion of candidates according to exclusion criteria 1 and 2. The evaluation of exclusion criterion 1 is based on the online availability of a software and does not include the license of the framework; i.e., we do not exclude a software that is available online but does not use an open-source license. Similarly, we do not exclude a framework due to exclusion criterion 2 if it does neither specify a particular architecture nor exclude smaller MCUs specifically.

Table 3.1: Weight w_i of selection criteria based on result of preference analysis in appendix B.2. Criteria in this table are sorted by their weight.

Criterion	i	w_i
Real-time	7	0.22
Multi-threading	6	0.19
Documentation	2	0.16
Porting	5	0.14
Support	3	0.11
Message-passing	8	0.11
Ports	4	0.05
Open-source	1	0.03

Table 3.2: Exclusion of frameworks not suitable for selection due to Availability (1) and Suitability for small MCUs (2). Only frameworks passing both criteria will be candidates for the selection.

Framework	**Availability**	**Suitability for small MCUs**	**Candidate for selection**
CAST	✗	?	✗
cFS	✓	✓	✓
COrDeT	✓	?	✓
CubedOS	✓	✓	✓
F'	✓	✓	✓
GERICOS	✗	✓	✗
KubOS	✓	✗	✗
NMF	✓	✗	✗
OBC-NG / ScOSA	✗	✗	✗
SPA	✗	✓	✗
fsfw	✓	✓	✓

Rating We rate the remaining five frameworks by assigning a value to each criterion and calculate the final score as weighted sum of the value of the criteria. Appendix B.3 presents the rating for each framework in detail. Table 3.3 summarizes the result of this process.

3.1.4 Available Operating Systems

OSs are a second group of candidates as baseline of the proposed framework. Therefore, available OSs that are intended for space applications or used by one of the previously presented frameworks in a real space environment are additional candidates for the selection process. This section first presents the candidate OSs in alphabetical order, followed by applying the exclusion criteria and evaluating the remaining candidates.

FreeRTOS The FreeRTOS kernel is a minimalistic real-time capable kernel targeting small MCUs with only limited memory and processing resources [8]. Libraries for Transmission Control Protocol (TCP) [10] and standard connectivity protocols [9] simplify the development of embedded applications. Additional support for the Amazon Web Services for internet of things applications [11] simplifies development of interconnected embedded devices. A lot of documentation and other online resources are available[1]. FreeRTOS is publicly available and released as open-source project[2].

[1] https://www.freertos.org/
[2] https://github.com/FreeRTOS/FreeRTOS

Table 3.3: Summary of the rating of candidate frameworks. The final score for each framework calculates as the sum of the weighted score of the individual criteria. The weights shown in table 3.1 are used. The detailed analysis and reasons for the given scores is available in tables B.4 to B.12

Criterion	cFS	COrDeT	CubedOS	F'	fsfw
Open-source (1)	1	1	0	1	1
Documentation (2)	1	0.8	0.6	1	0.6
Support (3)	1	0.6	0.4	1	0.8
Ports (4)	1	0	1	1	1
Porting (5)	0.25	0	0	1	0.75
Multi-threading (6)	0.33	0	0	0.92	1
Real-time (7)	0	0	0	0	0
Message-passing (8)	0.6	0	0.4	0.6	0.6
Final Score	0.51	0.22	0.24	0.73	0.62

Linux and µClinux The well known Linux OS is another candidate for this selection. Although Linux targets mainly larger systems, such as desktop or server systems, it can still be used in embedded applications [234]. μClinux, a special version of Linux, extends the range to smaller MCUs without a dedicated MMU [234]. Various spacecraft use Linux as the main OS of the on-board computer [113]. Leppinen [113] states the main benefits of Linux in spacecraft software as the reliable Linux source code maintained by a large community, the support for a wide range of platforms and communication protocols, and the available software.

RODOS RODOS is a simplistic real-time operating system designed for usage in systems with a high demand for dependability [151]. It is designed as network centric operating system, thus aims to provide a reliable network for less reliable nodes [151].

According to Montenegro and Dannemann [149], RODOS is split into two major parts, the core, and the middleware. The core provides a simple micro-kernel including resource management, threading, input/output abstraction and interrupt management [149]. On top of the RODOS core, the middleware implements a publisher-subscriber based messaging mechanism over so-called RODOS topics [149]. This messaging does not only support communication within a single hardware unit, but also provides gateways to connect multiple physical components into one large system [149]. The location of publishers and subscribers in such a network can be dynamic and change at runtime of the system [149]. RODOS is implemented in C++ and offers an object-oriented interface [149].

The source code of RODOS is maintained by Prof. Montenegro at the University of Würzburg and is publicly available on GitLab[3].

RTEMS RTEMS is a real-time operating system with support for multiple APIs and support for 18 different processor types [215]. It provides thread synchronization, locking features and a configurable real-time scheduler [214]. Access to different file systems, periphery interfaces and support for BSD user space applications is available as part of RTEMS [214]. Various board support packages provide the required information to connect a processor and a specific target platform [214]. Extensive documentation of RTEMS is available online[4], the source code is available in a git repository[5].

ThreadX Azure real-time operating system (RTOS) ThreadX is a OS for embedded internet of things applications and is developed by Microsoft [136]. ThreadX has a minimal footprint of only a few kB for instructions and RAM and is thus suitable for a wide range of MCUs [136]. The ThreadX picokernel architecture reduces the amount of layers and connects all components directly to the core of the

[3]https://gitlab.com/rodos/rodos
[4]https://ftp.rtems.org/pub/rtems/releases/5/5.1/docs/html/
[5]https://git.rtems.org

system [135]. While maintaining a small footprint, ThreadX provides a wide range of features for multi-threading, time and memory management, interrupt handling, and a priority based preemptive scheduler [136]. ThreadX complies with the Motor Industry Software Reliability Association (MISRA)-C standard, and is UL and TüV certified [135]. Additional information on ThreadX is available in the Microsoft online documentation [142] and the ThreadX GitHub repository[6].

VxWorks VxWorks is a real-time OS used for various applications over the last 30 years [233]. It offers a save, secure, reliable, and certifiable platform for many areas, including certification for avionic systems [232]. VxWorks includes networking support, a fault-tolerant file system, and higher level support for multimedia and artificial intelligence applications [233]. Developers can tune the set of features for their specific application and combine it with drivers and support for a target platform packaged into a board support package. Windriver provides wide range of board support packages online[7]. Additional information is available at the Windriver product page[8]. The source code of VxWorks is not publicly available.

Exclusion and Rating The exclusion of OS from the selection process uses the criteria presented in section 3.1.1. Table 3.4 presents the result of the exclusion process. Only VxWorks is not publicly available and therefore excluded from further analysis. Regular Linux is not suitable for smaller embedded controllers, especially controllers without MMU. μClinux, the embedded version of Linux is no longer available online. The project website[9] is no longer publicly available[10], thus no source code of μClinux is available. Therefore, no version of Linux does take part in the selection process.

Table 3.4: Exclusion of operating systems not suitable for selection due to Availability (1) and Suitability for small MCUs (2). Only OSs passing both criteria will be candidates for the selection.

Framework	Availability	Suitability for small MCUs	Candidate for selection
FreeRTOS	✓	✓	✓
Linux	✓	✗	✗
RODOS	✓	✓	✓
RTEMS	✓	✓	✓
ThreadX	✓	✓	✓
μClinux	✗	✓	✗
VxWorks	✗	✓	✗

The remaining OSs are rated based on the criteria defined in appendix B.1. Table 3.5 presents the result of this process. Tables B.14 to B.20 in appendix B.4 show additional details on the reasoning for this rating.

3.1.5 Selection

The final selection combines the previous scoring of available frameworks and OSs. Table 3.6 presents the three candidates with the highest reached score. The RODOS OS reaches a score of 0.95 and is the candidate with the highest cost-benefit score.

The second and third candidates (RTEMS and FreeRTOS) differ from RODOS mainly regarding criterion 8 (Message-passing) due to the unavailable distributed publisher-subscriber message passing in these OSs. If these become available in the future, an additional abstraction layer can provide the possibility to use the framework suggested within this thesis in combination with any of these OSs. To

[6] https://github.com/azure-rtos/threadx
[7] https://bsp.windriver.com/products
[8] https://www.windriver.com/de/products/vxworks
[9] https://www.uclinux.org
[10] Last internet archive snapshot in April 2021: https://web.archive.org/web/2021*/http://www.uclinux.org/index.html

Table 3.5: Summary of the rating of candidate operating systems. The final score for each OS calculates as the sum of the weighted score of the individual criteria. The weights shown in table 3.1 are used. The detailed analysis and reasons for the given scores are available in tables B.14 to B.20

Criterion	FreeRTOS	RODOS	RTEMS	ThreadX
Open-source (1)	1	1	0.75	0.5
Documentation (2)	1	1	1	1
Support (3)	0.9	1	1	0.8
Ports (4)	1	1	0.88	1
Porting (5)	1	1	1	0.75
Multi-threading (6)	0.92	1	1	1
Real-time (7)	0.75	0.75	0.75	0.75
Message-passing (8)	0.2	1	0.5	0.2
Final Score	0.83	0.95	0.87	0.79

focus on the core functionality of the suggested framework, we do not define this layer now. Instead, future developments can include this abstraction layer if the environment demands the use of a different baseline OS.

Table 3.6: Result of the three candidates with the highest score within the baseline framework/OS selection. For information on all other candidates see table 3.3 and table 3.5

Framework/OS	Score
RODOS	0.95
RTEMS	0.87
FreeRTOS	0.83

3.2 Physical Interconnection of Nodes

The physical interconnection of nodes is the second aspect analyzed prior to the design of the actual framework. This interconnection of nodes is critical for the simplicity of the proposed system. It changes the number of electrical interconnection required for each node, and thus is critical for the complexity of the resulting hardware setup. Additionally, a simple network potentially decreases the complexity of synchronization algorithms as it might reduce the effort required for routing traffic to reach a destination node.

Within this section, we will first select a network topology suitable for the proposed distributed on-board computing. Sections 3.2.1 to 3.2.3 present this comparison and selection. Afterward, section 3.2.4 compares available physical interface standards and selects the interface standard used for the proposed system.

3.2.1 Network Topology Constraints

A network topology represents the structure of the interconnection between nodes in a network [23]. A physical topology represents the physical interconnection of hardware components; a logical topology the relationship of communicating entities at a higher level and can be interpreted as virtual overlay network [200]. A link in a physical topology represents the wiring or similar connection between nodes [200]. Thus, the selected physical topology affects the required cabling and the complexity of future extensions of the hardware setup. Additionally, the used physical topology has an impact on the required complexity of higher layer protocols. This is especially important if the logical topology cannot be directly mapped into the physical topology.

To simplify the integration of the proposed system, we want to minimize the required cabling and reduce the amount of required interfaces per node. Additionally, it should be possible to disable any node of the network without disabling the entire network of nodes. The network shall be able to function when any node is or suddenly becomes unavailable. This is required for later power saving modes where some parts of the system are disabled to reduce the overall power consumption.

Distributed applications require the synchronization of all (or subsets of all) nodes. A simple broadcast (or multicast) mechanism reduces the complexity of this synchronization. Additionally, the publisher-subscriber provided by the previously selected RODOS needs a way to potentially send a message to all nodes of the system. Providing a simple broadcast or multicast on the lower layers of the network stack can thus reduce the complexity of the overall system. Especially the necessary packet routing on a higher network layer, if not all nodes can be directly reached, increases the complexity. Therefore, we prefer a physical topology that enables simple broadcast or multicast communication.

3.2.2 Candidate Topologies

We consider the five basic network topologies bus, star, ring, mesh, and tree as candidates for the suggested system.

Bus Topology

In a bus topology, all nodes share a trunk line for communication [23]. As this trunk line is a shared medium, only one node can send a message at a time [19]; all other nodes will receive this message [23]. As the bus topology does not require forwarding of messages, a failed node does not affect the overall network availability in general [19], although there are failure modes that could affect the entire network [185]. The trunk line is critical for this topology; a failure of the trunk line, e.g., if it is interrupted, will potentially propagate to the entire network [19]. The main benefit of the bus topology is the simplicity and the fact that it requires the least cabling effort of all the basic topologies [127]. Due to the possible collisions on the shared medium, the bus topology does not perform well under heavy traffic [127].

Star Topology

In a star topology, a central hub or switch connects all nodes of the network [23]. This central hub or switch controls the entire network and forwards data to the destination node [127]. In contrast to a bus topology, a failed cable will not affect the entire communication, but only a single node [23]. On the other hand the central hub or switch is a single point of failure for the entire network [127]. The star topology has a better performance than the bus topology, especially under heavy load [127]. This is achieved at the cost of a higher cabling effort [19] and a limited potential for expansion [127]. Broadcasting of messages in a star network is possible using a special address that will be forwarded by the central hub or switch to every other node on the network.

Ring Topology

Each node in a ring network receives messages from one neighbor and transmits or forwards messages to another neighbor in a ring like manner. Each message flows around this ring in the same direction until it reaches the destination node [200]. Such a network avoids the signal collisions impacting the performance of the bus topology [200]. Therefore, a ring topology has a better performance than a bus topology [127]. The deterministic transmission times in a ring network [23] could be beneficial for some real-time applications. The downside of a ring topology is the fact that a single failed node will interrupt the entire communication as messages cannot flow backwards [23]. Additionally, a ring network is in most cases more costly than a network using a star or bus topology [127]. Broadcast messages in a ring topology are possible by simply sending a message in a full circle.

Mesh Topology

In a full mesh, every node has a direct connection to all other nodes on the network [127]. As a full mesh is not practically possible in most cases, a partial mesh with only a subset of the links of the full mesh network is often used [200]. The main advantage of a mesh network is the increased reliability due to the redundant connections [23]. This redundancy assures that the failure of a single node does not affect other parts of the system [127]. This is achieved at the cost of additional cabling effort [19] and a higher effort required for routing messages to the destination node [23]. Broadcast or multicast messages are simple in a full mesh, but require additional routing effort in partial mesh topologies.

Tree Topology

In a tree topology, all nodes are connected hierarchically [200]. A tree consists of at least three levels, as otherwise it would be identical to a star topology [200]. A tree topology is thus a hierarchical extension of a star topology [23], which is mainly relevant for larger networks [200]. The hierarchical structure reduces the pressure on the interfaces of the central node [23] and thus simplifies expansion of the network [127] compared to a star topology. The cost for this simplified expansion is a communication overhead that further increases with the number of levels in the hierarchy [200]. Failures of nodes in higher hierarchies of a tree network affect the entire network similarly to the central node of a star topology and potentially break the entire internal communication [19]. Due to the hierarchical structure broadcast or multicast messages are possible without complicated routing mechanisms.

3.2.3 Topology Selection

A ring or tree topology does not allow disabling arbitrary nodes. This would be required for power saving modes and also represents one of the major failure modes of embedded MCUs. Although this could be mitigated by redundant connections, bypasses, or similar means, the added complexity should be avoided. The mesh topology requires some sort of routing to reach every node and requires a lot of cabling. As this is not feasible or would add additional undesired constraints on the final setup, the mesh topology is also excluded. The remaining candidates are the star and bus topologies. Due to the reduced implementation effort of the bus topology (no central hub or switch needs to be implemented), it is the preferred topology for the framework suggested within this thesis. Table 3.7 summarizes this comparison.

Table 3.7: Comparison of network topologies based on: required cabling effort; possibility to disable nodes during runtime, which also includes failing nodes; and simplicity of implementing a broadcast or multicast communication on top of the suggested topology. N represents the number of nodes in a network.

Topology	Cabling Effort (Required Connections)	Disabling Nodes Possible	Simple Multicast
Bus	✓ $\mathcal{O}(1)$	✓	✓
Star	✓ $\mathcal{O}(N)$	✓ (other than central node)	✓
Ring	✓ $\mathcal{O}(N)$	✗	✓
Mesh	✗ up to $\mathcal{O}(N^2)$	✓	✗
Tree	✓ $\mathcal{O}(N)$	✗	✗

3.2.4 Interface Standard

For real hardware, a physical bus interface standard implements the selected bus topology. This interface standard must at least provide a specification of the electrical interface, the electrical signalling scheme, media access control, detection of unsuccessful communication, and recovery from such situations. Therefore, the interface standard should at least cover the physical and data link layers according to the

International Organization for Standardization (ISO)/Open Systems Interconnection (OSI) reference model [86].

All nodes in the selected network will eventually transmit data. The point in time of this transmission might not be known in advance. Therefore, we prefer a bus that allows every node to initiate a transmission.

Availability of hardware support for the selected bus in COTS MCUs is beneficial as it reduces development effort and guarantees better integration in arbitrary nodes. A minimal amount of required cables will also simplify prototyping and integration in a full system. Therefore, we prefer a bus that does not require a large amount of connections to operate.

We identified three multi-master bus standards widely available in embedded COTS and space grade MCUs.

CAN

CAN is defined in ISO standard 11898 [81–85] and targets distributed control in automotive applications [81]. It is a multi-master bus with priority based media access control [81]. A single differential signal provides the electrical interface between nodes [82]. The physical connection of nodes can be performed in a star- or bus-like layout [83]; in both cases every message is broadcasted to all nodes [81].

CAN is already used in a few CubeSats [24]. The European Cooperation for Space Standardization (ECSS) engineering standard ECSS-E-ST-50-15C [59] standardizes additions for the use of a CAN bus in spacecraft.

Classical CAN is defined for data rates of up to $1\,\mathrm{Mbit\,s^{-1}}$ [81]. A CAN frame uses 44 bit[11] for its header and checksum fields, not including bit-stuffing or inter-frame spacing; each classical CAN frame can carry up to 8 B of user payload [81]. Therefore, classical CAN has an overhead of about 50 % and thus has an actual user data rate of only about $500\,\mathrm{kbit\,s^{-1}}$. A flexible data rate version of CAN provides a higher signaling rate of up to $5\,\mathrm{Mbit\,s^{-1}}$ and increases the allowed size of the payload [81, 82]. CAN with flexible data rate is not yet common in most COTS and especially not in space grade MCUs, but if required, it might provide increased performance in future hardware revisions.

The CAN media access control layer uses carrier sense multiple access with collision avoidance; thus any node can start transmitting a frame if the bus is currently idle [81]. If multiple nodes start a transmission at the same time, the nodes perform a bit-wise arbitration [81]. Nodes will read back the transmitted data; if it is different from the expected value, the node lost the arbitration and will stop its transmission [81]. Therefore, the first bits of every frame act as arbitration field; as it contains the message identifier, these identifiers also define the message priorities [81].

CAN provides several layers of error detection and recovery from failure states. A transmitting node will detect if the bits were successfully applied to the bus and if it receives a proper acknowledgement [81]. Receiving nodes will check the correct bit-stuffing, the trailing checksum, or illegal use of the CAN bus in general [81]. Whenever any node detects an error, it immediately signals the error to all other nodes [81]. A fault confinement entity counts detected transmit and receive errors; if these counter surpass a threshold, the fault confinement entity will separate the node from active participation in the communication [81]. While CAN FD, an improved version of CAN, provides higher bit rates [82], low speed CAN with a signaling rate of up to $125\,\mathrm{kbit\,s^{-1}}$ includes additional guarantees for various off-nominal scenarios [83]. It is specified to still operate with any single line of the differential signal shorted to ground, the supply voltage, or the other line [83].

I²C

I²C is a communication standard to simplify interfacing components between different vendors [220]. It is designed for inter integrated circuit control of MCUs and their periphery [220]. I²C uses two single ended signals, a clock, and a data signal, and it allows multiple controllers on a single bus [220]. Not

[11] 64 bit with extended identifiers

all nodes necessarily support the multi-controller mode, but COTS and space grade components with explicit multi-controller support are available[12].

According to Bouwmeester, Langer, and Gill [24], many CubeSats use an I²C bus. Although the authors do not mention the specific use of I²C in those CubeSats, they state that at least one mission failure can be attributed to I²C and multiple others are likely [24]. Therefore, we deem the use of I²C questionable for mission critical applications.

While I²C is specified for up to $5\,\mathrm{Mbit\,s^{-1}}$ in ultra fast-mode [220], many MCUs only support standard and fast-mode with up to $400\,\mathrm{kbit\,s^{-1}}$. The overhead of I²C is small compared to CAN. Each transaction starts with a 7 bit device address and a read-write flag indicating the type of the transaction [220]. After each byte, a single-bit acknowledgement indicates successful reception of the preceding byte [220].

The I²C arbitration of concurrent transaction is similar to the CAN arbitration. A controller stops an ongoing transmission whenever it transmits a recessive bit to the data signal line but receives a dominant bit on that line [220]. This arbitration only uses the address and data of the transmission; no priority can be included in this process [220]. Some undefined conditions exist within this arbitration, which are mentioned but not resolved in the I²C specification [220].

A receiving node has to assert at acknowledgement after each byte of the transmission [220]. This flag indicates various potential errors, including unsupported or invalid commands, overflowing buffers, or the absence of the receiver [220]. The I²C standard [220] does not include a checksum or other means to assure data integrity. Bus lockups are a common issue in I²C buses [24]. The I²C standard [220] suggests bus clear operations by consecutively toggling the clock line to resolve these situations; if it is not effective, a hardware reset or power-cycle is suggested. Various different user level mechanisms try to solve these issues with I²C on CubeSats [24]. Even with those custom mechanisms, many CubeSats have problems regarding the reliability of I²C [24].

TIA/EIA-485

The Telecommunications Industry Association (TIA)/Electronic Industries Alliance (EIA) standard TIA/EIA-485-A [216], formerly called recommended standard (RS)-485, specifies an electrical interface for multipoint systems. TIA/EIA-485 describes the required capabilities of transmitter and receiver hardware on a differential bus [216]. This is limited to the signal levels and does not include physical arbitration, timing, or any data link layer protocol suggestions [216]. The standard suggests the use with a signaling rate of up to $10\,\mathrm{Mbit\,s^{-1}}$ but allows devices with different rates [216]. Some experiments on the ISS use a TIA/EIA multidrop bus [99].

TIA/EIA-485-A [216] does not suggest any media access nor any error correction or recovery mechanisms. Thus, it only describes the physical layer interface and leaves all other details up to the user.

Selection

TIA/EIA-485 cannot be used without a matching data link layer protocol and does not provide any arbitration mechanism. I²C would provide the required basic features, enables multiple controllers on a shared bus including an arbitration and basic error detection. The major disadvantage of I²C is the potential for bus lockups and other error states, which could lead to mission failure. CAN on the other hand provides a rich set of error detection and recovery features, can operate in a degraded state with a partially interrupted physical connection and provides inherent features to exclude failed nodes from any communication. It also includes message integrity checks and a priority based arbitration. Support for CAN is available in the previously selected baseline OS RODOS. Therefore, we select CAN as the preferred bus standard for the proposed system.

[12] For example, the STM32L4 MCU family or the space grade VA41620 support multi-controller I²C.

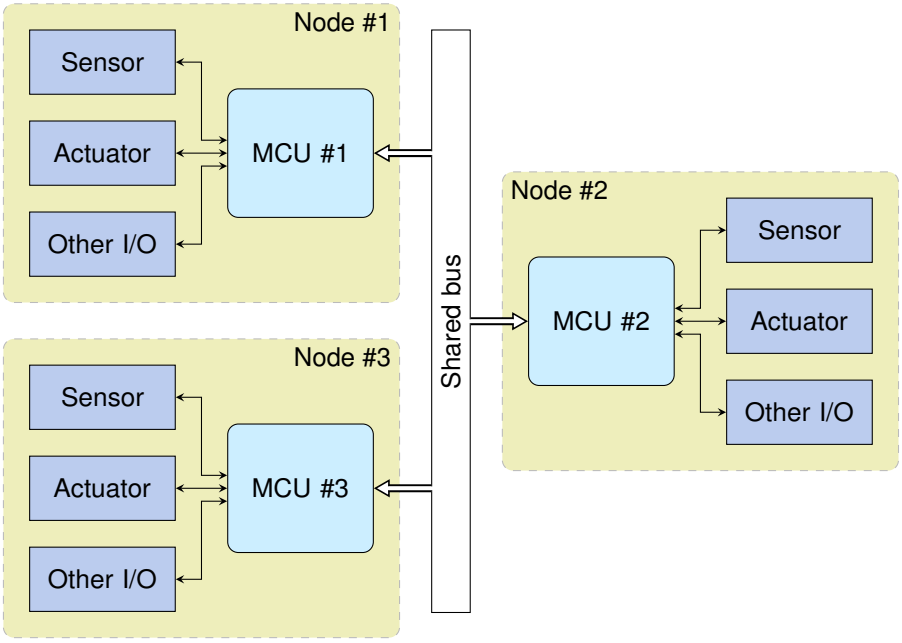


Figure 3.1: Basic DOSIS setup with three nodes. Each node contains an MCU and periphery. This periphery will mostly consist of sensors and actuators, but might also contain others like storage, external communication interfaces, or special purpose processing units.

3.3 DOSIS Framework Introduction

We propose the DOSIS framework as a light-weight framework that simplifies development of distributed applications for satellite control applications. It provides utilities to assemble a mission specific flight software as a combination of reusable components. Developers can pragmatically declare the interface of new components as a combination of pre-defined modules. The default behavior of the modules provides a basic implementation of the declared component. Developers can adapt the behavior of the component to its specific needs by providing a custom implementation for some modification points. Interfaces provide access to the implemented functionality in a distributed setup without deep knowledge about the internal message-passing or the networking capabilities of the individual nodes. The simplicity of the DOSIS framework encourages the use in single node as well as distributed setups.

We use a generic reference system to explain the general functionality of the DOSIS framework and the proposed overall system. The proposed system is a network oriented system, interconnected via a shared CAN bus. The CAN bus interconnects a number of smart nodes. Each node contains at least an MCU and optionally some periphery. Due to the simple porting of the used RODOS OS, a system designer can select different MCUs depending on the needs for a specific mission. The MCU provides the interface to a node's periphery like sensors, actuators, storage, co-processing units or external communication interfaces. Figure 3.1 visualizes such a setup of three generic nodes. Although this setup consists of three nodes, the DOSIS framework is not limited to this specific setup. A single node setup or setups with a higher node count are equally possible. Within a DOSIS setup, the distribution of software components to the available nodes is arbitrary. Only software components that provide access to specific periphery are bound to the respective node. All communication between different parts of the software is globally available within the overall network. This includes the availability of all information within the entire distributed system.

The main advantages of the DOSIS framework is its modularity and flexibility. It enables the development of a distributed application based on individual DOSIS *Components*. Each of these *Components* provides a distinct functionality to the application and can be reused within the same application or as part of a different mission. A *Component* itself consists of two tightly coupled, yet independent parts: its interface and its implementation. The interface, called *ComponentInterface*, defines how to interact with the *Component* and provides the functionality required to perform these interactions. This includes everything needed to interact with a *Component*, even though it might

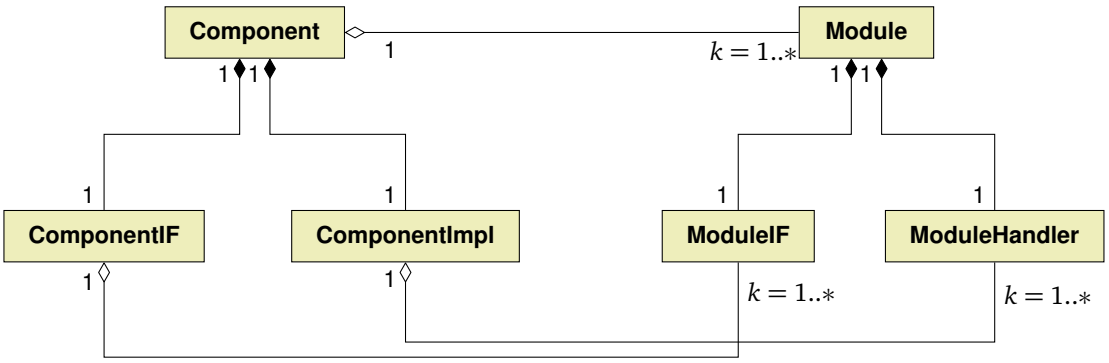


Figure 3.2: Generic associations between DOSIS *Components* and DOSIS *Modules*. Each *Component* consists of two parts: a *ComponentInterface* (ComponentIF) and a *ComponentImplementation* (ComponentImpl). One or more *Modules* define the general layout of a *Component*. The *ModuleInterface* (ModuleIF) and *ModuleHandler* of all of these *Modules* will be part of the *ComponentInterface* and *ComponentImplementation* respectively.

only exist on a different node. The implementation, called *ComponentImplementation*, is based on the definition of the interface. It provides a default functionality for all parts accessible from the *ComponentInterface*. A *Component* developer can change this behavior by adapting modification points within the *ComponentImplementation*.

The *Components* can be split into two groups: *Drivers* with direct hardware access and *Daemons* providing hardware independent logic. *Drivers* provide an interface to specific physical hardware. They are independent of any mission specific logic or data handling. *Daemons* on the other hand never directly access physical hardware. Instead, they access the data of sensors, target values of actuators, or other hardware capabilities via the corresponding *Driver*. *Daemons* are in charge of the *Drivers* they require, whereas a *Driver* will never actively change the behavior of a *Daemon*. *Daemons* are independent of the physical setup and distribution of periphery to nodes and only implement the higher level logic required for satellite control. Therefore, it is easily possible to move a *Daemon* to a different node without affecting the overall system. Thus, designers and developers can redistribute *Daemons* to different nodes, even at a late stage of the development process, to optimize resource utilization or adapt the system to a changed concept of operations. *Daemons* might contain mission specific logic that cannot be easily reused between different missions. In this case either the *DaemonImplementation* or the entire *Daemon* can be replaced. In the worst-case this affects other mission specific *Daemons*, but it will never affect any *Driver*. Section 3.4 presents the DOSIS *Components* in greater detail.

DOSIS *Modules* are the essential building blocks of DOSIS *Components* and further simplify the development. Each *Component* is a combination of one or more *Modules*. The *Modules* provide the individual building blocks of the *ComponentInterface* combined with a default implementation of the behavior of this part of the interface. Similar to *Components*, each *Module* consists of a *ModuleInterface* and a *ModuleHandler*. The *ModuleInterface* provides the required functionality to access the sub-part of the *Component* specified by this *Module*; the *ModuleHandler* provides the default implementation of the *Module*-specific functionality. *Component* developers can alter a *Module's* behavior by adapting the corresponding modification points within the *Component* itself. Figure 3.7 illustrates the basic relation between *Components* and *Modules*. Section 3.5 provides an in-detail presentation of the DOSIS *Modules*.

The DOSIS framework potentially increases the development speed, as a major part of the software can be reused over several missions. Additionally, it enables the parallel development of different components, even if they rely on the interface of one another. This is possible, as the declaration of a *Component* as a collection of *Modules* already provides a usable interface. It also enables and encourages early testing, as once the interface of a related *Component* exists, its implementation can be replaced by a test environment. This process can begin before the *ComponentImplementation* of the related *Component* is available and is transparent for the tested *Component*. The global availability of all communication further simplifies development and debugging tasks due to a simplified observation of a *Components* behavior. Therefore, the distributed setup not only enables later distribution of *Components* to nodes in a real system, but also supports testing and development processes. The development of

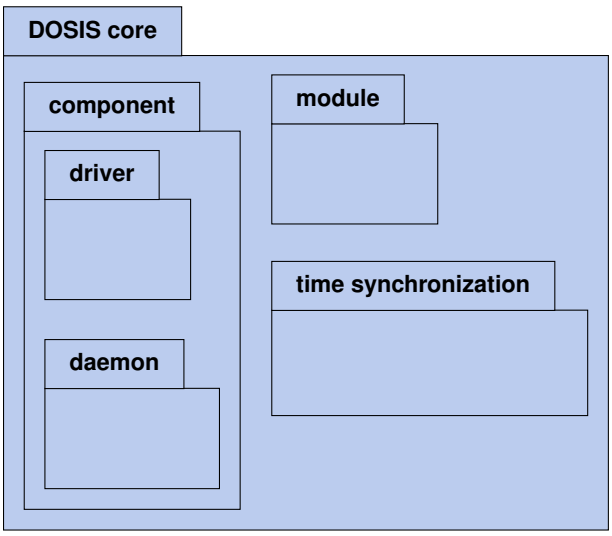


Figure 3.3: Top level packages of the DOSIS core framework.

individual components is therefore:

- independent of the node the component will be deployed to;
- independent of the development status of other *Components* it uses, especially the state of development of the *ComponentImplementation* of those *Components*;
- avoiding code duplication as *Modules* provide major parts of the otherwise replicated code.

The framework provides this simplicity for the developer by having a clean interface. It is not simply possible to exchange any information within the DOSIS framework other than by using the *ComponentInterfaces*. This reduces hidden information flow as much as possible and thus aids developers and designers with design and implementation of a minimal and yet expandable system.

Timing is critical for distributed control loops of real-time or otherwise time constrained systems such as an ADCS subsystem. Some *Modules* provide support for timed tasks (explained in more detail in section 3.5), but they require a precise time synchronization between the nodes of the network. To finally allow the use of the DOSIS framework to build such distributed control loops, a time synchronization mechanism is part of the DOSIS framework. Section 3.8 introduces this CAN and RODOS based time synchronization mechanism.

Figure 3.3 gives an overview of the various parts of the DOSIS core. The following sections will introduce the different parts of the DOSIS core. Section 3.4 provides a better insight into DOSIS *Components* and the available modification points for application developers. Afterward, section 3.5 introduces *Modules* as the basic building blocks of these *Components*. Finally, section 3.6 and section 3.7 provide more insight into the internal communication within the DOSIS framework and the orchestration and logical connection of multiple *Components*.

3.4 DOSIS Components

DOSIS *Components* are the largest basic building blocks within the DOSIS framework. Each component provides an independent and distinct feature or service to the overall system. To aid later distribution to specific MCUs, we separate *Components* into two categories: *Components* directly interfacing with specific hardware called *Drivers*; and *Components* independent of hardware such as controllers only relying on data provided by other *Components* called *Daemons*. DOSIS *Drivers* should be as minimal as possible and only provide access to hardware. It does not actively interface any other *Component* and does not have any dependencies besides the interfaced physical hardware. A DOSIS *Daemon* on the other hand performs high-level data processing and control of the *Drivers* and *Daemons* it interfaces with. In contrast to *Drivers*, *Daemons* never directly interface physical hardware, but instead access

the hardware indirectly via *Drivers*. This way *Drivers* can be easily reused in different missions while *Daemons* provide the potentially mission specific features. This increases the flexibility to adapt the overall software to the needs of yet unknown missions as demanded by design goal 2.

Each DOSIS *Driver* or *Daemon* consists of two parts: The first part is the interface, which acts as a contract for all external accesses and is called *DriverInterface* or *DaemonInterface* respectively. Each access to a functionality of a *Driver* or *Daemon* from outside uses this interface. It is independent of the node actually executing the *Driver's* or *Daemon's* internal logic and can be used multiple times to access the same *Driver* or *Daemon* from different locations. The second part is the implementation of the *Driver's* or *Daemon's* internal logic and is called *DriverImplementation* or *DaemonImplementation* respectively. In contrast to a *DaemonImplementation*, a *DriverImplementation* is not independent of the node it is running on as it requires access to the physical hardware.

Figure 3.4 visualizes the separation of DOSIS *Components* into *Drivers* and *Daemons* as well as the separation of interface and implementation.

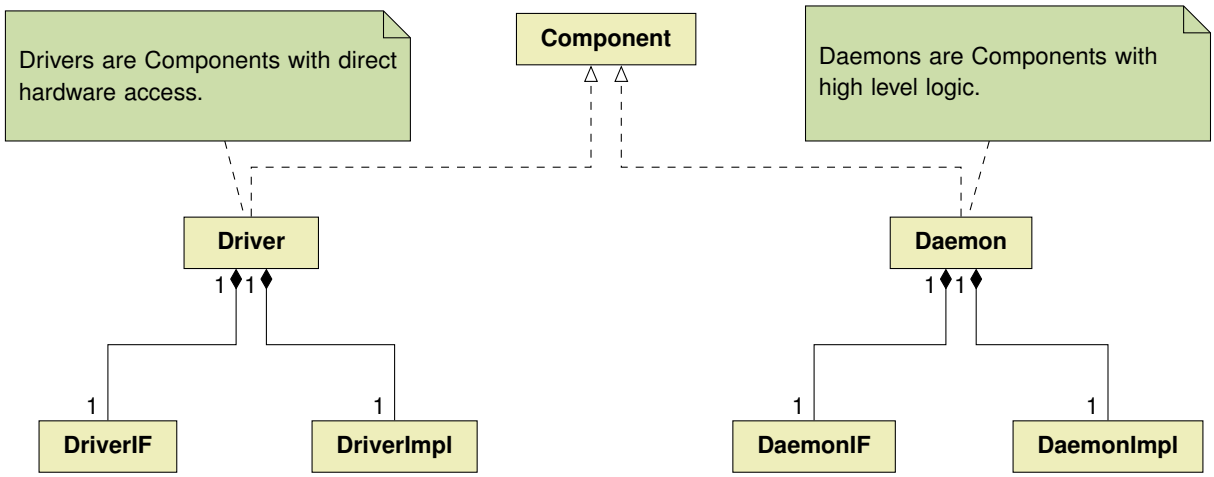


Figure 3.4: DOSIS *Components* split to *Driver* and *Daemon*. A *Driver* or *Daemon* itself consists of an interface (IF) and an implementation (Impl).

3.4.1 DOSIS ComponentInterfaces

ComponentInterface represent the contract of a *Component* towards its users. It is the single source of truth used to derive all messages to and from this *Component*. Additionally, it provides the high level interface to interact with the *Component* from an arbitrary node of the system. This section uses *ComponentInterface* to reference a *DriverInterface* or *DaemonInterface*. If a statement only relates to one of them, it uses the more specific *DriverInterface* or *DaemonInterface* explicitly.

A *Component* consists of a combination of one or more DOSIS *Modules*. DOSIS *Modules* provide generic functionality patterns for individual parts of a *Component*. To understand a *ComponentInterface*, these *Modules* can be imagined as registers with attached functionality and partly autonomous behavior. Section 3.5 introduces the DOSIS *Modules* in more detail. The *ComponentInterface* contains the part of these *Modules* responsible for the access to those *Modules* throughout the distributed system. Additionally, developers can add shortcut names or smaller helper functions to a *ComponentInterface*. This will at least contain human-readable names to all *Modules* within a *Component* and can include methods to further analyze or prepare the data when interacting with said *Modules*.

ComponentInterfaces provide parts of the message handling required for communication between the *ComponentInterface* and the *ComponentImplementation*. As a specific setup might use the *ComponentInterface* and *ComponentImplementation* on different nodes, the *ComponentInterface* has to assemble and send messages via a communication channel to the corresponding *ComponentImplementation*. At the same time, it provides access to the messages sent from the corresponding *ComponentImplementation* back to this *ComponentInterface*. Note that the *ComponentInterface* acts as a wrapper and only forwards these messages. The actual message handling is part of the *Modules*, which is introduced in section 3.5

in greater detail. A developer can thus use a *ComponentInterface* to access the functionality of the *Component* without knowledge about the internal messaging.

3.4.2 DOSIS ComponentImplementations

A DOSIS *ComponentImplementation* provides the functionality of the *Component*. It handles all messages received from an associated *ComponentInterface* and forwards them to the respective *Module's* implementations. Application developers can modify the behavior of the *ComponentImplementation* based on callbacks. Specifically, a developer can modify the following callbacks for any *ComponentImplementation*, i.e., *DriverImplementation* or *DaemonImplementation*:

System initialization (init) A modification point to execute setup functionality on initialization of the entire MCU. The default implementation does not perform any action.

***Module* data request (get)** Whenever data should be sent from the *ComponentImplementation* to the corresponding DOSIS interface, this callback provides the actual data. This callback can change the behavior of individual *Modules*. The default implementation uses the data of the associated *Module's* register.

***Module* data update (set)** Whenever data changes due to an update commanded via the *ComponentInterface* this callback is executed. Similar to the previous callback, it can be changed on a per-*Module* basis. The default action is storing the new data to the associated *Module's* register.

Additional modification points exist within *DaemonImplementations*, as *Daemons* implement higher logic and are often proactive *Components*. These *Daemons* repeatedly execute regular tasks that are not directly associated to any part accessible via the *DaemonInterface*. An example of such a regular task is the update of a controller. It regularly collects data from various *Drivers* using their respective *DriverInterfaces*, calculates a control output, and forwards the result to other *Drivers* that directly interface physical actuators. Two additional callbacks are available for this purpose:

***Daemon* initialization (*Daemon* init)** This callback provides a modification point for the behavior of the *Daemon* directly after system startup. This callback should set up the timing for subsequent invocations of the *Daemon* step callback. If a *Daemon* requires an initial configuration, the execution of this callback can be delayed until the *Daemon's Config Module* executed the first update of the entire configuration.

***Daemon* step (step)** A *Daemon* executes this callback in a regular interval. This interval can be changed at runtime and can thus adapt the behavior of the *Daemon* to different situations. Every proactive behavior of a *Daemon*, e.g., gathering data and command other *Components*, should be implemented within this callback.

Figure 3.5 visualizes the internal activities within a *Driver* or *Daemon*. The execution of a *Driver* or *Daemon* starts with an initialization phase. This includes calling the system initialization and for *Daemons* also the *Daemon* initialization callbacks. If required, a *Daemon* will wait for an initial configuration message and process this message within the *Config Module* and only execute the *Daemon* specific initialization callback afterward. After initialization, each *Driver* or *Daemon* will enter an infinite processing loop. This loop starts with the calculation of the time to the next internal activity. This activity is either an activity of one of the timed *Modules* (see section 3.5) or the next scheduled activation of the *Daemon* step callback. Afterward, the *Driver* or *Daemon* will wait for incoming messages and forwards those messages for handling within the respective *Module*. If the timeout passes before the *Component* receives any message, the scheduled activity will be executed. In both cases the execution will continue with the next loop iteration. Note that figure 3.5 does not display calls to the *Module* data request or *Module* data update callbacks. The *Modules* themselves activate these callbacks either as reaction to a received message or a time triggered activity. Section 3.5 presents the details of *Modules* and how they interact with these callbacks.

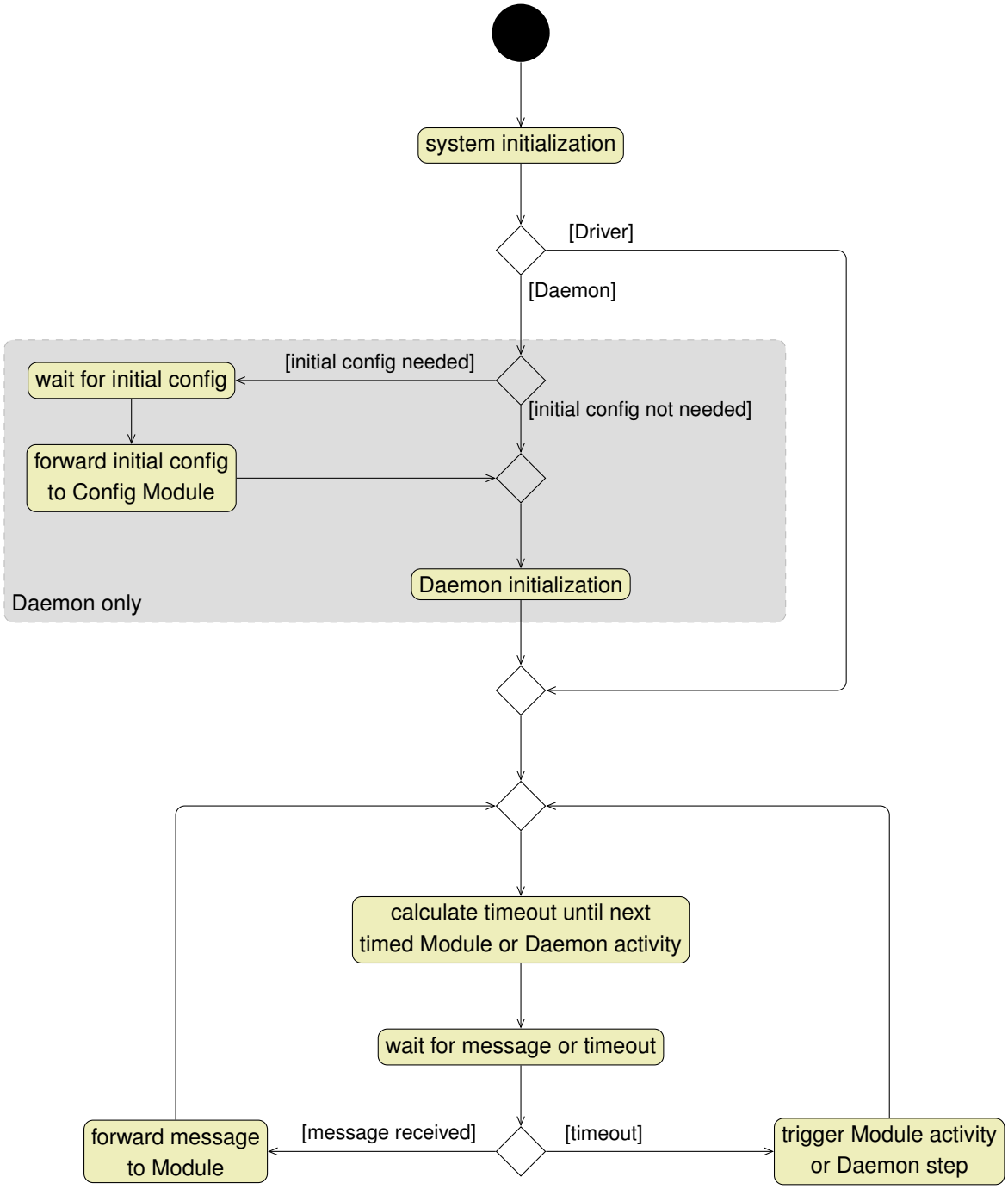


Figure 3.5: Internal activities of a *Driver*- or *DaemonImplementation*. It activates callbacks whenever their name is mentioned. This figure does not show callbacks for *Module* data request or *Module* data update, as the *Modules* call them from within the respective message or time triggered activity handling.

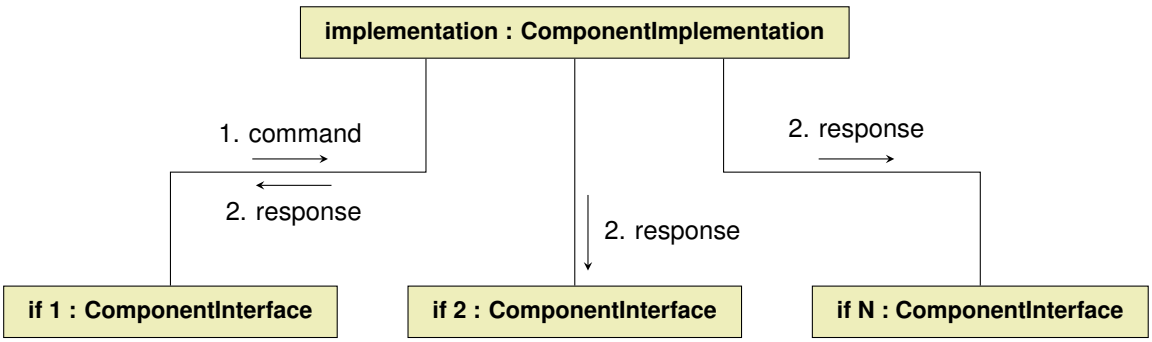


Figure 3.6: Communication diagram with multiple *ComponentInterfaces*. The *ComponentInterface* sends the response to all *ComponentInterfaces* at the same time.

3.4.3 General Interactions between Components

In a distributed application, multiple *Components* interact with one another to fulfill a certain task. A hierarchical organization of *Components* simplifies the interactions required to fulfill the task. *Drivers* are always at the lowest level of this hierarchy; they never command other *Components*. *Daemons* on the other hand command an arbitrary number of other *Components*. This hierarchy can have multiple layers, thus *Daemons* can command other *Daemons*. The *Daemons* are responsible for setup of all commanded *Components* as well as the recovery if one of them behaves off-nominal. Simple setups can use this simple hierarchy and thus only rely on 1:1 connections between *ComponentImplementations* and *ComponentInterfaces* where each *ComponentInterface* is part of another *DaemonImplementation*.

A *ComponentInterface* provides all necessary methods to access a certain *Component*. *Daemons* use this *ComponentInterface* to issue Commands and to access the returned data. The *ComponentInterface* always passively collects all messages from the associated *ComponentImplementation* and provides access to the various data points within the *ComponentImplementation*. Section 3.5.8 gives more details on how *Components* utilize *Modules* to handle the communication between a *ComponentInterface* and the corresponding *ComponentImplementation*.

3.4.4 Concurrent ComponentInterfaces

In most scenarios a simple 1:1 relation between *ComponentInterface* and *ComponentImplementation* is not sufficient. For example, multiple *Daemons* could access the measurements generated by a single *Driver* to perform their individual tasks. *ComponentImplementations* always send their messages to all associated *ComponentInterfaces*. The *ComponentInterfaces*, with aid of the used *Modules*, keeps track of the latest message received for each individual value within any of the *Modules* used to declare the *Component*. Section 3.5 provides a detailed description of *Modules* and the functionality they provide to access this data.

As long as only one *ComponentInterface* is used for active commanding of the *Component*, no conflicts can arise. In cases where this strict commanding hierarchy is not possible, the system designer has to take special care to avoid situations where different users execute conflicting command sequences at the same time. Although the *ComponentInterface* will always keep track of the latest known state of the *ComponentImplementation*, and thus simplify implementation of multiple commanding entities, it cannot assure that different control strategies are not contradicting each other. This 1:N relation of the commanding hierarchy can be used to add redundancy to a given system. Multiple instances of a commanding entity can co-exist. Just note that no direct mapping of response to commands is possible.

Figure 3.6 depicts a setup with multiple *ComponentInterfaces* for a single *ComponentImplementation*. The number of *ComponentInterfaces* is arbitrary and can change over time. As previously noted, each *ComponentInterface* receives a response sent from the *ComponentImplementation*. This response is entirely independent of the origin of the command or internal trigger that caused this response. In contrast to the response, the *ComponentInterfaces* do not receive the commands of other *ComponentInterfaces*. Thus, they can only observe the response of the *ComponentImplementation*.

3.4.5 Concurrent ComponentImplementations

Currently, the DOSIS framework does not directly support multiple *ComponentImplementation* instances connected to a single (or multiple common) *ComponentInterfaces*. In such a scenario the *ComponentInterfaces* can not keep track of the potentially different states of these *ComponentImplementations*, but instead will always use the latest response received from any of the *ComponentImplementations*. Currently, this needs to be solved on a higher level. Either the different implementations have to internally coordinate their response and ever only send a single response; or they behave like individual different *ComponentImplementations*, each connected to a different *ComponentInterface*. In the second case, another *Component* can use all of those *ComponentInterfaces* and perform the combination of the different answers manually. This would especially support high reliability scenarios, where redundant execution is necessary to avoid any kind of service interruption even for short periods of time. As most CubeSats, especially the MOVE-III and ORIGINS LRSM satellites, do not have such strict requirements, these features are not part of the core DOSIS framework. Section 3.9.6 presents additional information on redundant execution of certain tasks within the DOSIS framework.

3.5 DOSIS Modules

DOSIS *Modules* are the smallest building blocks within the DOSIS framework. They provide an abstract representation of individual elements of a *Component*. This includes a register to store the data associated with this *Module*, the abstract behavior of the *Module*, and the interface functionality to access and manipulate the stored data. *Modules* in combination with an identifier and a data type provide the basic building blocks for all *Components* (see section 3.4). Each *Module* can be separated into two parts: the interface to access and manipulate the *Module* within the context of a *ComponentInterface* (see section 3.4) called *ModuleInterface*; and the action handler providing the default actions, modification points for developers, and data storage called *ModuleHandler*. The *ModuleInterface* provides synchronous and asynchronous access to the corresponding *ModuleHandler*, i.e., it can either request data and wait for the response or request the data and access the response at a later point in time. The *ModuleHandler* provides methods to send a response back to the *ModuleInterface*. It also provides the default behavior with modification points to adapt the *Module* to the specific needs of a *Component*. The modification points make use of the callback functions within a *ComponentImplementation*, which are presented in section 3.4 in greater detail. Figure 3.7 visualizes this separation based on a *Settable Module*.

The *Settable Module* is one of seven DOSIS *Modules* with distinct features. The other *Modules* are *ReadOnly*, *Interval*, *TimedSettable*, *Actuator*, *Doable*, and *Config*. Figure 3.8 depicts these *Modules* and their relation to *Components*. Not shown are the identifying key (`Key`) and data type (`Type`) template arguments depicted in figure 3.7. The key acts as unique identifier of a *Module* within the context of a *Component*. *Modules* used to declare a certain *Component* can still be the same general type of *Module* and also use the same data type, but must use a different, unique key. The only *Module* that cannot be used arbitrarily is the *Config Module*. It has a special role and is only available as a mandatory *Module* of every *Daemon*.

The following sections will introduce the seven *Module* types and provide insight into their intended use, behavior of each *Module*, and available modification points. Finally, section 3.5.8 presents the interaction of *Modules* and *Components*.

3.5.1 ReadOnly

ReadOnly is the simplest DOSIS *Module*. It provides an internal register that cannot be modified via the *ReadOnlyInterface*. The *ReadOnly Module* is intended for data that is only sporadically required. An example use case is a serial number of a sensor controlled by a DOSIS *Driver*.

The *ReadOnlyInterface* provides methods to request the data stored in the internal register and to access the received response. A *ReadOnlyInterface* always keeps track of the latest received response, thus keeping track of the latest state of the internal value. The access to the response is available in both

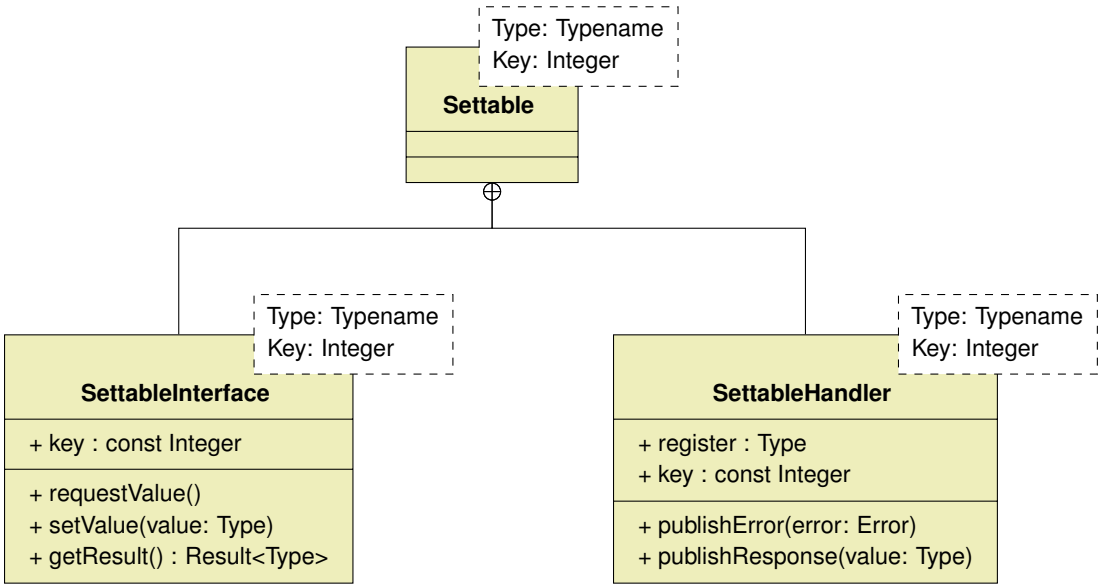


Figure 3.7: Split of the DOSIS *Settable* module into *SettableInterface* and *SettableHandler*. *Error* represents the type of error codes within the framework; *Result<Type>* represents the result sent from the *SettableHandler* to the *SettableInterface*. This result can either contain data or an error code. Depending on the specific module the available methods within the *ModuleInterface* vary. This diagram only displays the basic public methods. Additional blocking counterparts of the shown methods exist but are not shown in the diagram.

synchronous and asynchronous versions. If no new response is available, both versions provide access to the last known state of the internal value. A combined synchronous method to request and wait for the response provides an even simple user interface for basic use cases. All synchronous methods use an optional timeout parameter to avoid infinite blocking due to an unresponsive *Component*. Additionally, the *ReadOnlyInterface* provides access to the identifying key.

The *ReadOnlyHandler* provides direct access to the internal register for use within the *ComponentImplementation*. Methods to send data or error messages to the associated *ReadOnlyInterface* provide the capability to sporadically generate additional messages, e.g., on update of the value due to an external event. Similar to the *ReadOnlyInterface*, the *ReadOnlyHandler* provides access to the identifying key of the specific *ReadOnly Module*.

By default, the *ReadOnlyHandler* returns the content of the internal register whenever it receives a request. An implementation of the get callback for the key of this *ReadOnly Module* changes this default behavior. The return value of this implementation of the get callback replaces the value stored within the internal register and is forwarded to the *ReadOnlyInterface* instead.

3.5.2 Settable

A *Settable Module* is similar to a *ReadOnly*, but additionally allows setting the internal register of the *SettableHandler*. The *Settable Module* provides the required functionality for values that can be modified, but will not be accessed in a repetitive pattern or time critical sequence of commands. An example for such a use would be a calibration parameter of a sensor controlled by a DOSIS *Driver*.

The *SettableInterface* provides the same functionality as the *ReadOnlyInterface*. Additionally, it provides methods to set the content of the *ReadOnlyHandler's* register. A combined synchronous method to update the content of the register and verify the returned result enables simplified access patterns. The method expects the returned value to be identical to the value used for the call to the set method. If these values are not identical, an error will be returned instead.

The *SettableHandler's* interface for use within the *ComponentImplementation* is identical to the *ReadOnlyHandler's* interface.

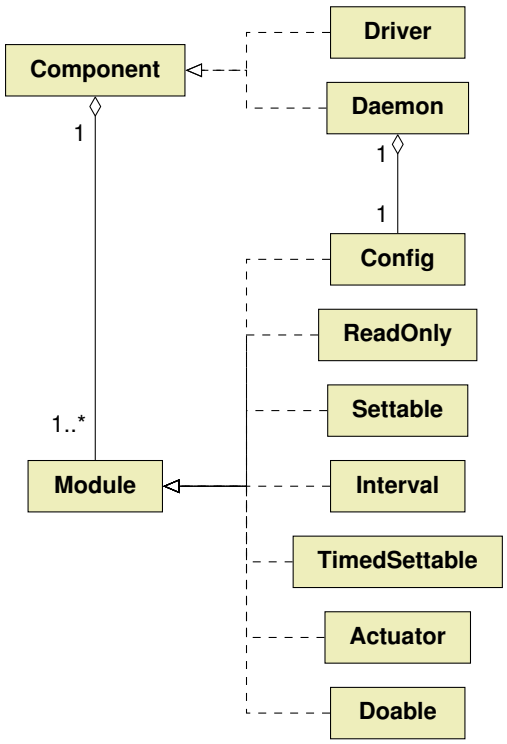


Figure 3.8: Relation between DOSIS *Components* and *Modules*. Each *Component* consists of one or more arbitrary *Modules*. The *Config Module* is a mandatory part of every *Daemon* and cannot be used in other contexts.

The default action for a received set command is an update of the internal register. The value will be overwritten and the *SettableHandler* returns the new content of the register to the *SettableInterface*. Similar to a *ReadOnly*, the *SettableHandler* returns the content of the internal register for every data request as its default behavior. Additionally, a modification point for updates to the internal register exists, which is similar to the get callback as mentioned for the *ReadOnly*. The corresponding set callback will be called whenever the *ReadOnlyHandler* receives a command to set the value of the internal register. This callback should post-process or forward the updated value to any hardware if it is required for a certain *Component*. If the update of the value fails, it can return an error code, which will be forwarded to the *ReadOnlyInterface*.

Figure 3.7 depicts the public interface of *SettableInterface* and *SettableHandler*.

3.5.3 Interval

The *Interval Module* is an extension of the *ReadOnly Module* and can provide its data automatically in a regular interval without an explicit request. The interval itself is changeable at runtime and thus can adapt its behavior to different scenarios throughout a mission. The *Interval Module* is the best fit for most sensor readouts.

The *IntervalInterface* provides all the methods of a *ReadOnlyInterface*. This includes the automatic update of the latest received value of the internal register within the *ReadOnlyInterface*. A user of the *IntervalInterface* can access the data provided in a regular interval using the same methods used to access the response after a request within a *ReadOnly Module*. Additionally, an *IntervalInterface* provides methods to access and modify the interval parameter. These methods are similar to the access methods of a *SettableInterface*, but operate on the interval parameter instead of the register. The *IntervalInterface* automatically keeps track of the latest state of this parameter used within the *IntervalHandler*.

The *IntervalHandler* is almost identical to a *ReadOnly* handler, but it provides additional methods to modify the configured interval from within the associated *Component*.

The default behavior of data requests is similar to the default behavior or a *ReadOnly Module*.

Additionally, it will generate a response in the configured regular interval. This mechanism triggers the same internal behavior as a received request and generates an identical response message. Similarly to the *ReadOnly Module*, the default behavior can be changed with a get callback. This callback will be activated for regular requests and as a consequence of the interval activation.

3.5.4 TimedSettable

A *TimedSettable Module* behaves like a *Settable Module* with delayed execution of a set command. Therefore, each set command contains an additional parameter specifying the point in time for actuation. This *Module* is intended for time critical activation that keeps the new state until commanded otherwise. Example use cases are a time critical configuration change or an actuation of a trigger at a certain point in time.

The *TimedSettableInterface* is a *SettableInterface* with an additional point in time parameter for every set method. Note that in contrast to a regular *Settable* the response will be delayed to the specified point in time. Any user of the *TimedSettableInterface* must take this into account, especially when using synchronous set methods. Similar to the previous *Modules*, the *TimedSettableInterface* keeps track of the latest state of the internal register.

The methods provided by the *TimedSettableHandler* are identical to the methods of a *SettableHandler*.

By default, the *TimedSettable* behaves like a *Settable* that delays the actual internal set of the register and transmission of the response. The available callbacks and their functionalities are identical to those of a *Settable*. The only difference is that the set callback is only called once the specified point in time is reached.

3.5.5 Actuator

An *Actuator* is similar to a *TimedSettable*, but features an additional default value and a reset timeout. In contrast to a *TimedSettable*, an *Actuator* will automatically reset its value to the specified default after the specified timeout is reached. An *Actuator* is intended for parameters that do not permanently change their state, but return to a default value after a certain timeout. For example coils of a magnetorquer will most likely use an *Actuator*, as they will only actuate for a certain time. Afterward, they should always return to a switched-off state.

An *ActuatorInterface* is similar to a *TimedSettableInterface*, but provides additional methods to interact with the default value and timeout parameters. It contains access to the default value and timeout parameters similar to the access of the *IntervalInterface* to the interval parameter. Again, the interface keeps track of the state of these parameters and always allows access to the latest known value without requesting the values again.

An *ActuatorHandler* is also similar to the *TimedSettableHandler*. Additionally, access to the default value and timeout parameters exists.

An *Actuator's* default behavior is similar to the behavior of a *TimedSettable*. The *Actuator* executes received set commands only once the specified point in time is reached. After the timeout has passed, the *Actuator* internally issues another set command using the specified default value. This will reset the internal register and send another data message to the associated *ActuatorInterface*. Again, the behavior can be modified with the get and set callback methods. Similar to a *ReadOnly*, the *ActuatorHandler* executes the get callback for every request. A set command on the other hand triggers two set callback invocations: the initial set of the register triggers the first invocation; the reset to the default value triggers the second invocation.

3.5.6 Doable

In contrast to all other *Modules*, a *Doable Module* has a return value with a potentially different type than the request value. Therefore, it does not behave like a smart register, but instead represents a

remote procedure call. Once triggered with a command, it will execute the internal logic and return the result. A *Doable Module* is intended for self-check and other functionality where a simple trigger generates a more complex output. Other scenarios where a small parameter triggers a large response will also benefit from a *Doable Module*. An example for such a scenario is a request of larger portions of data, e.g., configuration parameters from background memory. In this scenario the request parameter is a simple identifier, but the result is complex.

The *DoableInterface* provides a method to trigger the execution of the internal logic and methods to access the returned result. Synchronous and asynchronous versions are available and further simplify the use of the *Doable Module*. A *DoableInterface* stores the latest response and thus enables access to the response without triggering the execution again.

The methods available within the *DoableHandler* are identical to those of the *ReadOnlyHandler*. The *DoableHandler's* internal register stores the last received parameter.

Due to the different Types of the request and the result of a *Doable*, no default behavior is available. Therefore, the *DoableHandler* always executes a special version of the get callback with an additional parameter for the request parameter. This callback must return the result or error, which the *DoableHandler* forwards to the *DoableInterface*.

3.5.7 Config

The *Config Module* is the only *Module* developers of *Components* cannot use arbitrarily. Instead, it is implicitly part of every *Daemon*. It provides an interface to a *Daemon* capable of setting all modifiable parameters with a single command. This is used for *Daemons* that require an initial configuration. Designers may additionally use the *Config Module* for global state changes due to different mission phases or emergency situations. The *Config Module* simplifies those state transitions as only a single command has to be issued. This way, no configuration parameters can be missed and the configuration update of all *Modules* within the *Daemon* happens at the same time.

The *ConfigInterface* is similar to a *DoableInterface*, but has fixed data types for request and response. The request always contains a configuration for the specific *Daemon*. This response contains the identifier of the applied configuration. Similar to the *DoableInterface*, it always provides access to the latest received response.

No methods for direct use from within the *DaemonImplementation* exist. The *ConfigHandler* splits the received configuration message into individual configuration parameters of the *Modules* within the context of the current *Daemon*. Afterward, it calls each *Module* with the configuration update. This potentially triggers the set callback of the various modules, but will suppress the response these modules would normally produce. No modification points exist for a *Config Module*.

Configuration Update Content

A configuration update message contains all modifiable values of all other *Modules* within a *Daemon*. This contains not only the main value stored in the register of a *Module*, but also other configuration parameters like the interval parameter of *Interval Modules* or the default value and timeout parameters of an *Actuator Module*. As the *Config Module* immediately applies a received configuration update, the configuration message does not contain the point in time parameter used for set commands in *TimedSettable* and *Actuator Modules*. Additionally, each configuration update contains a user defined identifier. This identifier has no immediate functionality, but is returned to the interface on every configuration update, thus acting as an acknowledgement of success.

3.5.8 Interaction between Components and Modules

Components are collections of *Modules* and act as a container to access individual *Modules* and forward the data respectively. The *ComponentInterface* provides methods to access a *ModuleHandler* based on its key. Afterward, a user can directly interact with said *ModuleHandler* to send commands

to *ComponentImplementation*. Each command contains the key of the *Module*; the *ComponentImplementation* uses this key to identify the corresponding *ModuleHandler* and asynchronously forwards the content of the command. The *ModuleHandler* executes the necessary steps, including an invocation of the corresponding callbacks and sends a response back to the *ComponentInterface*. Similar to the *ComponentImplementation*, the *ComponentInterface* identifies the corresponding *ModuleInterface* based on the key contained in a response message and stores the response respectively. The user can access the response via the *ModuleInterface* he initially used to issue the command. If multiple *ComponentInterface* instances exist for a single *Component*, each of them receives the response message and stores it within the respective *ModuleInterface*. Figure 3.9 depicts this process based on a simple data request.

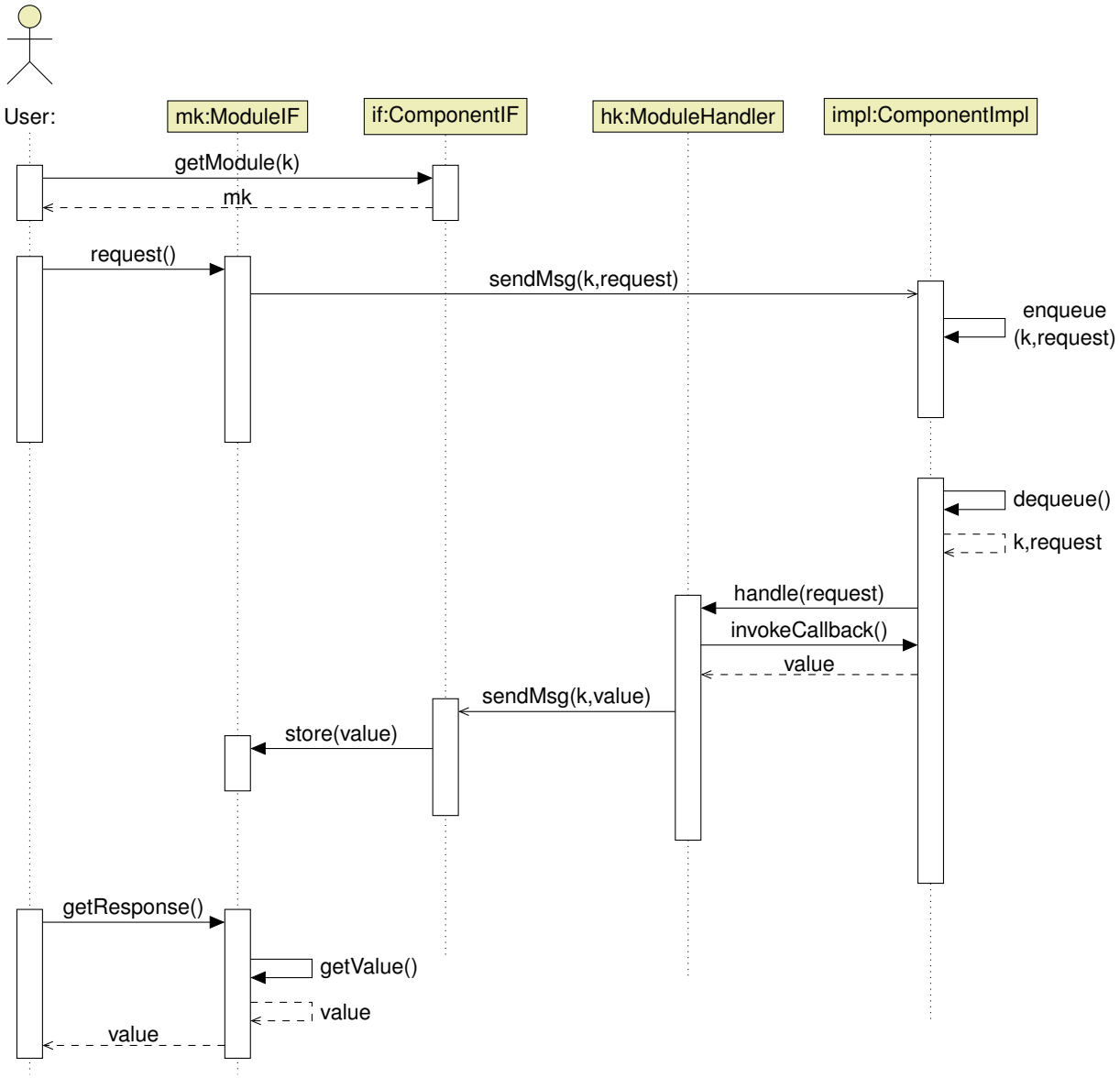


Figure 3.9: Interactions between *Components* and *Modules*. This figure depicts access to the data stored in a *Module* with read access, e.g., a *ReadOnly*. First the user gets access to the *Module* mk identified by its key k. Afterward, the user utilizes mk to send a request to the *ComponentImplementation* impl. Impl asynchronously handles the request and, after invoking the callback, sends the current value to the *ComponentInterface* if. As depicted in figure 3.5, the *ComponentImplementation* is waiting for incoming messages continuously, which is not shown in this sequence diagram. Finally, the user can access an available response via the *ModuleInterface*. If a request is not successful and the *ModuleHandler* returns an error, the *ModuleInterface* returns this error to the user instead. If no new response is available within the *ModuleInterface*, the *ModuleInterface* returns the latest known value or error.

3.6 DOSIS Communication Abstraction

The DOSIS communication abstraction separates the different concerns of the messaging between *ComponentInterface* and *ComponentImplementation* into layers of a communication stack. This enables transparent communication within a single node or an entire network of nodes while maintaining the possibility for future exchange, adaptation, or improvement of individual layers in the future. While this layering adds flexibility for future adaptation, it also provides the required capabilities for runtime changes of the distribution and/or availability of *Components*. Additionally, it enables many to many relations between *ComponentInterface* and *ComponentImplementation*.

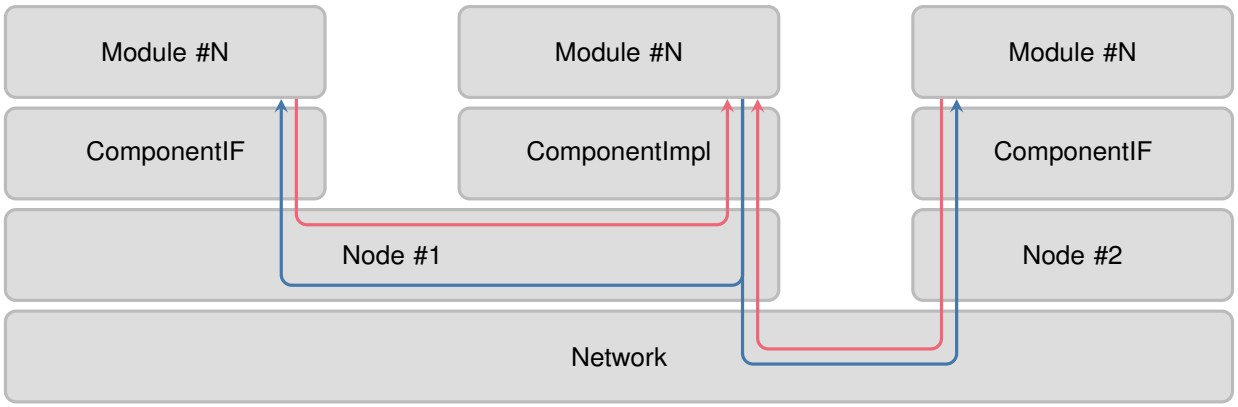


Figure 3.10: Layered communication within the DOSIS framework. A *ComponentImplementation* and the associated *ComponentInterfaces* can exchange messages independent of their distribution to nodes in the network.

The communication can be separated into three distinct levels with independent tasks: On the highest level, a message within a *ComponentInterface* or *ComponentImplementation* is forwarded to the respective *Module*. The main purpose on this level is the addressing of the *Module* in combination with some meta-information. The next level is responsible for addressing a specific *Component*. This level assures that a message for a specific *ComponentInterface* is forwarded to all instances of said *ComponentInterface*. While this level can directly forward a message to its local destinations, it relies on a third level to reach instances on other nodes. The third task is the forwarding of messages between nodes. The network level solves this task and provides all necessary means to reliably transfer a message to all other nodes. Figure 3.10 depicts the layered communication approach.

This separation of concerns enables transparent communication. The *Components* and *Modules* are independent of any knowledge about the node they are located on or the network connecting all nodes of the network. Additionally, this enables message-passing to all instances without knowledge about the amount of instances in a certain system or their distribution across several nodes and thus different locations in the network.

The CAN bus, which was previously selected in section 3.2, provides the lowest level's functionalities. This includes the physical and data link layers according to the ISO/OSI reference model [86]. Section 3.6.1 describes the CAN bus in general, its most important features, and the layout of a CAN data frame we will use for communication within the DOSIS framework. The RODOS message-passing provides addressing of individual *Components*. The publisher-subscriber message-passing in RODOS uses so-called topics to send a published message to an arbitrary number of subscribers. Section 3.6.2 presents this publisher-subscriber message-passing and its usage of the CAN bus to transparently attach to the communication from arbitrary nodes. Finally, section 3.6.3 presents the DOSIS internal messages used to address a certain module and its functionality in the context of a *Component*. This section provides details on the DOSIS message format and its usage of the RODOS topic messages. Figure 3.11 gives an overview over the used protocols on the four lowest layers according to the ISO/OSI reference model [86].

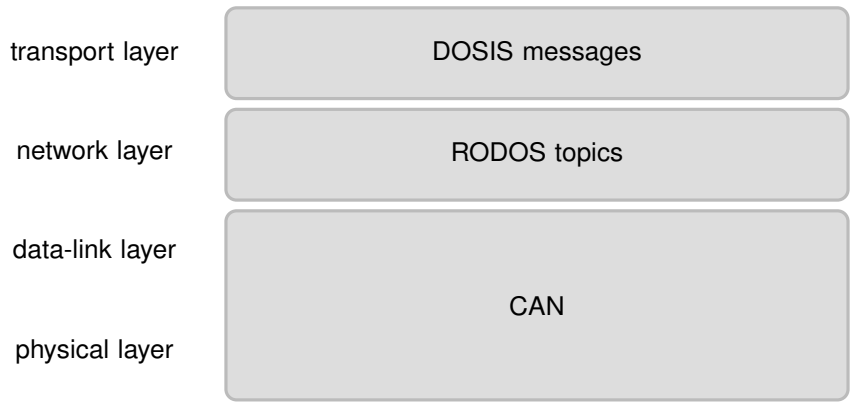


Figure 3.11: Network stack of DOSIS messaging using layer names according to the ISO/OSI reference model [86]. Only layers 1 (physical) to 4 (transport) are shown. Higher layers are not part of the DOSIS framework.

3.6.1 CAN

CAN as specified in ISO 11898-1:2015(E) [81] provides the physical and data link layer within the DOSIS communication abstraction. It provides the physical layer specification including media access with priority based arbitration, advanced error detection and recovery mechanisms, and includes message identifiers to filter incoming traffic [81]. We will use classical CAN data frames with extended identifiers at a CAN signalling rate of $1\,\mathrm{Mbit\,s^{-1}}$. Remote transmit requests will not be used, instead the DOSIS *Components* handle data requests on a higher layer.

One of the most important features of CAN is its guaranteed data consistency: Either all active nodes on the common bus correctly receive a certain message, or at least one node marks the message as erroneous and all nodes reject the message [81]. In combination with the automatic retransmission of unsuccessfully transmitted messages [81], we can rely on CAN to transmit any data until it is consistently received within all nodes. This check includes all nodes of the network, which independently verify a broadcasted message independent of the message's identifier; only afterward, a message can be dropped by an incoming message filter [81].

Each CAN frame starts with a single dominant symbol, called the start-of-frame marker, followed by the arbitration field [81]. This field contains the message identifier and is the only field used for message arbitration [81]. The subsequent control field contains the length of the following payload data in byte [81]. Classical CAN frames allow payload sizes of 0 B to 8 B [81]. A 15 bit cyclic redundancy check (CRC) calculated from all previously mentioned fields ensures data integrity [81]. As mentioned before, each node will verify the frame and transmit a corresponding acknowledgement [81]. An end-of-frame marker signals the end of the current frame [81]. A node may only transmit the next CAN data frame three bit times after said marker [81]. Figure 3.12 depicts a generic classical CAN data frame with extended identifiers.

Extended sequences of identical symbols in non-return-to-zero encoded data can lead to synchronization issues within receiving nodes. CAN mitigates this by adding stuff-bits after series of five consecutive identical symbols [81]. I.e., CAN inserts a recessive symbol after five consecutive dominant symbols and vice versa.

Dominant bits on the shared bus encode a logical 0, recessive bits a logical 1 [81]. As CAN uses a bit-wise arbitration and transmits the message identifier starting with its most significant bit, a lower message identifier has a higher priority and will win the arbitration process [81]. If the arbitration is still undecided after the last bit of the arbitration field and any of the subsequent bits of two concurrent transmission differ, the arbitration has failed [81]. An error frame will indicate such a failed arbitration [81].

Fault Confinement Entity

The CAN fault confinement is another mechanism to prevent faulty nodes from inhibiting communication [81]. In contrast to the mentioned CRC and message acknowledgement, which operate on a message basis, the fault confinement checks the behavior of a node. It provides separate counters for detected transmit and receive errors within a certain node [81]. Different errors cause an increment of the respective counter by a certain value; successfully transmitted or received frames cause a decrement of the respective counter [81]. If any of those counters surpasses a threshold, the fault confinement prohibits the node from sending error frames and increases the inter-frame-space the node has to wait prior to transmitting the next data frame to eleven bit-times [81]. The fault confinement entirely disconnects a node from active participation in the communication if the transmit error counter ever exceed a second even higher threshold [81]. Only an explicit restart or power cycle of the device reconnects a node after such a disconnect [81].

Optional CAN Extensions

A number of extension exist for further enhancements or adaptations of CAN to a specific domain. The low speed high reliability CAN as defined in ISO 11898-3:2006(E) [83] increases the reliability of the physical layer implementation. While it increases resilience to physical errors like shorts of one of the CAN wires to a certain voltage or ground level, it limits the usable data rate to $125\,\mathrm{kbit\,s^{-1}}$ [83]. A mission designer using the DOSIS framework can replace the suggested high speed CAN with ISO 11898-3:2006(E) compatible CAN without other changes to the framework.

Likewise, the ECSS standard ECSS-E-ST-50-15C [59] defines a physical layer with increased reliability. It not only adds a specification for off-nominal wiring situations, but additionally specifies the use of redundant CAN buses. While ISO 11898-3:2006(E) limits the maximum usable data rate to $125\,\mathrm{kbit\,s^{-1}}$, ECSS-E-ST-50-15C allows data rates of up to $1\,\mathrm{Mbit\,s^{-1}}$. Furthermore, ECSS-E-ST-50-15C includes higher level usage of CAN based on the CANopen [43] application layer. While the physical layer specifications and the redundant CAN bus suggested in ECSS-E-ST-50-15C can be used without any changes, the CANopen based extensions are not compatible to the DOSIS framework.

A time triggered version of CAN as specified in ISO 11898-4:2004(E) provides features that could be used for future extensions of the DOSIS framework. Although many parts of the DOSIS framework, especially the DOSIS *Modules*, provide various time triggered activities, they do not require a time triggered physical and data link layer.

3.6.2 RODOS Publisher-Subscriber Message-Passing

This section provides a short summary of the RODOS publisher-subscriber message-passing. The following is a summary of the information provided in [149], [148], and the public GitLab repository[13].

The RODOS middleware provides publisher-subscriber based message-passing. This message-passing enables asynchronous communication between different applications. It is implemented using so-called topics, which provide independent communication channels. An application can publish its data on a topic, which is received by any subscriber of the same topic. Each topic has a unique 32 bit identifier and an individual maximum length given by the used data type; the overall maximum length is limited to 1300 B. RODOS reserves the first 1000 topic identifiers for internal use. The number of active publishers and subscribers of a topic can change at any time. This enables flexible distribution of applications and redundant instantiation of applications. Gateways further expand this middleware over various nodes. Implementations for universal asynchronous receiver-transmitter (UART), CAN, User Datagram Protocol (UDP), and shared memory in multicore or multiprocessor systems are available.

[13] https://gitlab.com/rodos/rodos

RODOS Topic Messages over CAN

The DOSIS framework uses CAN gateways to connect the middleware of all nodes. The RODOS CAN gateway uses the classical extended CAN frame format with 29 bit identifiers as specified in [81]. The identifier contains a static prefix, a topic identifier, and a node identifier. The static prefix enables use of the CAN bus for messages other than RODOS topic messages; the node identifier has mainly informative character and avoids unsuccessful collision resolution of the CAN frames if multiple nodes concurrently publish messages on the same topic. Figure 3.13 visualizes the CAN identifier used for RODOS topic messages. The topic identifier of only 16 bit limits the usable topics over CAN to 65 536.

Each classical CAN frame can contain up to 8 B of payload. A simple fragmentation protocol splits the potentially larger RODOS topic messages into blocks fitting into individual CAN frames. Each frame contains a 1 B sequence number; the first frame contains an additional length field of 2 B; the remaining 5 B to 7 B contain the user data (see figure 3.14). Therefore, a message can be fragmented into up to 256 frames, with 5 B of user data in the first frame and 7 B in each subsequent frame.

A topic message on CAN could thus have a maximum size of $5\,\text{B} + 255 * 7\,\text{B} = 1790\,\text{B}$, which exceeds the global limit for RODOS topics of 1300 B. Therefore, any RODOS topic message can be transmitted via a CAN gateway. A gateway is capable of receiving interleaved messages from different nodes, but it will never transmit multiple interleaved topic messages. If multiple messages are to be transmitted by different nodes at the same time, the CAN arbitration prefers messages with lower topic identifiers.

The fragmentation of RODOS messages over CAN adds a significant communication overhead due to the small fragment size of only 8 B. This overhead is 300 % for a 1 B topic message, dropping below 20 % for 50 B and below 15 % for 433 B messages. This overhead is not critical for most applications but impacts the different time synchronization mechanisms presented in section 3.8.

Use of RODOS Topic Messages within DOSIS

RODOS topics provide the communication channel for messages between a *ComponentInterface* and a *ComponentImplementation*. Two separate topics exist for each instance of a *Component*: All *ComponentInterfaces* for said *Component* use one topic for messages towards the *ComponentImplementation* called command topic. The *ComponentImplementation* uses a second topic to send messages to all its *ComponentInterfaces*, called data topic. If multiple instance of a *Component* exist, each of them uses a different pair of topics for its internal communication. The topic identifier between 1001[14] and 65 535[15] are used for this communication. Therefore, a DOSIS based system can consist of up to 32 267 different *Component* instances.

3.6.3 DOSIS Messages

While RODOS topic identifiers address *ComponentInterfaces* and *ComponentImplementations*, the DOSIS messages are responsible for addressing individual *Modules* within a *Component*. For command topics, DOSIS messages mark commands as request or set commands; for data topics they differentiate between regular data messages and error messages. For *Modules* with more than one accessible value, i.e., *Modules* with additional configuration parameters, the DOSIS message also addresses the internal value of the *Module*. To avoid unnecessarily bloated messages, we aim to keep the DOSIS message overhead as minimal as possible without unnecessarily limiting the usability of DOSIS.

To fulfill its purpose, a DOSIS message must contain at least an identifier of the addressed *Module*, an inner identifier to address the sub-parts of a *Module*, and a message type. While the identifiers uniquely identify a register or configuration parameter of a specific *Module* within a DOSIS *Component*, the type identifies the commanded action for command messages, and differentiates data and error content for a data message. A DOSIS message does not require a length field. RODOS messages already provide such a length field; Furthermore, it is not required at all as the *Modules* know the message content data type based on the other fields of the DOSIS message header and therefore can derive the message length. The differentiation between command and data messages is also not contained

[14] 1001 is the first identifier not reserved for RODOS internal use.

[15] RODOS topic messages over CAN use 16 bit identifiers. The maximum value of an unsigned 16 bit integer is 65 535.

in a DOSIS message. It is not required as a RODOS topic only ever contains either command or data messages.

Module Identifier

The key to address *Modules* within *Components* as presented in section 3.5 is used as *Module* identifier. Most simple *Components*, such as sensors and actuators only require a few *Modules*. On the other hand a *Component* may also address an entire external subsystem, including a wide range of configuration parameters and telemetry data outputs. Therefore, the *Module* identifier's size of 8 bit is a compromise between reducing the size to avoid a bloated header and a reasonable limit to allow large *Components*. A size of 8 bit provides the capability to address $2^8 = 256$ different *Modules*.

Inner Identifier

The internal addressing of the data register or a configuration parameter of a specific *Module* requires another identifier. An *Actuator* has an internal data register and two configuration parameters, thus requires three different internal identifiers. Although this could be encoded in a 2 bit field, we will use a 3 bit field to account for future modules that may require additional parameters. This allows up to eight different registers or parameters within a single *Module* and thus provides sufficient capacity for future extensions.

Message Type

The last field of a DOSIS message is the message type identifier. The *Modules* as presented in section 3.5 require three different command types (get, set, and do). The corresponding data messages only require two different types, namely to distinguish data and error payloads. To represent the three different types of command messages, a DOSIS message contains a 2 bit type identifier.

Figure 3.15 depicts the layout of a DOSIS header.

DOSIS messages of any size only contain 2 B of header data. Although this generates a significant overhead for short messages, e.g., 25 % of overhead for 4 B of payload data, the relative overhead for larger payloads is quite low. A message with a payload of 100 B has a relative overhead of only 2 %.

Use in DOSIS *Modules* and *Components*

As shown in section 3.5, *Modules* are responsible for assembling and sending DOSIS messages. *ModuleHandlers* assemble and send data messages via the data topic; *ModuleInterfaces* assemble and send command messages via the command topic. *Components* never directly assemble or send a message. They only act as receiving end-point and forward the messages to the respective *Module*. Thus, *Modules* have full freedom regarding the content of the message. This assures that future extensions can add *Modules* with a new set of features without limitations regarding the internal messaging.

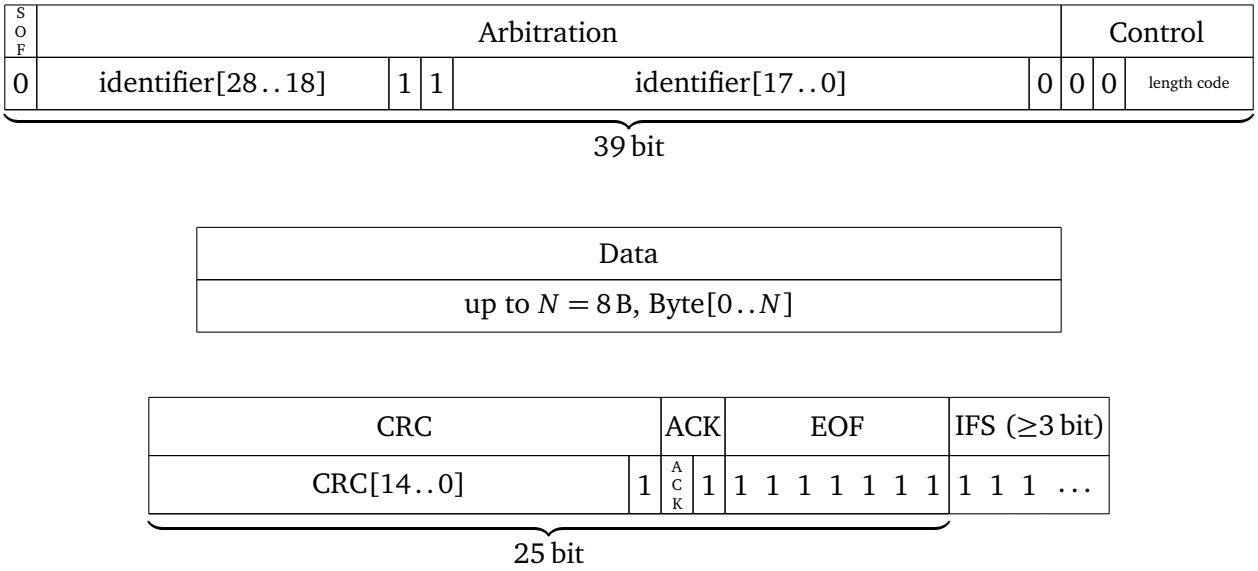


Figure 3.12: CAN data frame with extended identifier according to ISO 11898-1:2015(E) [81]. Each frame starts with a start-of-frame (SOF) marker, which is always set to 0, followed by the arbitration field containing the message identifier and some additional flags. This figure depicts the fixed values for data frames with extended identifiers; more information on these flags is available in [81]. The subsequent control field contains a length code indicating the length N of the following data field. The CRC field contains the calculated CRC-15 and a CRC delimiter, which is always set to 1. The acknowledgement field (ACK) contains a single ACK flag, which is set by receiving nodes, a value of 0 indicates successful transmission, and an ACK delimiter, which is always set to 1. Seven consecutive 1s mark the end-of-frame (EOF). Afterward, an inter-frame-space (IFS) of at least 3 bit is preserved prior to the next data frame. Some special frames, such as error frames, do not necessarily respect the IFS.

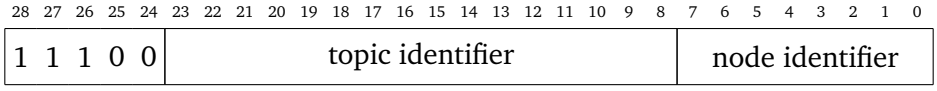


Figure 3.13: RODOS use of the CAN extended frame format's 28 bit identifier.

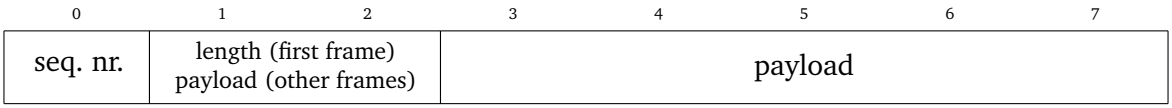


Figure 3.14: Fragmentation of RODOS topic messages over CAN. Shown is the allocation of the 8 B within a classical CAN frame.

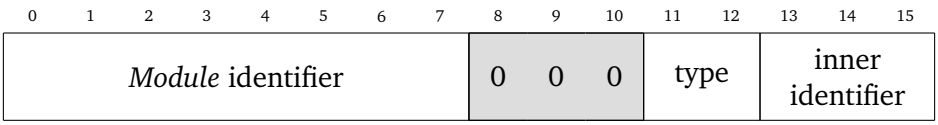


Figure 3.15: Header of DOSIS messages. A gray background indicates reserved bits always set to zero.

3.7 DOSIS Device Definitions

As previously mentioned, a *ComponentImplementation* and a *ComponentInterface* exchange messages via a pair of RODOS topics. The used topics must be unique within a certain system, thus the topic identifiers used for a certain *Component* must not be used elsewhere. Embedding these identifiers directly into a *Component* is not possible as multiple instances of a *Component* can exist at the same time. For example, multiple instances of a *Driver* for a sensor used multiple times in various locations of the system can exist. A device definition (*deviceDef*) uniquely assigns the required pair of topic identifiers to a specific *Component's* instances, i.e., the *ComponentImplementation* and *ComponentInterface* instances. Additionally, a human-readable name and the *Component's* specific type are part of the *deviceDef*. These support designers and developers while connecting *Components*, try to avoid wrongful connections, and enable human-readable output during development. A unique *deviceDef* must exist for each *Component* instance. It reflects the static connection and is therefore not changeable at runtime. Figure 3.16 visualizes the use of *deviceDef* as connecting entity between *ComponentImplementation* and *ComponentInterface*.

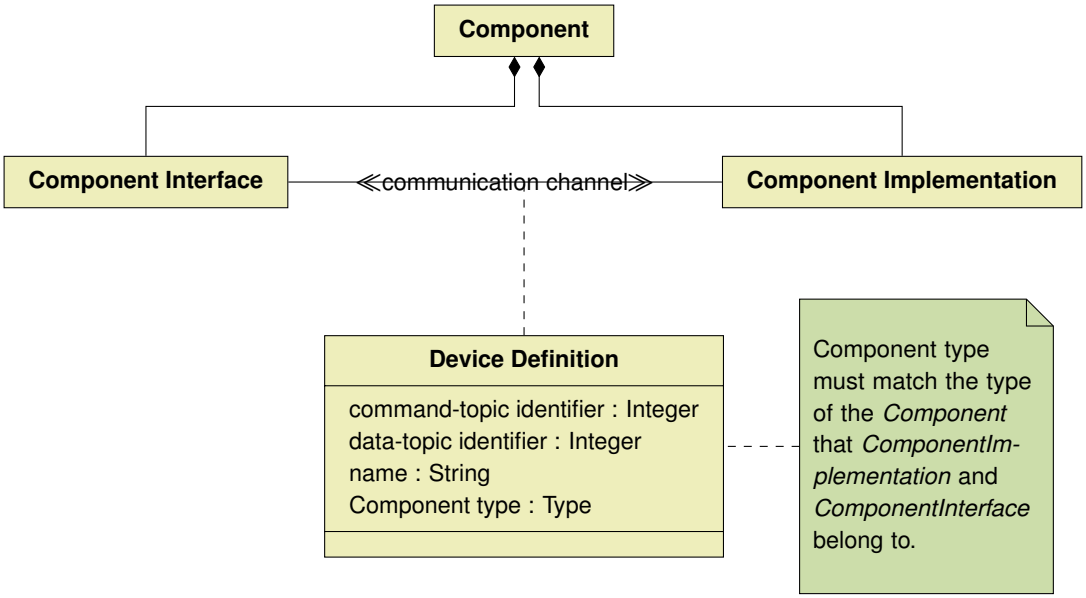


Figure 3.16: Usage of a *deviceDef*. A *deviceDef* provides all the details to establish a DOSIS communication channel consisting of a data topic and a command topic (see section 3.6). To avoid potential bugs, the *deviceDef* contains the type of the *Component* it can be used for. Using the *deviceDef* in a different context is forbidden.

3.8 Time Synchronization

To this point, we designed a framework that simplifies definition and implementation of *Components* and enables the assembly of distributed applications. *Modules*, which are the basic building blocks for *Components*, provide features supporting time sensitive applications. They enable developers to regularly read a sensors value, schedule update of actuator values to a specific time in the future and connect all those parts with some controlling *Daemon*. All of those features rely on a common understanding of system time, as otherwise scheduling of any event would not be possible. Thus, a common understanding of time must be established in all nodes.

To better understand the problem, we need to take a closer look at the timing requirements of different applications. The ADCS subsystem is the subsystem with the strictest timing requirements, as the accuracy of said system highly depends on precise measurement, forward propagation, and actuation of attached wheels, magnetorquers, or other active components. For a CubeSat ADCS, we assume a control loop that updates all parameters at a rate of 5 Hz. Such an update loop requires all

sensor values every 200 ms, calculates a control output parameter from these values, and updates the actuators accordingly. A precise knowledge about the exact sampling point and a regular sampling interval increases the accuracy of the overall ADCS system. To minimize the negative effect of timing uncertainties, a remaining time uncertainty between nodes of less than 2 ms is desired. To minimize effects on the estimated derivation of a measured value over time, the time should furthermore increase monotonically and reduce other effects that could negatively impact such a measurement.

Another important aspect is the accuracy of the local clock or oscillator in individual nodes. Tirado-Andrés and Araujo performed an analysis of the available clock sources for low-power COTS MCUs for wireless sensor networks [217]. They found that even temperature-compensated crystal oscillators have an accuracy of about 10 ppm and a drift of about 5 ppm [217]. Temperature-compensated crystal oscillators as commercially available for space applications show similar numbers in their respective specification[16]. Assuming a component with an accuracy and relative drift of less than ± 1 ppm could be found, two nodes could still drift apart more than the desired 2 ms in about 30 min[17]. Such a clock source obviously cannot meet the requirement for an ADCS system over extended periods of time. Additionally, other solutions like externally synchronized clocks as part of each node (e.g., GNSS conditioned clocks) are not feasible in a CubeSat due to their space and power requirements. Therefore, we have to rely on a local oscillator for short time intervals and provide a synchronization mechanism that assures long time synchronization of all nodes of the system.

A specific mission's needs drive the design of the time synchronization setup. Mainly due to the potentially heterogeneous system with different clock sources on each node, a single setup optimal for all scenarios does not exist. Therefore, we will have a look at a few options with reduced complexity that are still capable of meeting the requirement for a remaining time uncertainty of less than 2 ms. Related scenarios that also use a number of relatively simple nodes, e.g., wireless sensor networks, can be separated in two categories: time synchronization based on a reference clock used to synchronize all others, and time synchronization using an entirely decentralized approach. While a few decentralized solutions exist [66, 189, 199] that could be adapted to the DOSIS framework, synchronization to a reference clock gives better accuracy according to [209]. Additionally, the time distribution from such a reference clock can be tailored to the DOSIS framework and simplify the procedure due to the relatively simple CAN based network setup. Therefore, we suggest a simple clock synchronization based on a reference clock. The selection of the reference clock may be a dynamic process and enable redundancy even though the synchronization uses only one reference clock at a time.

The suggested time synchronization can be split into three independent sub parts: First we have to choose a reference clock selection procedure. Section 3.8.1 presents the advantages and disadvantages of static selection, dynamic selection and hybrid approaches. Afterward, we take a closer look at the time representation and the transfer of time updates from a reference clock to all other nodes. Sections 3.8.2 and 3.8.3 introduce the different methods and their advantages. Finally, section 3.8.4 presents the update of the local time on a receiving node.

3.8.1 Reference Node Selection

As previously stated, the suggested DOSIS time synchronization uses a reference clock attached to any node in the network. While the selected clock type is highly mission dependent, e.g., some missions might contain a GNSS based clock where others only contain some kind of local oscillator, the selection of the reference node can be decided on a higher level. The selection of a reference node can be performed statically or dynamically at runtime.

Static Selection

A statically selected reference node simplifies the implementation. Therefore, the designer has to carefully select the reference node based on system properties. This scenario fits best for scenarios

[16] e.g., Vectron: https://www.vectron.com/products/space/space.htm, Xsis: https://xsis.com/index.html, or Q-Tech: https://q-tech.com/products/tcxo-products/tcxo-for-space/

[17] Assuming the worst-case where both nodes are off by 1 ppm but in opposite direction; thus the total relative error is 2 ppm.

where only a single node has the required capabilities, i.e., a reasonably stable clock source. In CubeSats this could be a node with GNSS access or a real-time clock with alternate power source to ensure a reliable absolute time even after power cycles of the software. The main disadvantage of a static selection is the inherent single point of failure. The entire clock synchronization does not function anymore once the selected reference node fails. Thus, a purely static selection is only a reasonable option for single-string setups without redundancy.

Dynamic Selection

In contrast to the static selection, dynamic selection of the reference node is entirely independent of parameters that a designer assigned to the nodes. Instead, the nodes themselves dynamically select the reference node randomly or based on a simple heuristic. While this dynamic selection can always succeed, the selected reference node may not be optimal. For example, a node with a low-quality physical oscillator may become the reference node. This would inevitably lead to a system-wide clock with reduced quality. The absence of a single point of failure is the main advantage of an entirely dynamic selection. As long as at least one node is active, a reference clock can be selected. Thus, the fully dynamic selection based on a random decision or a simple heuristic might be a viable option for missions where the local clock of all nodes is of similar quality. Depending on the number of nodes in a certain system the entire dynamic selection may generate a not negligible complexity and a potentially unwanted communication overhead during the selection.

Hybrid Selection

Scenarios where only a few nodes use high-quality local oscillators need a different selection method. We suggest a hybrid of a static and a dynamic selection process for these scenarios. While the designer statically selects the potential reference nodes, a dynamic selection chooses the active reference node at a specific point in time. This combines the advantages of both selection processes. While the dynamic part still generates some overhead and complexity, it is limited to a few nodes. These nodes all have a high-quality clock, thus the static pre-selection excludes situations in which a low-quality clock may become the reference clock. The dynamic selection of one out of a few potential reference clocks avoids a single point of failure regarding the time synchronization.

Mission Specific Reference Node Selection

Depending on the specific mission's needs, any of the selection approaches may be optimal. While no choice fits all missions, we suggest using one or a few statically assigned potential reference nodes. In most CubeSats only a single node with a high-quality, e.g., GNSS based, clock reference exists. It may be accompanied by a real-time clock with alternate power source attached to either the same or a different node on the system. If such a backup is available, it should be used as possible fallback solution, otherwise only a single potential reference clock exists, thus static selection is the best option. If more than one reference node is available, the simplest dynamic selection possible should be used. This reduces the risk of potentially mission critical software bugs in this essential part. If multiple otherwise more or less equal nodes exist, we suggest selecting the reference node using a simplified bully leader election. Section 3.9.4 presents this leader election.

3.8.2 Time Format

The format used to represent time is important for the possible quality of system-wide time knowledge, overhead for time transfer between nodes (see section 3.8.3), and simplicity of local time update mechanisms on each node (see section 3.8.4). CCSDS standard CCSDS 301.0-B-4 [35] and the ECSS CAN bus extension standard [59] define various time formats. For simplified handling in all nodes, we also have to consider the time representation in RODOS. RODOS time is a local system uptime in ns [149] represented in a 64 bit signed integer[18]. For simplified calculation in later ADCS

[18] Extracted from the RODOS GitLab repository at https://gitlab.com/rodos/rodos.

implementation, we also consider the ease of converting the time representation to a modified Julian date [131], which is widely used for astronomical calculations.

Out of the four suggested time representations in CCSDS 301.0-B-4, only two representations are efficient for computer interpretation: CCSDS unsegmented time code and CCSDS day segmented time code [35]. The unsegmented time code represents a single monotonically increasing counter value uciteCCSDS-301. The day segment code represents a counter for full days, a second counter for ms of the current day, and an optional counter for fractions of the current ms [35]. The RODOS time, which is a monotonically increasing counter for ns, can be converted to a CCSDS unsegmented time code without any computational overhead. The conversion to and from a modified Julian date, which is a representation of fractional days [131], is best represented by a day segment code. We assume that more calculations will be done with the RODOS internal time, e.g., for scheduling purposes, compared to the modified Julian date representation. We use the RODOS time, which is similar to CCSDS unsegmented time code with a basic unit of ns, 8 B to represent the basic time unit, and no fractional time unit. This minimizes computational effort of time handling and reduces overall complexity.

Although CCSDS 301.0-B-4 suggests an epoch of January 1, 1958, and a representation in seconds according to the International Atomic Time (TAI), we only synchronize the local time as known to RODOS. This limitation is necessary as we cannot assume that every system using the DOSIS framework will include an externally synchronized clock source, such as a GNSS based clock reference.

The used 64 bit signed integer used to represent the system time can represent time up to

$$(2^{63}-1)\text{ns} \approx 9.22 \cdot 10^{18}\,\text{ns} \approx 292\,\text{y}. \tag{3.1}$$

This should be sufficient as uptime or mission elapsed time for most CubeSat missions with a duration of only a few years. Even using the suggested TAI epoch, this time format can represent time until the year 2250.

3.8.3 Time Transfer

As previously said, reference nodes will directly transfer their time to all other nodes of the network. To reduce complexity and utilization of the communication channel, we use a broadcast message to transfer these time update messages. Four options on different abstraction levels are available for such a broadcast: using DOSIS *Components*, using RODOS topic messages, directly using CAN frames, or using a dedicated clock distribution line.

Dedicated Clock Line

With a dedicated clock distribution network, the clock update broadcast uses a communication channel separated from all other communication. While dedicated clock pulses are simple to implement and enable precise timing, they cannot transfer the entire time information without an additional protocol or side-channel. While it promises the best time accuracy and precision, it requires additional wiring, which may be undesired in CubeSats. Therefore, we avoid the use of dedicated clock distribution networks and aim at clock distribution based on the available communication channel as presented in section 3.6.

CAN Based Time Transfer

A time transfer based on the existing CAN bus does not require any additional hardware setup and especially does not require any additional cabling. On the other hand, it requires special care regarding the implementation and the expected message delay. Knowledge about the message delay is critical for the quality of the time synchronization [130]. Thus, we will first have a closer look at the expected delay, followed by a presentation of the considered alternatives, and the final selection of the time transfer mechanism.

Message Delay Considerations[19] RODOS topic messages use extended CAN frames (see section 3.6). Including the inter frame spacing a full extended CAN frame has an overhead of 67 bit and a payload of up to 8 B. Bit-stuffing might additionally increase the size of a CAN frame by up to 29 bit [158].

As defined in section 3.6.2, we use a CAN data rate of $b_{\text{CAN bus}} = 1\,\text{Mbit}\,\text{s}^{-1}$. The transmission time of an extended CAN frame with 8 B of payload is thus:

$$T_{\text{CAN frame}} = \frac{S_{\text{CAN frame}}}{b_{\text{CAN bus}}} = \frac{67\,\text{bit} + 8\,\text{B} * 8\,\text{bit}\,\text{B}^{-1}}{1\,\text{Mbit}\,\text{s}^{-1}} = 131\,\mu s \tag{3.2}$$

$$T_{\text{stuff bits}} = \frac{S_{\text{stuff bits}}}{b_{\text{CAN bus}}} = \frac{29\,\text{bit}}{1\,\text{Mbit}\,\text{s}^{-1}} = 29\,\mu s \tag{3.3}$$

$$T_{\text{CAN frame+}} = T_{\text{CAN frame}} + T_{\text{stuff bits}} = 131\,\mu s + 29\,\mu s = 160\,\mu s \tag{3.4}$$

with

$T_{\text{CAN frame}}$	:	transmission time of an extended CAN frame excluding stuff bits,
$S_{\text{CAN frame}}$	:	size of an extended CAN frame without stuff bits,
$b_{\text{CAN bus}}$	:	data rate on the CAN bus,
$T_{\text{stuff bits}}$	:	transmission time of stuff bits in a CAN message,
$S_{\text{stuff bits}}$	:	size of stuff bits in a CAN message, and
$T_{\text{CAN frame+}}$	:	transmission time of an extended CAN frame including stuff bits.

Reduction of bit-stuffing jitter [37] and jitter less communication [38] are possible by specifically encoding the transmitted data. As this also increases the overhead, we will not make use of these mechanisms, although they should be noted if future extensions require a more precise time synchronization in DOSIS.

Time transfer messages have the highest priority, i.e., the lowest used CAN identifier, in our system. This guarantees predictable worst-case delays on the shared bus, as such a message will always dominate CAN arbitration. Thus, the worst-case wait time before transmission of a time transfer message starts is the time to finish the previous message. This wait time will always be less than $T_{\text{CAN frame+}} = 160\,\mu s$.

Messages sent via RODOS topics have a size of up to 1300 B. As introduced in section 3.6.2, the RODOS fragmentation protocol splits this payload into several CAN frames: the first frame contains 5 B, each subsequent frame contains 7 B of the topic message. Thus, RODOS splits a 1300 B topic message into

$$N_{\text{CAN frames}} = 1 + \frac{1300\,\text{B} - 5\,\text{B}}{7\,\text{B}} = 186 \tag{3.5}$$

extended CAN frames with 8 B of CAN payload each. The total transmission time of such a message with and without bit stuffing is

$$T_{1300\,\text{B RODOS message}} = N_{\text{CAN frames}} \cdot T_{\text{CAN frame}} = 131\,\mu s \cdot 186 \approx 24.4\,\text{ms} \tag{3.6}$$

$$T_{1300\,\text{B RODOS message+}} = N_{\text{CAN frames}} \cdot T_{\text{CAN frame+}} = 160\,\mu s \cdot 186 \approx 29.8\,\text{ms} \tag{3.7}$$

with

$T_{1300\,\text{B RODOS message}}$	:	transmission time of 1300 B RODOS message via CAN excluding stuff bits,
$N_{\text{CAN frames}}$	:	number of CAN frames of a 1300 B RODOS message,
$T_{\text{CAN frame}}$	:	transmission time of an extended CAN frame excluding stuff bits,
$T_{1300\,\text{B RODOS message+}}$	:	transmission time of 1300 B RODOS message via CAN including stuff bits, and
$T_{\text{CAN frame+}}$	:	transmission time of an extended CAN frame including stuff bits.

As RODOS topics do not interleave different messages from a single source, this has to be added to the worst-case delay. Additionally, this could be interrupted by higher priority messages on the CAN

[19] Partially based on own publication [184].

bus originating from other sources. In this case the maximum delay is infinitely long, and no worst-case guarantees can be given at all.

DOSIS *Component* based communication additionally increases this delay. As they handle received messages asynchronously and outside the regular topic message handling, other processes with equal or higher priority may delay the message handling.

DOSIS *Component* Based Time Transfer A time synchronization implemented based on DOSIS *Components* and their interfaces provides a high-level abstraction. An implementation would consist of a time synchronization provider *Component* that uses an *Interval* module to regularly transmit its own local time. All other nodes use a time synchronization receiver *Component* that utilizes the provider's *ComponentInterface* to access received time synchronization updates. While this abstraction simplifies the implementation, it has the largest uncertainties regarding the expected message delay. Thus, it is only an option for systems with reduced timing requirements. While it may achieve a reasonable quality in absence of other CAN bus load and higher priority tasks within the receiving nodes, it cannot give any worst-case guarantees.

RODOS Topic Based Time Transfer The direct use of RODOS topic messages is similar to the use of DOSIS *Components*. In contrast to DOSIS *Components*, it handles messages synchronously and thus processes received time transfer messages without additional delay. Therefore, it is not affected by other tasks running on a specific node. Nevertheless, RODOS messages cannot give any worst-case guarantees. If other messages originate from the same node as the time transfer messages, the delay may vary substantially.

Thus, we do not encourage the direct use of RODOS based messages for time transfer between nodes. Although it may decrease the delay variations compared to DOSIS *Components*, it still cannot give any guarantees and provides less abstraction and breaks uniform interfaces within the entire system.

Direct CAN Usage Based Time Transfer While the direct use of CAN frames provides the least abstraction, it enables timing guarantees for high priority messages. As previously mentioned, the expected delay for a CAN frame with the highest priority is less than $T_{\text{CAN frame}}$. The direct use of CAN removes all other delays due to RODOS not interleaving messages or asynchronous handling of DOSIS messages. Additionally, it enables the transmission of an entire 8 B time message as selected in section 3.8.2 in a single CAN frame. No additional overhead for fragmentation or inner addressing is required. Thus, a reference node completes transmission of a time message in $[T_{\text{CAN frame}}, 2 \cdot T_{\text{CAN frame+}}]$. Although this does not account for computational delays, i.e., the time between generating the time message and the actual transmission thereof as well as the time required to generate a timestamp after receiving the message, it gives a reasonable estimate for the remaining uncertainty.

Suggested Solution We suggest the direct use of CAN for time transfer. This solution provides the best knowledge about delays and thus the least uncertainty. Therefore, it promises the best time synchronization quality out of the previously mentioned solutions.

Tests of the different time transfer mechanisms and their effect on time synchronization quality confirm the superiority of the direct CAN based approach. Chapter 5 presents these tests and the results thereof in greater detail.

3.8.4 Local Time Update

At this point, all nodes know the time of the reference node at a certain point in time. The last step of the time synchronization is the update of the local time on each individual node. A local time update should be able to compensate the relative error of a local clock. The relative error can be separated into three individual error terms: a time difference between clocks called offset, a difference between the frequencies of the individual clocks called skew, and a difference in the change of the frequency

over time called drift [217]. Based on the clock definition of Tirado-Andrés and Araujo, an uncorrected clock C_i is defined relative to the reference time t as

$$C_i(t) = o_i + s_i \cdot t + d_i(t) \tag{3.8}$$

with

$C_i(t)$: output of clock i at time t,
t : reference time,
o_i : offset of clock i,
s_i : skew of clock i, and
$d_i(t)$: drift of clock i at time t.

Figure 3.17 visualizes these error terms. In the remaining part of this section, we will have a look at three basic update mechanisms and their characteristics.

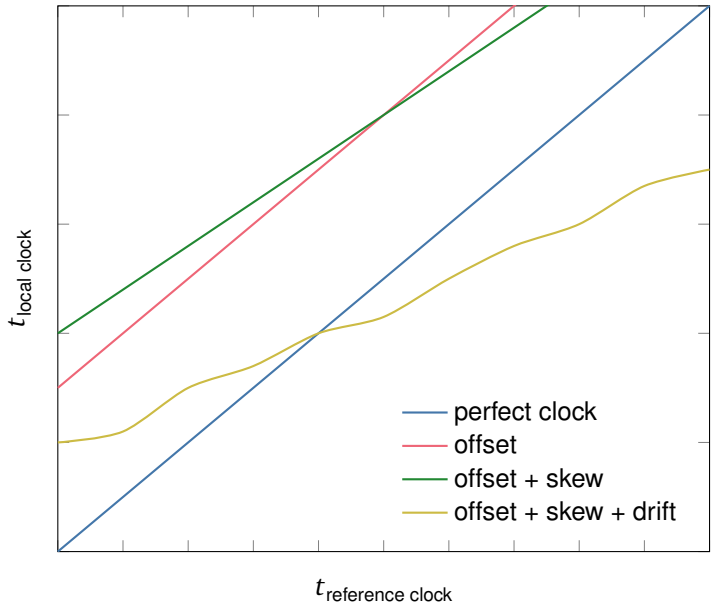


Figure 3.17: Visual representation of different clock errors for a clock model similar to Tirado-Andrés and Araujo [217].

Direct Set

The first update mechanism is a direct update of the current value of the local clock. This update mechanism modifies the offset of the local time after each received time update. We define a direct-set updated clock as

$$C_i(t) = \underbrace{o_i + s_i \cdot t + d_i(t)}_{\text{uncorrected local clock}} + \underbrace{c_i(k)}_{\text{correction term}} \tag{3.9}$$

with

$C_i(t)$: output of clock i at time t,
t : reference time,
o_i : offset of clock i,
s_i : skew of clock i,
$d_i(t)$: drift of clock i at time t,
$c_i(k)$: correction term of clock i in synchronization interval k, and
$k \in \mathbb{N}_0$: identifier of synchronization interval.

The recursive definition of $c_i(k)$ for this clock is

$$\begin{aligned} c_i(0) &= 0 \\ c_i(k+1) &= t_{\mathrm{rx},k} - \underbrace{(C_i(t_k) - c_i(k))}_{\text{local time at } t_k \text{ without correction}} \qquad \forall k \in \mathbb{N}_0 \end{aligned} \tag{3.10}$$

with

$k \in \mathbb{N}_0$: identifier of synchronization interval,
t_k : time of reception of time update message k, and
$t_{\mathrm{rx},k}$: received reference time in time update message k after delay compensation.

Due to the direct update, the only remaining error $\forall t \in \{t \mid \exists k \in \mathbb{N}_0 : t = t_k\}$ is the remaining uncertainty due to the unknown transmission delay as presented in section 3.8.3. The main advantage of this update mechanism is its simplicity and stability. The major disadvantage of this method are the jumps of the local time.

P-Controller

A simple proportional (P)-controller, steering the local clock towards the received time by adjusting the frequency of the local clock, avoids the jumps of the local time. The controller updates the clock skew for each interval between two consecutive time synchronization points. For each interval between two local clock updates, the last synchronization point and the local time associated with this synchronization point are the reference for forward propagation of the local clock. Thus, a P-controlled clock $C_i(t)$ is defined as

$$C_i(t) = \begin{cases} \overbrace{o_i + s_i \cdot t + d_i(t)}^{\text{uncorrected clock}}, & t \leq t_1 \\ \underbrace{C_i(t_k)}_{\substack{\text{local time at} \\ \text{start of interval}}} + \underbrace{(s_i + u_i(k))}_{\text{corrected skew}} \cdot \underbrace{(t - t_k)}_{\substack{\text{time since} \\ \text{start of interval}}} + d_i(t), & t > t_1 \end{cases} \tag{3.11}$$

with

$C_i(t)$: output of clock i at time t,
t : reference time,
o_i : offset of clock i,
s_i : skew of clock i,
$d_i(t)$: drift of clock i at time t,
k : identifier of synchronization interval,
t_k : time of reception of time update message k,
$u_i(k)$: skew correction term of clock i in synchronization interval k,
$k_t \in \mathbb{N}_0$: identifier of time synchronization interval at time t.

The skew correction term $u_i(k)$ is the output of a P-controller that uses the time deviation as input error term:

$$\begin{aligned} e_i(k) &= t_{\mathrm{rx},k} - C_i(t_k) \\ u_i(k) &= K_p \cdot e_i(k) \end{aligned} \tag{3.12}$$

with

$e_i(k)$: clock error at of clock i at the beginning of synchronization interval k,
$t_{\mathrm{rx},k}$: received reference time in time update message k after delay compensation, and
K_p : proportional gain koefficient.

The main advantage of such a P-controlled clock compared to the direct-set update is the absence of jumps in local time. Although the P-controlled clock update can compensate arbitrary clock offsets,

it cannot entirely compensate the clock skew. A static offset, a common problem with P-controllers, also affects this clock update mechanism. While a clock with only an offset error can be successfully compensated[20], a clock with offset and skew errors will converge towards ${}^{1-s_i}/_{K_p}$[21].

Large offsets may potentially lead to situations where a P-controlled clock generates decreasing output performance. To avoid these situations, K_p has to be selected carefully; depending on the scenario $u_i(k)$ must be additionally limited to never output a control parameter less than or equal to -1. Values of -1 or below could lead to a stopped or backwards running clock with unexpected side effects.

PI-Controller

Adding an integral (I) component to the previously presented P-controller compensates the remaining error in $e_i(k)$ [120, p. 412 ff.]. The only difference of the resulting proportional integral (PI)-controller compared to the P-controller based clock update is a modified definition of $u_i(k)$

$$u_i(k) = K_p \cdot e_i(k) + K_i \cdot \sum_{n=1}^{k} e_i(n) \tag{3.13}$$

with

$u_i(k)$: skew correction term of clock i in synchronization interval k,
k : identifier of synchronization interval,
K_p : proportional gain koefficient,
$e_i(k)$: clock error at of clock i at the beginning of synchronization interval k, and
K_i : integral gain koefficient.

The integral part of the PI-controller is represented by a sum over an integer range, as the controller acts on discrete time input.

Although the PI-controller adds some complexity and increases the implementation effort, it is still reasonably simple and can thus be used for time updates. Note that although a PI-controlled clock update can compensate clock skew and does not generate any jumps in its output, it may still suffer from a reverse running clock output. Similar to the P-controlled clock, this can happen with large offsets or other large distortions to the input. Careful tuning of K_p and K_i can reduce the risk of these situations, but only a limit similar to the P-controlled clock guarantees a monotonically increasing clock output.

Suggested Solution

Figure 3.18 depicts the expected output for the suggested clock update mechanisms. As previously mentioned, a PI-controlled update provides the best error compensation of all candidates. While more advanced controllers would be possible, we suggest using the relatively simple PI-controller to keep the complexity of the time synchronization minimal.

The remaining concerns are the potentially backwards running clock if a large offset between reference and local clock is present. We suggest limiting the controller output to a reasonable range. While selecting the range for a specific physical setup, the quality of the physical clock has to be considered. The clock synchronization cannot compensate a skew difference of the reference and the local clock larger than the selected limit.

In addition to the PI-controlled clock update, we suggest an initial hard synchronization of the local and the reference clock. This is especially important for systems with a large initial clock offset. Although this generates an initial jump of the local time, it reduces the time needed to first synchronize the local clock. Afterward, the PI-controlled and limited clock update guarantees a monotonically growing clock output. Equations (3.14) to (3.17) define the suggested clock:

[20] See appendix C.1.
[21] See appendix C.2 and appendix C.3.

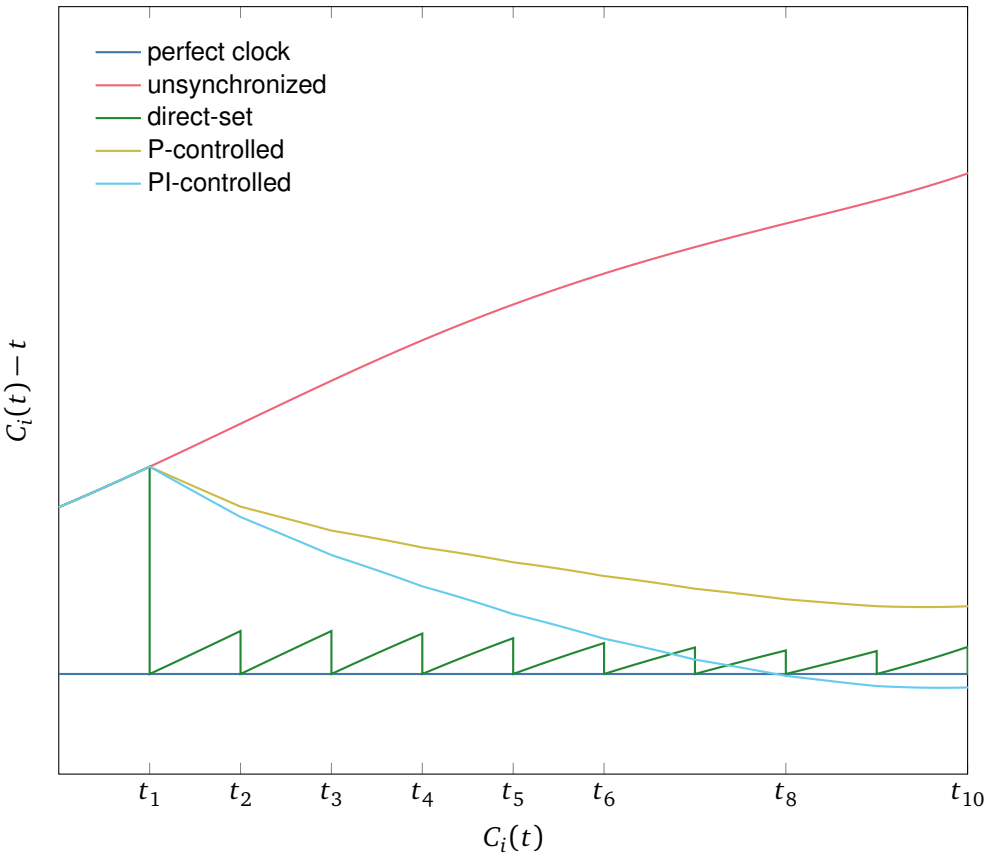


Figure 3.18: Qualitative comparison of clock update mechanisms. Shown is the time difference of the reference clock and a local clock with offset, skew, and drift for the first 10 time synchronization intervals. The direct-set update suffers from time jumps in the local time. A P-controller based update mitigates these jumps, but cannot compensate the remaining offset. Finally, a PI-controller compensates the remaining offset.

$$C_i(t) = \begin{cases} \overbrace{o_i + s_i \cdot t + d_i(t)}^{\text{unsynchronized clock}}, & t \leq t_1 \\ \underbrace{o_i + s_i \cdot t + d_i(t)}_{\text{unsynchronized clock}} + \underbrace{c_i}_{\text{inital offset correction}}, & t_1 < t \leq t_2 \\ \underbrace{C_i(t_k)}_{\substack{\text{local time at} \\ \text{start of interval}}} + \underbrace{(s_i + u_i(k))}_{\text{corrected skew}} \cdot \underbrace{(t - t_k)}_{\substack{\text{time since} \\ \text{start of interval}}} + d_i(t), & t_k < t \leq t_{k+1} \forall k \geq 2 \end{cases} \tag{3.14}$$

$$c_i = e_i(1) \tag{3.15}$$

$$u_i(k) = K_p \cdot e_i(k) + K_i \cdot \sum_{n=1}^{k} e_i(n) \tag{3.16}$$

$$e_i(k) = t_{\text{rx},k} - C_i(t_k) \tag{3.17}$$

with

$C_i(t)$	:	output of clock i at time t,
t	:	reference time,
o_i	:	offset of clock i,
s_i	:	skew of clock i,
$d_i(t)$	:	drift of clock i at time t,
c_i	:	correction term of clock i,
t_k	:	time of reception of time update message k,
k	:	identifier of synchronization interval,
$u_i(k)$	:	skew correction term of clock i in synchronization interval k,
$e_i(k)$	:	clock error at of clock i at the beginning of synchronization interval k,
K_p	:	proportional gain koefficient,
K_i	:	integral gain koefficient, and
$t_{\text{rx},k}$	:	received reference time in time update message k after delay compensation.

For a reliable and unique time representation, the suggested solution assumes an implementation that immediately handles a received time message and updates the clock parameters accordingly. More specifically, it assumes that the transition from synchronization interval k to $k+1$ including the update of $u_i(k)$, t_k, behaves like a single atomic operation. Any process accessing the current time will evaluate $C_i(t)$ based on the latest set of parameters. Detailed information regarding the actual implementation of this time synchronization is available in section 4.3.

3.9 Reliability

Reliability is an important aspect of space missions. Many CubeSats are either dead-on-arrival or fail within the first months of operations [24]. Insufficient testing and issues regarding the solar array deployment are the main reasons for the high infant mortality in CubeSats [64]. A reliability assessment based on detected errors during testing proposed by Langer aims at a good compromise between short time to orbit and reasonable reliability [105]. His method mainly targets the infant mortality due to insufficient testing and thus remaining design and implementation errors in a CubeSat. A study by Dubos, Castet, and Saleh shows that failure rate for small satellites grow again after only a few years [55]. This can no longer be attributed to infant mortality. The harsh space environment and the effects of radiation are one factor leading to defects in various components [223]. A major contributing factor to single event effects of radiations are protons trapped in earth's magnetic field [168]. Especially within the SAA, which is one of the high interest regions for the AFIS mission (see section 2.1.1), an increased intensity of trapped protons is to be expected [145, 168, 223]. Thus, a countermeasure for this kind of defect in combination with proper pre-flight testing promises the best overall reliability.

Design goals 2 and 4 demand a system that designers and developers can adapt to the needs of a specific mission. Identification of failure scenarios and faults leading to these failures provides the

required details to design an appropriate fault tolerance scheme. Within the following sections, we will use the terms fault and failure according to ISO/IEC/IEEE 24765:2017(E) [87]: A fault in this sense is a defect in any component of the system such as a software bug; and a failure is an observable effect of a fault leading to a malfunction of the system where the system can no longer fulfill its purpose.

Within this section, we will first have a look at the different faults, the sources thereof, and the resulting failures. Section 3.9.1 will introduce a classification of faults and failures, followed by an introduction to specific scenarios in section 3.9.2. Finally, section 3.9.3 presents mitigation strategies and solutions a mission designer can use on demand.

3.9.1 Classification

Prior to an analysis of specific scenarios, we require a basic classification of faults and the resulting failures. This not only simplifies the selection of scenarios that require special treatment, but also provides a structured overview to better understand those scenarios.

Origin of Faults

We consider four major categories for the origin of a specific fault:

Design Fault The first source of faults is the overall system design. Faults in the design may manifest themselves due to unsuitable systems engineering practices or a lack of knowledge about a system's parameters. Critical design faults can be detected with extensive testing of the integrated system.

Implementation Fault Implementation faults are faults due to a software or hardware component that does not stick to its specification, e.g., software bugs. Proper testing of all parts of a system and their respective interfaces may uncover implementation faults.

Hardware Fault — Singe Event Effects SEE are faults due to external factors that appear without any prior warning. As their name indicates, the impact of a single charged particle may result in an observable effect, such as a bit-flip in memory. In contrast to design and implementation faults, these cannot be eliminated by a simple modification to the specification or the software implementation. Mitigation strategies exist that reduce the impact of SEEs on the overall system and potentially correct such a fault before they generate an observable failure.

Hardware Fault — Wear out Wear out and the prior gradual degradation of performance parameters of electronic devices is another source for faults in CubeSats. Especially due to the harsh space environment, a faster degradation of components must be expected. After some time components may fail due to wear out and cannot be recovered. The duration until components fail may be extended with shielding or the selection of components specifically intended for space environment.

Failures

Failures are the consequence of faults that render the system uncapable of fulfilling its task. To select proper countermeasures for specific faults or failures of subcomponents, a better understanding of the observable failures is necessary. Any of the previously prevented faults may result in:

- no effect at all, i.e., the fault cannot be observed as it either does not generate any effect or a mitigation strategy hides the effect on a lower, non-observable layer of the system;
- transient effects, i.e., effects that can only be observed for a short period of time;
- and permanent effects, i.e., destructive scenarios where a recovery of the specific component is not possible at all.

The effect can be again split into three different categories: fail-silent, wrong data, or bogus behavior. Fail-silent failure modes can be observed by the absence of an expected output. An example for a fail-silent failure mode would be an unavailable component due to power loss. Wrong-data failure modes do not affect the responsiveness or behavior of a component; only the content of the output is wrong. The failure mode of a sensor reporting wrong values is an example of a wrong-data failure mode. The last category, bogus behavior, summarizes all failure modes that affect the control flow of a component. This can lead to any kind of unexpected output like additional unwanted interaction with other components of the system.

3.9.2 Considered Scenarios

Concrete failure scenarios provide the basic information for the design and implementation of mitigation strategies to finally increase a system's reliability. This section will present the considered failure modes divided into three categories: failures due to a priori existing faults, failures affecting a single node, and failures affecting the communication between nodes. While the first category focuses on the origin of the fault and the fact that it could have been detected during testing, the other two categories focus on the part of the system affected by a certain fault.

A-Priori Existing Faults

In this category, we consider two separate faults that may lead to system failures: design faults and implementation faults. Although for a specific fault it may not be clear if it originates from the design or the implementation, it is important to separate them for proper mitigation strategies. Both failure scenarios have a permanent effect. Thus, they can only be resolved with a new revision of the system.

Design Faults The design of a system may contain faults that lead to system failures. Such faults can be hidden in various parts of the design, but ultimately lead to an invalid state of the overall integrated system. An example for such a fault is a high-level state machine controlling the system that contains a potential lockup, which freezes the system if a certain condition or sequence of conditions appears. Although according to ISO/IEC/IEEE 24765:2017(E) a design fault is a "[...] fault that results from a human error during system design [...]" [87, p. 131], we consider all faults of high-level interaction between components as well as faults originating from invalid or contradicting assumptions about a system as design faults. Thus, design faults only appear in the interaction of components and analyzing a subset of the system may not reveal them. Design faults appear deterministically, i.e., if the same sequence of actions leading to a system failure as a consequence of a design fault occurs, the same failure happens again. Any failure mode can be activated due to design faults, i.e., a system may fail-silent, generate wrong data, or behave wrongly.

Implementation Faults We consider classical bugs in a system's realization as implementation faults. These faults can be summarized as a breach of the interface contract of a single component and may appear in hardware or software. Similar to design faults, implementation faults appear deterministically and may lead to any failure mode. In contrast to design faults, implementation faults can be observed by applying the same sequence of input values to a single component of the system.

Failures Affecting a Single Node

The second category contains all failures of a single node due to a fault that did not exist within the design or implementation of the system and only manifested at a later state of a mission. We consider three different scenarios: bit errors, unexpected resets of a node, and permanent failure of a node.

Bit Errors Bit errors, e.g., bit-flips or stuck bits, may appear spontaneously in any part of a node. In many cases these bit errors are a consequence of single-event effects [168, 223]. Depending on the affected memory region, a bit-flip may lead to any failure mode. Bit-flips in data or background memory

used for log files and measurements are in most cases less critical to the overall system performance. Bit-flips in code memory or the caches and registers of an MCU have a higher impact and will most likely lead to unexpected behavior. In most cases, bit-flips are a temporary effect and will not permanently damage the memory cells.

Unexpected Reset An unexpected reset is a temporary fault that resets an entire node into its initial configuration. Many faults can lead to this failure, e.g., power reset due to EPS protection circuits, watchdog resets due to a faulty firmware, resets triggered by the firmware itself, or uncorrected bit errors in code memory. In either case, we assume the node to reset its entire state and restart its operation as if it was just powered on. We furthermore assume that the underlying fault only sporadically affects the system and that the fault is no longer active after a system reset. Nevertheless, a node will lose its entire runtime state and any synchronization with other nodes of the distributed system as a consequence of an unexpected reset.

Permanent Failure The last failure mode of a single node is a permanent failure of the entire node or parts thereof. While the previous failure mode is temporary and recovers its functionality after a full system reset, recovery from a permanent failure is not possible. An example for a permanent failure is a node that can no longer operate due to physical damage or wear out, e.g., due to radiation or thermal cycling.

Failures Affecting the Communication between Nodes

The third category contains two considered scenarios of failed communication between different nodes: either wrong communication with invalid or unexpected data or missed communication where the transmission of parts of the information is unsuccessful.

Wrong Communication If faults occur during transmission of information between nodes, a receiving node may process invalid data. We name this failure state a wrong communication failure. Depending on the further use of the received information this may have a wide range of side effects.

Missed Communication Whenever a transmitting node sends data to another node, we expect the receiving node to receive and process the data. If it does not receive any data at all, a missed communication failure occurred. A missed communication failure may be temporary, e.g., a receiver missed a single message, or permanent, e.g., an interruption of the physical interconnection between nodes.

3.9.3 Mitigation Strategies

We divide the mitigation strategies into three categories based on the abstraction layer suggested for their respective mitigation: early detection prior to launch, mitigation in hardware, and mitigation in software.

Early Detection of Design and Implementation Faults

We do not suggest runtime migration for design or implementation faults, but instead suggest the use of extensive testing of all parts of the system. While unit testing can detect most implementation errors, integration testing provides insight into the interaction of different components. Within MOVE-II, we made good experiences with early integration testing in various different scenarios. Although even with combined unit and integration testing some faults may still be hidden in the system, but Langer suggested that a saturation in detected faults over time can be achieved [105]. Further analysis of design and implementation faults and best practices regarding testing and verification of a CubeSat is not part of this thesis.

Fault-Tolerance in Hardware

The different fault scenarios require an individual handling. Out of these scenarios, bit-flips, node resets, and communication problems can be partially handled in hardware.

Bit-Flips We suggest the use of hardware specifically suitable for space environment to reduce the risk of bit errors and wear out related symptoms in critical nodes. This is especially critical for the MCU and memory used for code storage and other critical mission data. COTSs suitable for this purpose exist, such as the ARM-M based MCUs by Vorago Technologies [228]. The radiation hardening of this device exceeds all requirements for LEO applications. If the internal memory of such a device is not sufficient, additional external memory provides the storage for program code and critical data. MRAM or FRAM are sufficiently radiation tolerant for LEO applications [71, 78]. Thus, we assume that these devices' protection against bit-flips in any registers, relevant sections of code memory, and critical data memory is sufficient for our application.

Node Reset or Failure While the use of space rated hardware also reduces the risk of node resets or permanent failures, it cannot entirely prevent them. Especially lockups in a system may still happen due to design or implementation faults that were not detected during testing. To avoid permanent loss of a node, we suggest the use of watchdog timers. These watchdogs must be repeatedly reset by the software and otherwise trigger a reset of the entire node. They avoid permanent node failures due to stuck software by simply resetting an unresponsive node, but cannot recover from permanent node failures due to defective hardware components. The specific watchdog setup depends on the requirements of the respective mission and many CubeSat designers and developers rely on their capabilities to recover from off-nominal situations [24]. Thus, higher level software must still be able to recover from sporadic node resets and in some cases even permanent node failures.

Communication CAN already covers a wide range of faults and is capable of detecting errors during transmission and retransmission of failed frames [81–83] and may be sufficient for most CubeSats. In addition, especially for missions with increased reliability requirements, we suggest using the CAN extension as suggested in ECSS-E-ST-50-15C [59]. This standard suggests the use of hardware redundancy and a physical layer that provides the high reliability features of ISO 11898-3:2006(E) [83] at higher signaling rates than originally suggested. This setup mitigates the risk of wrong communication and reduces the probability of missed communication due to physical reasons to a minimum. While these features ensure successful transmission of CAN frames, higher layer protocols may still suffer packet loss due to the finite size of buffers in real implementations. Therefore, missed communication must still be considered as a relevant scenario on higher layers.

Fault-Tolerance in Software

The remaining failure modes are unexpected node resets, permanent node failures, and missed communication. In most CubeSats short downtimes due to a system reset or a missed communication are acceptable, thus a mitigation scheme based on recovery from off-nominal situations can be used. In those systems, it is essential to automatically activate all components after a reset as fast as possible, reattach the components to the respective communication and resume their respective tasks. *Components* that require an internal state, i.e., such components that require knowledge about past events, may include an additional resynchronization step to synchronize their state with other nodes or reload the last known state from permanent memory. If fail-over entities exist and these entities need to agree on a fail-over situation, a leader-election algorithm assures a synchronized decision in all nodes. Section 3.9.4 proposes a bully algorithm for this election. If an application is only active on a single node at a time, a checkpoint and restore mechanism provides the necessary features. It can transfer a state either to permanent memory or to a different node. Section 3.9.5 presents the basics of such a mechanism.

Missed communication failures are only a minor concern as sporadically missed messages can be tolerated similar to a system reset. In both cases the application needs to be aware of the possibility of

such errors and carefully treat every communication. Especially situations where software infinitely waits for a missed message can lead to failures of the entire system. In general, application developers should implement the handling of missed communication within the DOSIS framework. The framework itself provides the required asynchronous interfaces to messages and the latest state of the sub-values of any register (see sections 3.4 and 3.5).

If short downtimes are not acceptable for a specific application, this application requires other mitigation mechanisms. One or more hot redundant copies of an application can seamlessly continue the required task. The DOSIS framework and the global knowledge of information provides all means to implement such a mechanism as part of the application. Section 3.9.6 presents the basic options for redundant execution of a specific task and the required considerations for application developers.

3.9.4 Simplified Bully for Leader Election

This section will present the simplified Bully leader election tailored to the environment of the DOSIS framework. The first part of this section introduces the original Bully algorithm and modifications thereof, followed by a detailed description of the suggested simplified Bully leader election.

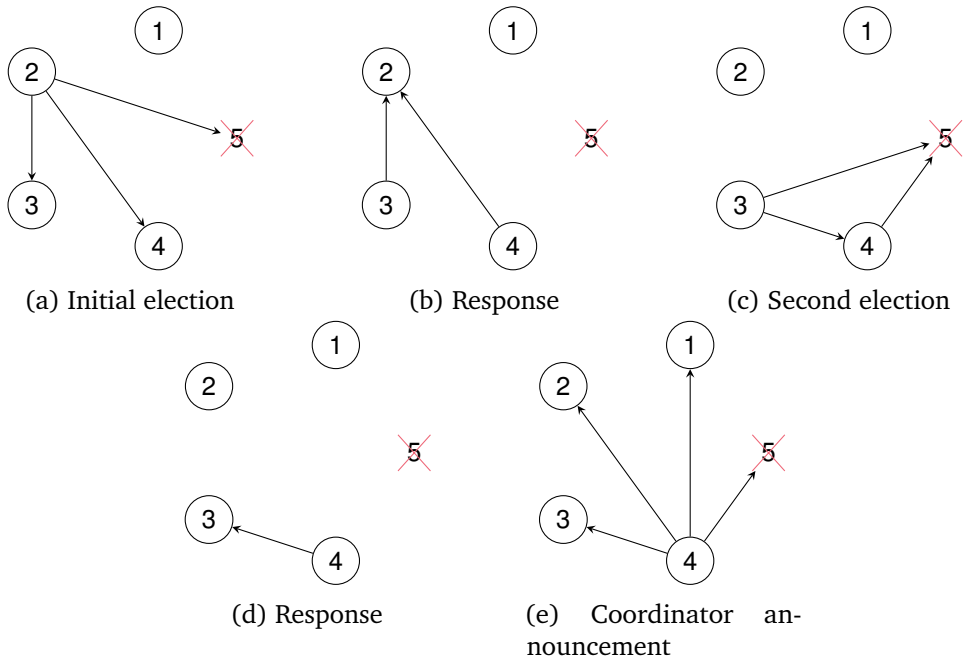
Previous Versions of the Bully Leader Election Algorithm

The first version of the Bully algorithm, proposed by Garcia-Molina, is a widely applicable algorithm for leader election in a distributed system [69]. This paragraph summarizes the algorithm according to [69]: Each node has a unique identifier in a certain range. Any node may initiate an election by sending a special election message to all nodes that have a larger identifier than the initiating node itself. If those nodes are alive, they will individually respond to the initiator of the election with an ok message. Thus, the original initiator knows that a node with higher identifier is available and thus knows it lost the election process. After replying to an election message, each of the nodes will initiate its own election. If a node does not receive a single ok message in response to its election message, it is the node with the largest available identifier. Thus, it wins the election and sends a message to all other nodes to inform them about the changed leader. Figure 3.19 depicts the Bully leader election after the previous leader has failed.

The main disadvantage of the Bully algorithm is the amount of messages required for a full election process. The amount of messages of a full election using the Bully algorithm is in $\mathcal{O}(n^2)$ [70] as each node initiates an election and receives one response from each live node. The high message count can be reduced with a nomination based extension of the Bully algorithm proposed in [111]. In this version of the algorithm the election initiating node collects the responses of all other nodes and nominates the node with the highest identifier. Afterward, the nominated node announces itself as winner of the election process using a coordinator message. Figure 3.20 presents an exemplary election process for this version of the Bully algorithm.

The authors of [97] and [70] present a similar modification, which calls the nomination message grant messages. Additionally, this version adds stop messages to terminate a concurrently started election process and thus further reduce the amount of messages. The number of messages required for a full election using this modified Bully algorithm is in $\mathcal{O}(n)$ [70]. The main disadvantage of this version of the Bully algorithm are the large timeouts and thus the long runtime of an election [155]. Soundarabai et al. add a flag to suppress concurrently started elections to the grant based Bully algorithm. Extending the state during the election to also keep track of the second-highest identifier reduces the overhead if the potential candidate fails mid-election [4].

A different approach by Mamun, Masum, and Mustafa reduces the number of required messages without an additional nomination or grant message [129]. Instead, a node that initiated the election also announces the new leader and directly transmits the coordinator message [129]. Additional messages simplify the recovery process of previously failed nodes without initiating a full election [129]. The solution in [129] does not address some issues regarding concurrently initiated elections and some special cases of recovering nodes [155]. Murshed and Allen proposes an adaptation of the algorithm presented in [129] that addresses these concerns [155]. They split the set of nodes into potential candidates and ordinary nodes of an election [155] and thus reduce the amount of messages required



(a) Initial election (b) Response (c) Second election

(d) Response (e) Coordinator announcement

Figure 3.19: Leader election using the original Bully algorithm according to [69]. Node 2 initiates the election by sending an election message to all nodes with larger identifier (a). All live nodes respond to the election message, which also stops the active election process in node 2 (b), and initiate their own election (c). Again, all live nodes respond to the election message (d). The node that has the largest identifier among the live nodes does not receive any response. Therefore, it announces itself as new coordinator to all nodes (e).

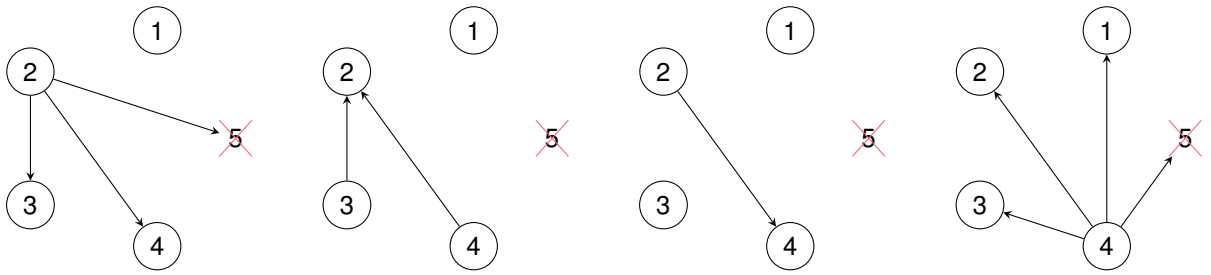
for a full election. An ordinary node may still become the elected leader if all candidate nodes fail [155]. Different timeout values for received ok messages resolve concurrently initiated elections, as a node with larger identifier will end the election first [155]. This version also uses multicast messages to keep the amount of messages for an election process minimal [155]. Kabashi, Zeqiri, and Zabeli suggest a simplification for some cases where the nodes with highest and second-highest identifier may become the leader without a full election [89]. Figure 3.21 depicts a nominal election as suggested in these versions of the Bully algorithm. Similar to the nomination or grant based Bully algorithm, this approach has a worst-case complexity in $\mathcal{O}(n)$ [89, 129, 155].

Proposed Simplified Bully Leader Election Algorithm

We propose a simplified version of the enhanced Bully algorithm from [129, 155]. The suggested simplifications reduce the complexity of the algorithm and make use of the CAN communication's broadcast nature.

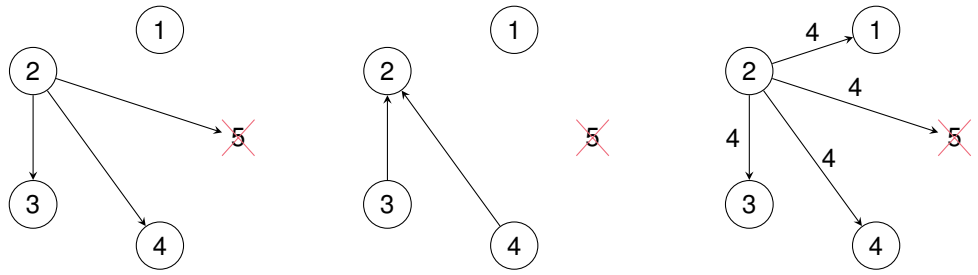
The leader election must fulfill two main criteria: safety and liveness [155]. In this context safety means that all nodes must agree on a single leader; liveness mains that after an election all nodes must agree that the election has finished and accept the elected leader. The algorithm furthermore assumes that:

1. all nodes use the same algorithm for leader election;
2. the system is synchronous and enables the use of timeouts;
3. the system never halts temporarily;
4. the network does not fail, only individual nodes may fail;
5. the network strongly orders all transmitted messages;
6. every node n has a unique identifier $i_n \in \mathbb{N}^+$;



(a) Election (b) Response (c) Nomination (d) Coordinator announcement

Figure 3.20: Leader election using a nomination or grant based Bully algorithm similar to a nominal election in [4, 70, 97, 111, 201]. Node 2 initiates the election (a) and receives the response of all live nodes (b). After a timeout, node 2 selects the node with the highest identifier, in this case node 4, and nominates this node as new leader (c). Node 4 announces itself as new coordinator to all nodes (d).



(a) Election (b) Response (c) Coordinator announcement

Figure 3.21: Leader election using a Bully algorithm where an election initiating node announces the new leader as used in [89, 129, 155]. Node 2 initiates an election (a), gathers the response of all live nodes (b), and directly announces the new elected leader (c).

7. every node knows the set of used identifiers $I_{\text{all}} := \{x \mid \exists n : x = i_n\}$;
8. the nodes do not know the subset of live nodes $I_{\text{live}} := \{x \in I_{\text{all}} : x \text{ is alive}\}$ and can detect the failure of an elected leader only based on timeouts; and
9. crashed nodes may rejoin the election at any time.

These assumptions are similar to the assumptions in [69] and the assumed identifiers in [155]. Only item 5 differs and enforces a total order of all messages. Garcia-Molina [69] only assumes a strong ordering of messages from a certain node. This assumption can be made as CAN sequentially broadcasts all messages on a single physical connection. It is clear that every node will receive those messages in the same sequential order.

Additionally, every node recognizes the current leader and stores the unique identifier of said leader i_l in a local variable `leader`.

The suggested algorithm uses five types of messages similar to the messages in [129]:

election An *election* message initiates an election. It is sent by a node d that detected the absence of the previous leader. The message contains the unique identifier i_d of the sending node.

ok An *ok* message is the response to a received election message, indicating that a node n is alive. This message contains the unique identifier i_n of the sending node.

coordinator A *coordinator* message announces a new leader node l. It contains the unique identifier i_l of said leader.

query A resuming node r sends a *query* message to request information about the current leader without initiating a new election. A *query* message contains the unique identifier i_r of this node.

answer *Answer* messages are the reply to query messages. An *answer* message contains the unique identifier i_l of the current leader.

In contrast to [129], all messages are broadcast-messages.

Regular Leader Election Whenever an arbitrary node d detects the absence of the leader, it initiates an election. The election starts with an *election* message; Node d broadcasts this message including its own identifier i_d and starts a timeout T_{election} to wait for *ok* messages. Every live node n with $i_n > i_d$ will wait for $T_{\text{send ok}}$ and afterward reply with an *ok* and start a timeout T_{ok}. The timeout prior to sending the *ok* message reduces the amount of messages for concurrently initiated elections. This is presented in more detail in Concurrent Elections on page 81. All other live nodes do not reply and start a timeout T_{fail}. Unless multiple nodes fail during the election process, this timeout will not be reached. If node d did not receive a single *ok* before its timeout has passed, it wins the election, broadcasts a *coordinator* message using its own identifier i_d, and updates its own `leader`$:= i_d$. Otherwise, it selects the largest identifier of the received *ok* messages as the new leader, i.e., $i_l = \max(\{i_n \in I_{\text{all}} \mid i_n \in \text{received } ok \text{ messages}\})$, broadcasts a corresponding *coordinator* message and updates its own `leader`$:= i_l$. All live nodes update their respective `leader` with the i_l of the received *coordinator* message. This election is similar to the election process shown in figure 3.21, but uses broadcast messages filtered at the receiving node instead of individual messages.

Node Recovery A previously failing node may reappear and rejoin the set of live nodes I_{live}. This node r will execute a special recovery sequence similar to the recovery presented in [89]: If the recovering node has the maximum identifier of all nodes, i.e., $i_r = \max(\{i_n \in I_{all}\})$, it immediately broadcasts a *coordination* message announcing itself as new leader and updates `leader`$:= i_r$; Otherwise, it broadcasts a *query* messages and waits for T_{query}. The currently elected leader l will reply with an *answer* message containing its own identifier i_l. If i_r does not receive an *answer*, it knows that the current leader node has failed and initiates a regular election; Otherwise, it compares its own identifier with the received identifier. If $i_r > i_l$ as received within the *answer* message, node r immediately announces itself as new leader in a *coordination* message and updates its `leader`$:= i_r$; Otherwise, it accepts node l as leader and updates its `leader`$:= i_l$.

Failure of Detecting Node A detecting node d may fail after transmitting the initial *election* message but before announcing the elected leader. All nodes that replied to this *election* message detect the absence of d after T_{ok} if they do not receive a *coordinator* message. In this case the node with the largest identifier among the *ok* messages will announce itself as new leader. It publishes a *coordinator* message containing its own identifier and updates its `leader` accordingly.

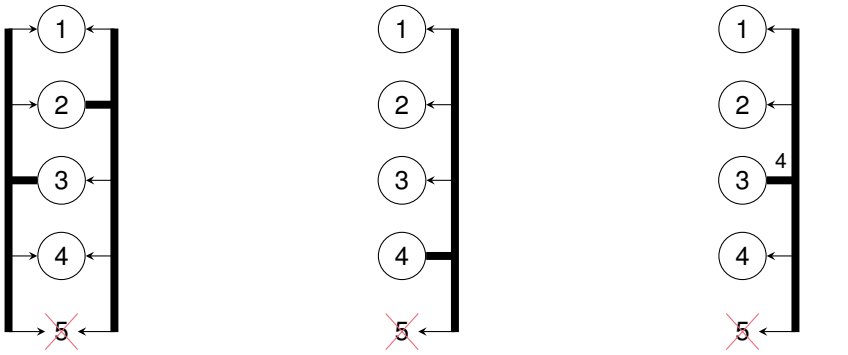
In contrast to [155], we do not use individual timeouts but instead directly base the decision on transmitted messages: The winning node detects that it won the election based on the *ok* messages of all other nodes. The broadcast nature of all messages on the CAN bus enables access to this information. The total order of messages on the CAN bus guarantees that each node receives the initial *election* message prior to any of the *ok* messages. Thus, it is safe to only take those *ok* messages into account that were received after the initial *election* message.

If node d was the node with the highest identifier alive, no other node does reply with an *ok* message. In this case, no node can decide which node should become the new leader. Therefore, no node announces a winner with a *coordinator* message. Nodes that sent an *ok* can detect the absence of the *coordinator* message after waiting for an additional $T_{\text{await coordinator}}$, all other nodes after their T_{fail} timeout. In this case, those nodes will restart the election process. The same timeouts also recover from a concurrent failure of node d and the node with the highest identifier mid-election after it sent the *ok* message. This special case with a failure of a very specific node or a concurrent failure of two nodes is unlikely. Thus, we accept the large amount of concurrently initiated elections.

Failure of New Leader Mid-Election If node l' with $i_{l'} = \max(\{i_n \in I_{\text{all}}\})$ at the beginning of the election process fails mid-election, it may still be announced as new leader. This happens if it fails after sending

its *ok* message. All live nodes agree on a leader and the election process terminates regularly. The result is similar to the elected leader failing after the election has completed. One of the remaining live nodes will eventually detect the absence of the leader and initiate a new election. Thus, no special treatment of this situation is required.

Concurrent Elections In some situations, multiple nodes may detect the absence of the leader concurrently. Each of those nodes will initiate an election by transmitting an *election* message. As previously mentioned, all nodes that potentially reply to the *election* message will delay this response and thus only send a single *ok* message each. Once a node reaches its $T_{\text{send ok}}$ timeout, it decides whether to send an *ok* message based on its own identifier and the highest identifier received in one of the previous *election* messages. An election-initiating node a will not continue its active election after it receives an *election* message from a node b with $i_b > i_a$. Thus, only a single node with the highest identifier among the election-initiating nodes will continue the election process. This mechanism stops all but a single concurrently initiated election, similar to the stop messages used in other versions of the bully algorithm, e.g., [70], and does not require individual timeouts as suggested in [155]. The remaining election will continue similar to the regular leader election presented on page 80. Figure 3.22 depicts the messages of two concurrent elections.



(a) *Election* (b) *Response* (c) *Coordinator*

Figure 3.22: Concurrently initiated elections with the proposed Bully algorithm. In (a), nodes 2 and 3 concurrently initiate an election. All nodes wait for $T_{\text{send ok}}$ after receiving the first *election* message. Afterward, nodes send an *ok* message if and only if their own identifier is higher than the highest identifier of the previous *election* messages. This is only the case for node 4, which sends an *ok* message in (b). Node 2 does not continue with its election as it received an *election* message from node 3. Finally, in (c) node 3 broadcasts a *coordinator* message announcing node 4 as new leader.

Suboptimal Node Elected In real scenarios, especially if high load on the CAN bus increases message transmission times, a suboptimal node l' where $i'_l \neq \max(\{i_n \in I_{\text{live}}\})$ may win the election. To resolve this situation, a node that receives a *coordinator* message with an identifier lower than its own identifier immediately sends another *coordinator* message announcing itself as new leader. A similar behavior was also suggested by Kabashi, Zeqiri, and Zabeli [89]. Although this may lead to many *coordinator* messages, it resolves unintended situations and thus assures that all live nodes will eventually agree on a single leader. This is guaranteed, as all nodes handle received *coordinator* messages and the strong order of messages on the shared CAN bus assures that all nodes process these messages in this order.

Timeouts We can define a time T_{resp} as upper bound until a node that sends a request to another node receives the response. Due to the synchronous system and the assumption of a never failing network, this timeout will be respected unless a node fails. A failure detector based on this timeout can be relied on as initial failure detector and as failure detector during an election process [129]. According to [129], T_{resp} is defined based on the transmission time of a message T_{trx} and the processing time T_{p}

as

$$T_{\text{resp}} = 2 \cdot T_{\text{trx}} + 1 \cdot T_{\text{p}} . \tag{3.18}$$

Each message can be encoded in a single CAN frame, assuming the message, consisting of the message's type and the respective node identifier, can be embedded in 8 B or less. From equation (3.4) we know that the transmission of a single CAN frame may require up to $T_{\text{CAN frame+}} = 160\,\mu s$. Thus, for N nodes sending a message concurrently, the transmission time for all messages is less than $T_{N \text{ CAN messages}} = N \cdot T_{\text{CAN frame+}} = N \cdot 160\,\mu s$. The propagation delay of the CAN message in the physical medium is small compared to $T_{\text{CAN frame+}}$ and thus neglected for simplicity. For a real application where other traffic exists on the CAN bus, we suggest a $T_{\text{trx}} > T_{N \text{ CAN messages}}$. The added margin also covers the neglected propagation delay. Furthermore, if $T_{\text{trx}} \gg T_{\text{p}}$ and it contains some margin, we can simplify equation (3.18) to

$$T_{\text{resp}} \approx 2 \cdot T_{\text{trx}} \tag{3.19}$$

as stated in [155].

Based on T_{trx} and T_{resp}, we suggest the following timeouts:

$$T_{\text{send ok}} \geq 1T_{\text{trx}} \tag{3.20}$$

$$T_{\text{await coordinator}} \geq 1T_{\text{trx}} \tag{3.21}$$

$$T_{\text{election}} \geq T_{\text{resp}} + T_{\text{send ok}} \geq 3T_{\text{trx}} \tag{3.22}$$

$$T_{\text{ok}} \geq T_{\text{election}} \geq 3T_{\text{trx}} \tag{3.23}$$

$$T_{\text{fail}} \geq T_{\text{send ok}} + T_{\text{ok}} + T_{\text{await coordinator}} \geq 5T_{\text{trx}} \tag{3.24}$$

$$T_{\text{query}} \geq T_{\text{fail}} \geq 5T_{\text{trx}} \tag{3.25}$$

We suggest a single T_{trx} prior to sending an *ok* message, as this assures that a node receives all *election* messages of concurrently initiated elections. No node will decide to initiate an election while a previously started election has not yet finished. Thus, a single *ok* message is sufficient even if all nodes concurrently initiate an election.

T_{election} is defined similar to [129] extended by the time a node waits before transmitting its *ok* message. T_{ok}, which is used by all nodes after transmitting an *ok* message to wait for a *coordinator* message, is equal or longer than T_{election}. A node will start this timer after sending the *ok* message, i.e., after $T_{\text{send ok}}$. This timer will only trigger at least T_{trx} after the election-initiating node broadcasts the *coordinator* announcement. If the election-initiating node fails and does not broadcast a *coordinator* massage, nodes which did reply with an *ok* message select the leader as previously suggested (see page 80). One of those nodes may publish a *coordinator* message, therefore all other nodes wait for at least T_{trx} to await this message. Only if this message is also missing, the entire election failed. Therefore, the timeout of nodes that do not send an *ok* message should be at least as long as all timeouts of those nodes that sent such a message combined.

Finally, the timeout after sending a *query* message to await an *answer* should be longer than the time until an initiated election finally fails, i.e., T_{fail}. This assures that a resuming node, which will not receive an *answer* while an election is still running, does not interfere with the election process. If an election was running, the resuming node will receive a *coordinator* message before this timeout has passed. If it does not receive such a message, either the election has finally failed or the selected leader has failed. In both cases it is safe to initiate a new election once this timeout has passed.

Algorithm Summary Algorithms 3.1 to 3.5 depict the procedures of the suggested bully algorithm. As mentioned earlier, each node knows its own identifier i_{own} and the set of used identifier I_{all}. Additionally, each node has a local variable `leader` that stores the identifier of the currently elected leader. In addition to the identifiers, this variable may also store a special value indicating that the leader is currently unknown. A node triggers algorithm 3.1 whenever it detects the absence of the current leader. Whenever a node recovers from a previous fault condition, it executes algorithm 3.4. Algorithms 3.2, 3.3 and 3.5 are triggered whenever a node receives the corresponding message type. The only exception

is algorithm 3.2: A node does not trigger this algorithm again, if it is already running and currently waiting for additional election messages in line 7. In all algorithms, WAIT is a special function that suspends the execution for a period of time specified with the first parameter; BROADCAST sends a message of a given type and optionally with a specific identifier on the CAN bus; and ABORT aborts processing of the specified procedure if the local node is currently executing this procedure.

Algorithm 3.1 Initiate election

```
 1: procedure INITIATE ELECTION
 2:     leader←unknown
 3:     BROADCAST(election, i_own)
 4:     WAIT(T_election)                          ▷ Wait for incoming ok messages
 5:     N ←number of received ok messages
 6:     if N = 0 then
 7:         leader← i_own
 8:     else
 9:         leader← max(i_n ∈ {received ok messages})
10:     end if
11:     BROADCAST(coordinator, leader)
12: end procedure
```

Algorithm 3.2 Receive *election*

```
 1: procedure RECEIVE ELECTION(i_source)
 2:     leader← unknown
 3:     if i_own < i_source then
 4:         ABORT(INITIATE ELECTION)                ▷ Stop concurrent elections
 5:     end if
 6:     ABORT(RESUME)
 7:     WAIT(T_send ok)                   ▷ Wait for additional election messages
 8:     i_max ← max(i ∈ {received election messages})
 9:     if i_max > i_own then                                ▷ active participation
10:         BROADCAST(ok, i_own)
11:         WAIT(T_ok)          ▷ A coordinator message and thus an abort is expected here
12:         if i_own > max(i ∈ {received ok messages}) then
13:             leader← i_own
14:             BROADCAST(coordinator, i_own)
15:         else
16:             WAIT(T_await coordination)
17:         end if
18:     else                                      ▷ no active participation in election
19:         WAIT(T_fail − T_send ok)
20:     end if
21:     INITIATE ELECTION                 ▷ Only called if election failed entirely
22: end procedure
```

Algorithm 3.3 Receive *coordinator*

```
1: procedure RECEIVE COORDINATOR(i_new leader)
2:     ABORT(INITIATE ELECTION)
3:     ABORT(RECEIVE ELECTION)
4:     ABORT(RESUME)
5:     if i_new leader > i_own then
6:         leader← i_own
7:         BROADCAST(coordinator, i_own)
8:     else
9:         leader← i_new leader
10:    end if
11: end procedure
```

Algorithm 3.4 Resume

```
1: procedure RESUME
2:     leader← unknown
3:     if i_own = max(i ∈ I_all) then
4:         leader← i_own
5:         BROADCAST(coordinator, i_own)
6:     else
7:         BROADCAST(query)
8:         WAIT(T_query)                          ▷ Wait for answer message
9:         if answer received then
10:            i_answer ← i from received answer message
11:            if i_answer < i_own then
12:                leader← i_own
13:                BROADCAST(coordinator, i_own)
14:            else
15:                leader← i_answer
16:            end if
17:        else                                   ▷ No answer received
18:            INITIATE ELECTION
19:        end if
20:    end if
21: end procedure
```

Algorithm 3.5 Receive *query*

```
1: procedure RECEIVE QUERY
2:     if leader = i_own then
3:         BROADCAST(answer, i_own)
4:     end if
5: end procedure
```

Time Synchronization Reference Node Selection

As stated in section 3.8.1, multiple candidate reference nodes may be available. The runtime selection of the currently active node uses the proposed Bully algorithm for its leader, and thus active reference-node selection.

The proposed Bully algorithm elects the leader with maximum identifier alive. If a node with high priority resumes, it will be immediately elected as the new leader. If the reference node does not contain an absolute time source, it may have a large initial deviation from the system time of the remaining system. Immediately starting to broadcast the time from such a node may be undesired, as it leads to potentially large time deviations and thus potential timing issues. Thus, we suggest a resume procedure surrounding the Bully algorithm.

A resuming node will first wait for at least three time synchronization intervals T_{ts} and behave as a regular node during this time. Due to the initial direct update of the local clock in the hybrid clock update mechanism, larger clock differences will be mitigated after this startup phase. Only afterward, the node will resume its Bully implementation and may thus become the reference node. If for a certain system all reference nodes use an absolute clock source, e.g., a GNSS based time source, and the system time is an absolute time, a designer can choose to skip this extended resume mechanism. The absolute time sources mitigate the risk of a potentially large initial time offset.

Please note that resuming a node and resuming the time synchronization as potential reference node may not be directly connected. If a node has an external reference clock that requires a prolonged startup, the node may resume its operation and participate in the time synchronization as regular node. A GNSS time source is an example of such a time reference that may require some time for initialization and first synchronization with the external time source.

Any potential reference node may detect the failure of the currently elected reference node due to repeatedly missing time update messages. Any of those nodes may initiate an election according to the proposed Bully algorithm. To avoid repeated detection of a failure while an election is still in progress, we suggest using a time synchronization interval $T_{\text{ts}} \gg T_{\text{fail}}$ of the Bully algorithm.

3.9.5 Task Migration

In some situations it may be required to continue the execution of a task on a different node of the distributed system, e.g., due to a failed node or a change of the mission phase. Visionary Data Management System for Nano-Satellites (VIDANA), a RODOS based system for task migration [229], targets dynamic scalability to increase dependability and performance of a distributed system [152]. While [152] mentions the possibility to scale this system to very small satellites, it still requires sufficient resources for standby instances of a task on all nodes that may eventually execute said task [150]. Regular context updates keep the state known to all standby instances [150]. A so-called Task Distributor knows the entire system state and on-demand activates a task on a specific node [150].

VIDANA provides basic scalability and increased reliability due to the possibility to resume work on a different node. The downside of VIDANA is the required RAM due to the standby instances of all tasks on all nodes. Therefore, we suggest using a system capable of runtime migration of the entire RODOS thread. Such a system would be similar to distributed systems in automotive contexts, e.g., as presented in [192], which enables task migration [193] based on a runtime checkpoint and restore mechanism implemented for embedded system using the L4 microkernel [191].

For a system based on RODOS and DOSIS, a full migration, including migration of executable code at runtime, is not entirely feasible. The lack of an MMU and virtual addresses prevent the execution of arbitrary code from arbitrary locations in memory. Thus, we suggest a thread migration that requires the code present on all nodes similar to the migration suggested in VIDANA. Only the state of a thread should be migrated at runtime, including the creation of RODOS threads and thus enabling the reuse of resources, especially RAM, for different tasks. This system can transfer tasks between nodes at runtime. If a node periodically publishes a snapshot of its internal state similar to the status updates in VIDANA [150] or an external node replicates said state, a task can be resumed on a different node after an arbitrary node failure. Konlechner [96] and Föger [65][22] demonstrate the basics of a runtime

[22] Both theses supervised by the author of this work.

migration of entire RODOS threads in their theses.

3.9.6 Redundant Execution

In contrast to the migration of tasks, which mainly guarantees that a service can be resumed if a certain node fails, redundant execution enables a continued operation without any service interruption. In the presentation of [150], the authors suggest the redundant execution of a critical task within VIDANA, although they do not mention any details or the expected impact on user applications.

The DOSIS framework enables the redundant execution of a task on arbitrary nodes of the system. As mentioned in section 3.4, multiple concurrent *ComponentInterfaces* and *ComponentImplementations* are possible, although a system designer and developer has to take special care in this case. We distinguish two general scenarios: Either multiple copies of a *ComponentImplementation* are fully active at the same time and do not perform any internal coordination, or these *ComponentImplementations* internally track the state of an active node and act as if they were a single entity.

Fully Active *ComponentImplementations*

A set of fully active replications of a *ComponentImplementation* enables an uninterrupted service even with arbitrary nodes failing. This is achieved with N instances of a *ComponentImplementation* and voting on all output values within interfacing nodes.

In such a system, all replicated *ComponentImplementation* instances behave as if they were the only available instance; each uses an individual *deviceDef*. The command-topic identifiers of all *deviceDefs* may be identical, but the data-topic identifier is unique for each *deviceDef*. Another *Component* that accesses these instances will utilize N copies of the *ComponentInterface*. Commands sent via any of these *ComponentInterfaces* will be received by all instances of the *ComponentImplementation* if the command-topic identifiers are identical; otherwise, the accessing *Component* commands each replication individually. Each *ComponentImplementation* instance replies to the command on its assigned data topic. The accessing *Component* handles all replies individually and combines them accordingly. Accessing *Components* similarly handles data generated in the replicated *ComponentImplementations*, e.g., within an *Interval Module*.

A replicated *Component* may also command other *Components*. These *Components* will receive (up to) N copies of any command. Thus, they must be aware of the replication and combine these commands accordingly. Afterward, these *Components* should output a single response.

Although N copies of a *ComponentImplementation* provide a high level of reliability, they also require a lot of effort and care by a system designer. Especially the required changes to components outside the replicated *Component* itself reduce the potential reuse and thus increases the development effort drastically.

State Tracking *ComponentImplementations*

With state tracking replications only a single instance within the set of replicated instances will generate any output at a specific point in time. This enables encapsulation of all changes required for this replication into the replicated *Component* itself. This simplifies the implementation effort and increases the potential for reuse between missions as a consequence of the encapsulation, but it cannot provide the same level of reliability as fully active redundant. Yet it potentially shortens the detection and recovery time compared to a task migration after a failure.

In contrast to the fully active replicas, the replicated instances know the presence of the other instances and act accordingly. They all share a single *deviceDef* and additionally monitor the data topic, i.e., their own output. All replications receive all commands and process them individually. A single and dynamically selected replication generates the response to this command; we call this replication the main instances, all others tracking instances. Tracking instances compare this response to their own version. If the responses differ or the main instance's response is entirely absent, the tracking instances initiate a new selection of the main instance. We suggest the use of the bully algorithm presented in section 3.9.4 in combination with a prior voting on the response for this process. The additional voting

step ensures that only a failed main instance triggers an election; failed tracking instances will disable themselves if they loose this initial voting.

Similar to the data topic of the replicated *Component*, tracking instances monitor all output channels of the main instance. If they ever notice a missing or wrong output, they initiate a reelection of the main instance.

Chapter 4

Implementation

4.1 Hardware Selection

A concrete hardware platform is necessary for an implementation and demonstration of the DOSIS framework. While a representative candidate for the OBDH of the CubeSat missions requires a certain radiation tolerance (see section 2.3), ground testing and development of control applications can be simplified and sped up with easy to handle and readily available evaluation platforms. Therefore, we suggest two different platforms for the overall demonstration and evaluation.

Three criteria provide the basic guideline for selection of this hardware platforms:

Design goal 1: The evaluation platforms should be readily available as COTS components including a ready-to-use evaluation board.

Design goal 3: The platform should be compatible with the power budget of a CubeSat.

Design goal 4: One of the platforms should be well suited for the operation in the radiation environment of a multi-year LEO mission.

Additionally, support of the target hardware platforms in RODOS should be possible. Therefore, a 32 bit platform should be selected, preferably using a core that was previously demonstrated in one of the readily available RODOS ports.

The following sections will present the two platforms used within this thesis: Section 4.1.1 presents the radiation tolerant VA41620 MCU and the corresponding evaluation board. The VA41620 is the suggested candidate to demonstrate suitability for the space environment of the LRSM missions. Section 4.1.2 introduces the readily available, simple to use, and cheap STM32L4 platform suggested for simplified development of software components. Note that the DOSIS framework and the suggested system is not bound to these specific hardware platforms.

4.1.1 VA41620 Platform

The VA41620 MCU is a radiation tolerant ARM Cortex-M4 based MCU with integrated single-precision floating-point unit by Vorago Technologies. The MCU is based on HARDSIL [227] — a CMOS based technology with increased radiation and high temperature tolerance [14] — and utilizes triple modular redundancy (TMR) for registers and bit error detection and correction for RAM and code memory [227]. This technology has been previously demonstrated on the ARM Cortex-M0 based VA10820 [15] and was successfully tested for SEE [230].

The ARM Cortex-M4 core used is already used for other ports of RODOS. Additionally, the VA41620 has embedded support for CAN and is thus compatible with the suggested CAN bus based network selected in section 3.2.4. The wide range of supported peripheral interfaces furthermore simplifies the use of the VA41620 for various applications on-board a CubeSat. Finally, the pricing of the VA41620 is reasonable for a CubeSat mission with enhanced reliability requirements and the component is readily

Table 4.1: VA41620 key features according to the data sheet [227].

Core	32 bit ARM Cortex M-4 with floating point unit
Clock rate	≤100 MHz
RAM	64 kB plus external bus interface for additional 16 MB
code memory	320 kB, loaded from external memory during hardware initialization
Peripherals	UART, I²C, SPI, CAN, Ethernet, SpaceWire, ADC/DAC, GPIO
TID tolerance	guaranteed $3 \cdot 10^3$ Gy
SEE tolerance	low soft error rate[i] due to built in error correction and memory scrubbing; guaranteed latch up immune for LET of up to $1.1 \cdot 10^5$ MeV cm^2 g^{-1}.

[i] $1 \cdot 10^{-15}$ errors per bit per day with 2.54 mm Al shielding in geosynchronous orbit during a solar minimum.

available[1]. Table 4.1 summarizes the key features of the VA41620 MCU.

Overall, the VA41620 is a reasonable candidate MCU for the ORIGINS LRSM missions' CDH subsystem. Within this thesis, we will use the VA41620 MCU to demonstrate the availability of hardware suitable for the DOSIS framework's operation in harsh environments such as the SAA.

A VA41620 port of RODOS was developed by Faehling [63] as part of a thesis supervised by the author of this thesis.

4.1.2 STM32L4 Platform

While the VA41620 is affordable for the LRSM CubeSats, a cheaper microcontroller provides a platform for software development, testing, and verification. The STM32L4 microcontroller series by ST Microelectronics provides such a platform. It uses a similar ARM Cortex-M4 core, includes a floating-point unit, and natively supports CAN. Additionally, it is a relatively cheap COTS part[2] readily available at the LRT. MCUs of the STM32L4 series are also used for the MOVE stratospheric balloons.

The similar architecture and the availability render the STM32L4 MCU a good platform for development and testing of software. Within this thesis, we will use the STM32L496zg MCU — the version with the largest RAM and program memory of the STM32L4 series [207] — for demonstration of software interfaces and a time synchronization demonstration. Table 4.2 provides an overview of the key features of this specific controller.

Table 4.2: STM32L496 key features according to the data sheet [206].

Core	32 bit ARM Cortex M-4 with floating point unit
Clock rate	≤80 MHz
RAM	320 kB
code memory	≤1 MB
Peripherals	USB, serial audio, I²C, UART, SPI, CAN, single wire, ADC/DAC, GPIO

4.2 DOSIS Framework Implementation

The implementation of the DOSIS framework consists of the framework's core and a collection of common features.

The DOSIS core provides the public API of the DOSIS framework consisting of three major parts: The DOSIS *Components* including their respective *ComponentInterfaces*, the DOSIS *Modules*, and a collection of types used within the interface of those *Components* and *Modules*. An additional implementation part contains all features used to implement the public interface. This part contains helper types, e.g.,

[1] Approx. € 3.700 e.g., at Mouser Electronics (https://www.mouser.de/)

[2] An evaluation board for an STM32L4 MCU is available for less than € 20, e.g., via Mouser Electronics.

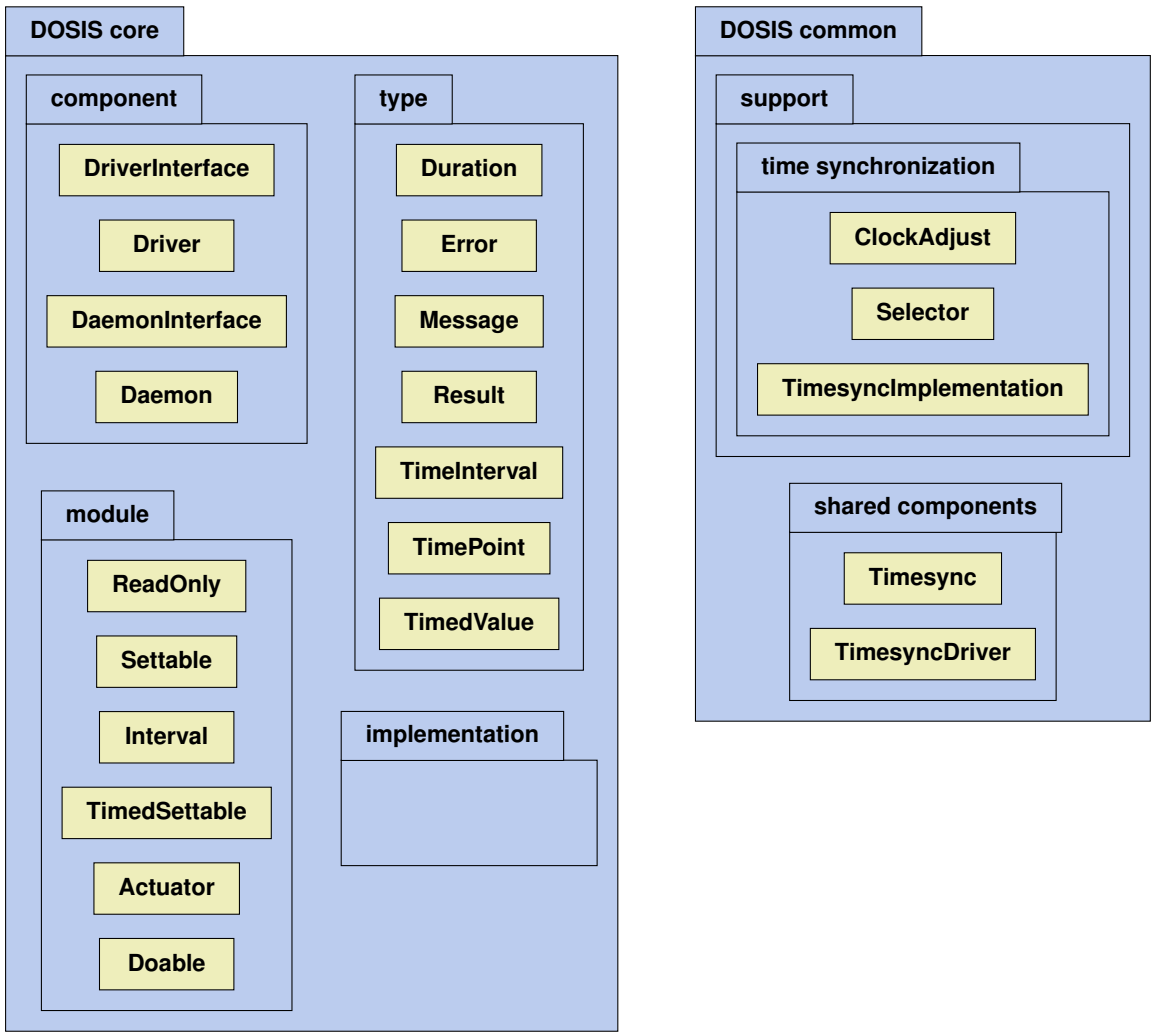


Figure 4.1: Overview over public parts of the DOSIS implementation and its split into DOSIS core and DOSIS common. This figure does not depict implementation internals, helper classes, and features other than the time synchronization in DOSIS common.

for internal message handling, and is not intended for direct use. The framework's core is maintained in a Git repository.

The common features contain support features such as the time synchronization and a collection of shared components. Out of these, this thesis focuses on the time synchronization only. A separate Git repository contains the DOSIS common features. Access to both Git repositories can be granted by the author of this thesis upon request[3].

Figure 4.1 depicts this separation into DOSIS core and DOSIS common, as well as their internal structure.

The following sections introduce the framework's implementation starting with the reasons for C++ template metaprogramming in section 4.2.1 followed by details about the implementation of DOSIS *Modules* in section 4.2.2. Section 4.2.3 presents details about the implementation of *ComponentInterfaces* followed by information on the *DriverImplementation* (section 4.2.4) and *DaemonImplementation* (section 4.2.5). Finally, the implementation of the DOSIS message handling is presented in section 4.2.6. Appendix D.1 presents the usage of the DOSIS framework on a simple example.

4.2.1 C++ Template Metaprogramming

Most parts of the DOSIS framework utilize C++ templates. These templates enable generic metaprogramming and the creation of a framework that is not bound to a specific data type. The DOSIS

[3] A release to a public repository is intended for the near future.

framework uses these features to enable strict typing of all parts of the framework while enabling the use of user-defined types. This enables a compile-time checked interface based on DOSIS *Modules* with pre-defined functionality bound to a specific type by the users of the framework. Similarly, a user of the framework can define a DOSIS *Component* based on a combination of DOSIS *Modules*. Finally, and in contrast to implementing similar features based on virtual classes, compile-time checks reduce the risk of runtime errors. The end user never directly accesses raw messages or buffers; instead interfaces provide a strictly typed and checked interface based on C++ templates. These templates avoid the need to implement a dedicated version of a method or function for every possible type.

4.2.2 Modules

The DOSIS core implementation contains all *Modules* as presented in section 3.5. A *Module* has at least two template parameters to specify the used data type and the identifying key within the context of a *Component*. The only exceptions are the *Doable Module*, which requires individual data types for requests and responses, and the *Config Module*, which requires all other *Modules* of the *Daemon* as template arguments instead of a data type. Each *Module* at least contains the identifying key and an inner class for its *ModuleInterface* and *ModuleHandler*. Additionally, *Modules* containing more than one internal message format provide C++ type definitions, deduced from its template parameter, for safe access to all messages.

A *Modules* class is a container only used at compile-time that provides a *Module's* key for compile-time usage. This is required for DOSIS *ComponentImplementation* and *ComponentInterface* implementations. Only the inner `Interface` and `Handler` classes are actually instantiated, provide runtime access to the key, and implement the *Modules* functionality.

Interface

A *Module's* `Interface` class provides all features to communicate with the corresponding *ModuleHandler* as specified in section 3.5 and figure 3.9. Therefore, it contains:

- A buffer to store the latest received value (or error). *Modules* with more than just a single value, e.g., an *Interval Module* that also contains an interval setting, also contain a buffer for these values. The buffers always store the latest value or error code received.
- Methods to synchronously and asynchronously access the data stored in these buffers. For each buffer, the interface provides a method to directly access the content if it is available (`getIfExists`) and a method to synchronously wait for updated data (`getSync`). The latter has an optional timeout parameter to abort the operation if nothing is received at all.
- Methods to request and/or set the value or setting. Similar to the access to local buffers, the interface contains a one shot version that does not wait for a response (`request` and `set`) and a synchronous version (`requestGetSync` and `setSync`) is available. In both versions, they clear the local buffer first. The synchronous versions are similar to calling the asynchronous version followed by a `getSync`.
- Methods to generate messages for this *Module*. These methods are mostly for internal use, but may be required for some subsystems that have to pragmatically assemble DOSIS messages. An example for such a subsystem would be the CDH commanding assembling arbitrary messages for other components.
- A method to access the identifying key of the *Module*.
- Callbacks to handle received messages (`handleValue` and `handleError`). As shown in figure 3.9, the *ComponentInterface* will call these methods (referenced as `store` in said figure).
- And type aliases for the *Module's* types, e.g., for its register, the message body, and the actual message used for communication between *ModuleInterface* and *ModuleHandler*.

A `GenericInterface` class provides most parts of the functionality regarding the primary value, a default implementation for the callbacks, and a number of internal helper methods to support the implementation of extended interfaces for additional functionality or parameters. Although the `GenericInterface` class's methods are not virtual, a specific `Interface` class may redefine the `handleValue` and `handleError` callbacks. Section 4.2.6 presents message handling and selection of appropriate callbacks in greater detail.

Handler

The `Handler` class is the counterpart of the `Interface` class. It is responsible for message generation on request or according to its own activities. Each *Module's* `Handler` contains at least:

- A register to store the current value. Optionally, a `Handler` may contain additional registers for parameters, e.g., the interval setting in an *Interval Module*.
- A `getRef` method to acquire a reference to the `Handler` instance's register. No such method exists for the optional registers for parameters. Instead, dedicated setter and getter methods provide access to these parameters; e.g., an *Interval Module's* `Handler` provides `setInterval` and `getInterval` methods to access the interval parameter.
- A method to access the identifying key of the *Module*.
- Methods to publish a data or error message and thus send the message to the corresponding *ModuleInterface*. Section 4.2.6 presents more details about the internal message handling.
- Methods to handle the modules internal activity within the context of a *Component*; i.e., to determine the next point in time for any internal activity of the `Handler` (`updateTriggerTime`), check if an activation is still pending (`needsActivation`), and trigger the execution of the activity (`triggerActivity`). The *Component* Implementation uses these methods to trigger a specific *Module's* activity at the appropriate time.
- Callbacks to handle received messages (`handle` and `handleConfig`). The `Handler` implements the message handling for all commands related to the internal parameters and forwards access to the data value to the corresponding user-defined `get` and `set` callbacks within the *ComponentImplementation*. Section 4.2.4 provides detailed information about these callbacks.
- A method to generate an error response for unhandled messages. Whenever a message cannot be handled within a *ComponentImplementation*, an appropriate error message is published instead. This may happen if the *ComponentImplementation's* queue of received messages is full. Instead of silently dropping a message, this method enables the `Handler` to notify the `Interface` appropriately.

Again, a `GenericHandler` provides implementations for common methods and a number of helper methods, which simplify the implementation of a specific *Module's* `Handler` class. In contrast to the `GenericInterface`, the `GenericHandler` does not provide a default implementation for callbacks.

Figure 4.2 depicts the public interface of the *Settable Module* including its inner `Interface` and `Handler` classes.

Limitations

The current implementation of the DOSIS framework limits the types possible as template arguments of *Modules*. Especially, it requires the types to be:

- default constructible,
- trivially copyable,
- and have an alignment requirement of a single byte.

Types must be default constructible to initialize registers and temporary values. Trivially copyable types enable to move the data to and from a message by simply copying the byte representation. The alignment requirement assures that no excessive padding is part of the data. This is an optimization to reduce the message size on the shared CAN bus and can be enforced for any type using the `[[gnu::packed]]` attribute.

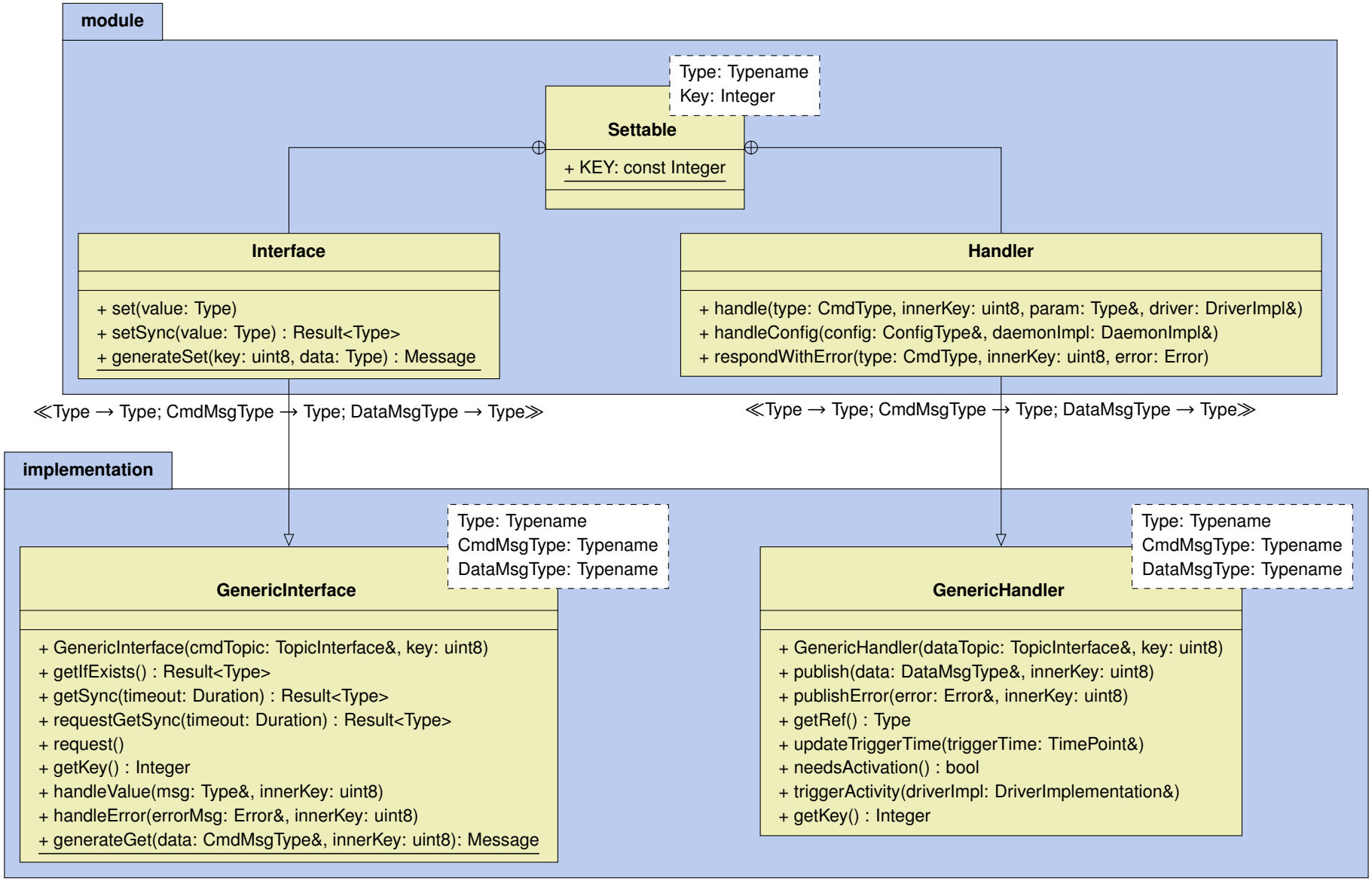


Figure 4.2: Class diagram of a DOSIS `settable` including the inner `Interface` and `Handler` classes as well as the `GenericHandler` and `GenericInterface`. For simplicity, type aliases, private members, protected members, and template parameters for methods are omitted.

4.2.3 ComponentInterface

As presented in section 3.4.1, a *ComponentInterface* is a combination of *Modules*. These are provided as C++ template parameter pack enabling an arbitrary number of *Modules*. Figure 4.3 depicts the `DriverInterface` and `DaemonInterface` classes.

A *ComponentInterface* mainly consists of:

- A constructor that expects a *deviceDef* and human-readable name for terminal output. The *deviceDef* identifies the associated data and command topics and binds an instance to said topics.
- A `get` method to access the associated *Modules'* `Interface` instances. This method returns a reference to said `Interface`. A type or interface finder deduces the type at compile-time from the `KEY` template argument.
- A `putGeneric` method as required for a RODOS `Putter`. This method handles all messages the internal subscriber receives on the associated data topic.
- A method to access the associated data and command topics.

Additionally, a *DaemonInterface* provides access to the automatically generated *Config Module*.

Limitations

The communication channel used within a *Component* accesses the associated *Modules* via an array using the *Modules'* identifying keys as index. This imposes some limitations on the way these keys and the *Modules* provided as template arguments:

- The identifying keys must be represented in an enumeration class named `Key`.
- This enumeration must not contain any gaps and start with zero.
- For *Daemons*, an additional key called `CONFIG` must be the last key in said enumeration.
- And the used `Module` classes must be provided as template arguments in order according to their respective keys.

Finally, a *Component* may only contain as many *Modules* as a DOSIS message (specified in section 3.6.3) can address. As an 8 bit field represents the key, 256 different *Modules* could be theoretically addressed. The implementation reserves the key of -1 (0xFF or 255 as unsigned integer) for the generic module types. Thus, a maximum of 255 *Modules* can be used for a single *Component*. This includes the internally generated *Config Module* in *Daemons*, thus only 254 user defined modules can be used in this case.

All of these requirements are verified at compile-time.

Access *Modules* by Key

Accessing a *Module's* `Interface` by its key using the `get<KEY>()` method requires a type deduction at compile-time. A `KeyTypeList` implements this type deduction.

Each *Module* contains a compile-time constant `KEY` with its respective identifier. The `KeyTypeList` provides a compile-time usable recursively implemented list of those *Modules*. Listing 4.1 depicts the `KeyTypeList` implementation. The inner `TypeFinder` class provides access to the Type for a specific key. The `TypeFinderImpl` class contains the actual implementation of this functionality. If the key searched for matches the identifying list of the first element (current `Head::KEY`), the `TypeFinderImpl` returns the type of said element. Otherwise, the result of the type finder applied to the remaining list (`KeyTypeList<Tail...>`) is returned. If no match is available, the compilation terminates with a compile-time error.

The `InterfaceChannel` is a child class of all *ModuleInterfaces* of the *ComponentInterface*. Thus, once the `TypeFinder` deduces the *Module's* type based on the key, the `InterfaceChannel` can be casted to said type. This way, the *ComponentInterface's* `get<KEY>()` method returns a statically type-checked reference to the `Interface` instance of the *Module* identified by the given key.

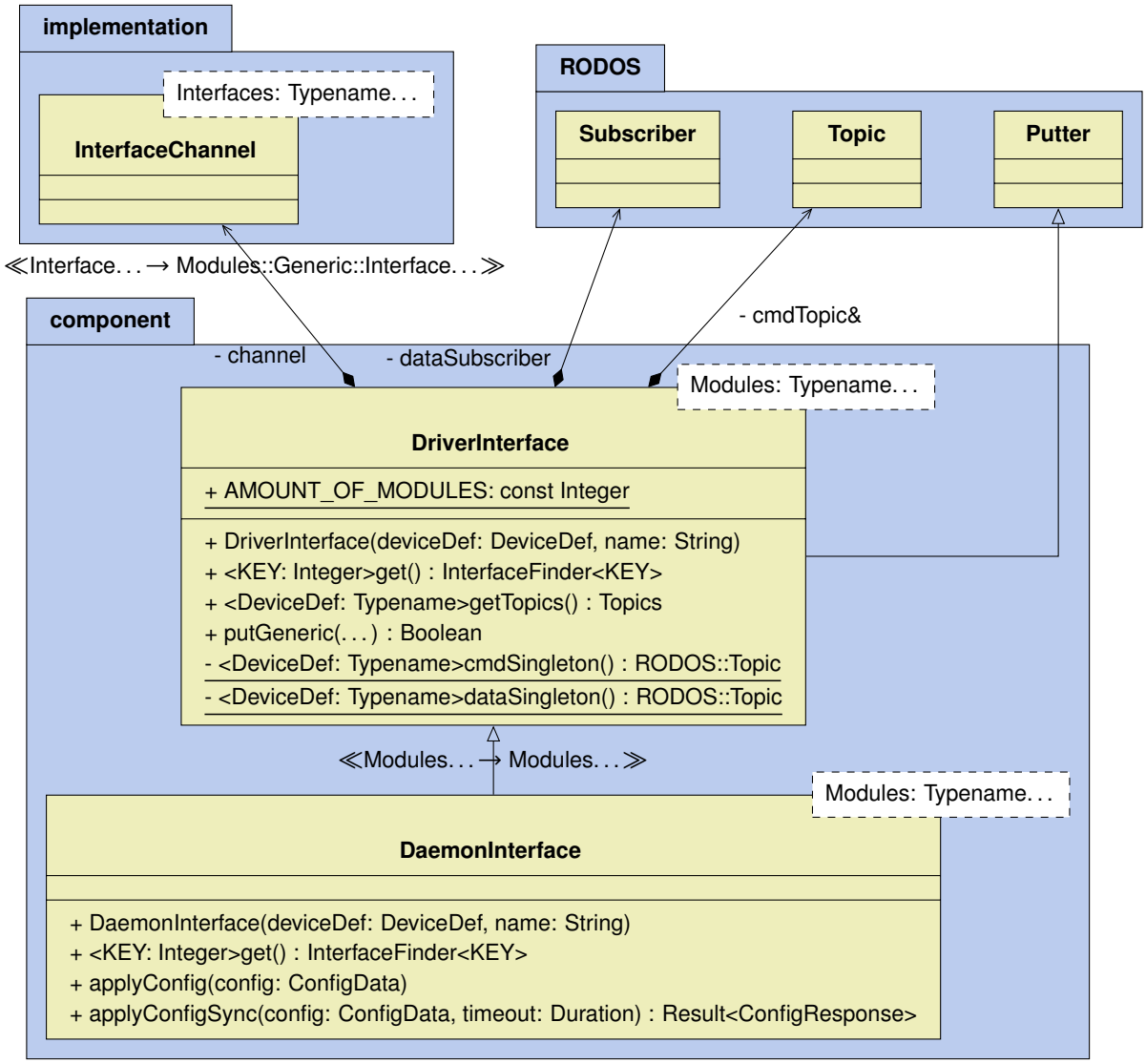


Figure 4.3: *DriverInterface* and *DaemonInterface* class diagram excluding type aliases. For simplicity, this figure simplifies RODOS types and internal implementation types and excludes type aliases. Section 4.2.6 presents more details about the `InterfaceChannel` class and its functionality.

```
template <typename... Type>
class KeyTypeList {
};

template <typename Head, typename... Tail>
class KeyTypeList<Head, Tail...> : public KeyTypeList<Tail...> {
  public:
  using EnumType = decltype(Head::KEY);
  using HeadType = Head;

  static constexpr int getHeadKey();
  static constexpr int getMaxKey();

  template <EnumType KEY, EnumType CUR>
  class TypeFinderImpl {
    public:
    using Type = typename KeyTypeList<Tail...>
      ::template TypeFinderImpl<KEY, KeyTypeList<Tail...>::HeadType::KEY>::Type;
  };

  template <EnumType KEY>
  class TypeFinderImpl<KEY, KEY> {
    public:
    using Type = Head;
  };

  template <EnumType KEY>
  class TypeFinder {
    public:
    using Type = typename TypeFinderImpl<KEY, Head::KEY>::Type;
  };
};
```

Listing 4.1: KeyTypeList definition excluding definition of static methods.

deviceDef

As presented in section 3.7, a *deviceDef* connects an instance of a *ComponentInterface* (or *ComponentImplementation*) to a pair of RODOS topics. The DOSIS framework implementation uses C++ classes containing only type aliases and static compile-time constants. Each *deviceDef* thus defines a separate class. Templated constructors of *ComponentInterfaces* and *ComponentImplementations* utilize said classes to prevent *deviceDefs* misuse and thus assure that the instances connect to the appropriate RODOS topics. The C++ concept `DeviceDefConcept` verifies that a *deviceDef* class contains all the required information. Additionally, this concept verifies that the used topic identifiers differ and are within the allowed range for RODOS topics.

While the `DeviceDefConcept` verifies some basic constraints, it cannot assure that a specific topic identifier is unique throughout the entire system. Therefore, the selection of appropriate topic identifiers is the user's responsibility.

Finally, the *deviceDef* also contains a type alias `InterfaceType` for the *ComponentInterface* type. The *ComponentInterface's* constructor verifies said type and thus assures that a *deviceDef* is only used for a specific *Component* and its respective *ComponentInterface*.

Topic Singleton

Each Topic must be initialized exactly once on every node using said topic. Therefore, the DOSIS framework must assure that only one instance of the command and data topics for a specific *Component* exists. As multiple instances of a *ComponentInterface* may exist within a single node, a singleton pattern assures that they reference the same topic instance and initialize it exactly once. The static methods `cmdSingleton` and `dataSingleton` implement this singleton within the `DriverInterface`.

4.2.4 Driver Implementation

A `DriverInterface` provides the necessary information for a `Driver` class. It provides information about available *Modules*, their respective keys, and the `TypeFinder` to access the *Modules'* `Handler` instances and all the related information. This `DriverInterface` is the first template parameter of a `Driver`. This way, we guarantee that the *Modules* and their respective keys of a specific `Driver` always match those of the corresponding `DriverInterface`. Similar to a `DriverInterface`, a *deviceDef* provided to the *Driver's* constructor connects the *Driver* to a pair of RODOS topics. Additionally, it assures that *deviceDef* was intended for this *Component* and thus avoids wrong connections between a *Component* and its *ComponentInterface* at compile-time.

The second template parameter is the user defined implementation class of the *Driver*, i.e., the actual implementation. The actual implementation of a *Driver* inherits from `Driver` and is provided as a template argument to the `Driver` class at the same time. This enables access to the user defined implementation from the generic `Driver` class, which is of special interest for the template methods used for callbacks.

The third template parameter of a `Driver` is the size of the first in, first out (FIFO) for received command messages. This parameter is optional and defaults to a size of five messages. Section 4.2.6 provides more information on the use of this FIFO. Figure 4.4 depicts the `Driver` class in detail.

Accessing a Module's Register

The `reg` method of a *Driver* provides access to the register of a specific *Module* identified by its key. This method utilizes the `TypeFinder` as previously presented and returns the reference provided by the *Module's* `getRef`. Due to the nature of a reference, this method enables read and update of a specific *Module's* register value.

To assure thread safe access to said register, a lock must be acquired prior to accessing the actual value. A *Driver's* `getLock` method provides access to this lock and returns a RODOS `ScopeProtector`. While this lock is active, the *Driver* pauses its activity other than enqueueing received commands. Thus, it should not be held for an extended time.

Callbacks

The callbacks as presented in section 3.4.2 are also part of the *DriverImplementation*:

At system initialization, RODOS calls the `init` method of all threads. As a DOSIS `Driver` is such a thread, this method can be directly overloaded. Thus, no additional modification point is required.

Additionally, a *Driver* contains modification points to get and set a specific *Module's* data. Three callbacks are available for this purpose:

getter Whenever a *Module* has to publish its data, it calls the *Driver's* `getter` callback. This callback returns a `Result` object containing an error code and the *Module's* data. The default behavior of the `getter` is to return the current content of the *Module's* register.

setter In addition to updating its register, a *Module* activates the `setter` callback whenever it handles a set command. This callback only returns an error code to indicate if the command was successful. By default, it returns an `Error::OK` indicating successful execution of the operation.

doer A *Doable Module* requires a different callback as it does not support individual get and set operations. Instead, it calls the `doer` callback on every do command. This callback expects the request as its argument and returns a `Result` object containing the response. In contrast to the `getter` and `setter` callbacks, no default implementation of the `doer` callback exists. Instead, a user defined version has to be provided for each *Doable* module.

Listing 4.2 shows the declaration of these callbacks and the default definitions of the `getter` and `setter` callbacks in a `Driver`. A user may redefine these callbacks for a specific key by providing a template specialization. Appendix D.1 presents how a user may modify them.

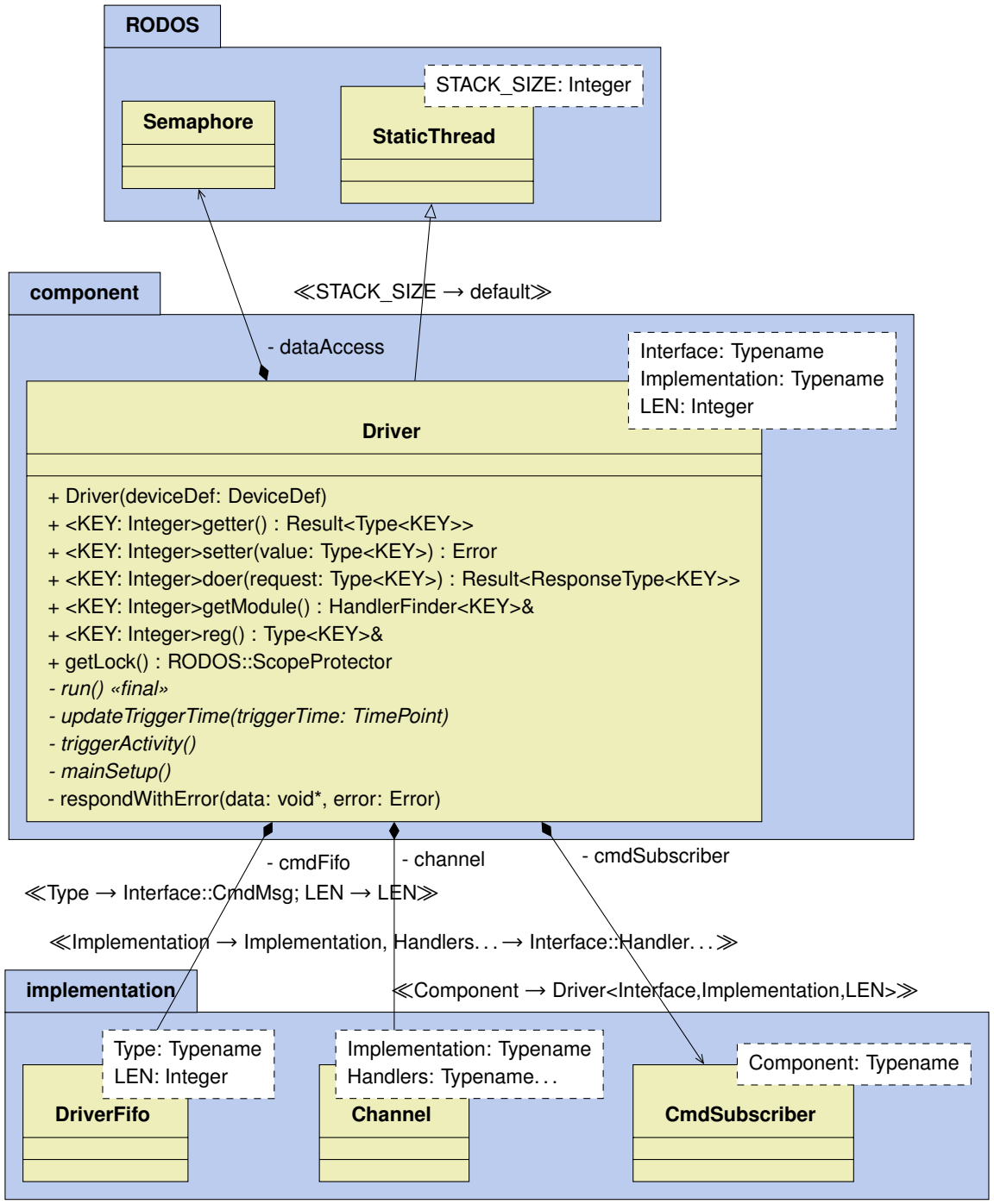


Figure 4.4: Simplified DOSIS `Driver` class diagram. For simplicity, this figure omits type aliases and types used in method signatures. Additionally, it simplifies some template parameters. The virtual private methods are not intended for direct use. Instead, they provide the interface internally required by the `Daemon` class presented in section 4.2.5.

```
template <auto KEY>
dosis::Result<Type<KEY>> getter()
{
    return { reg<KEY>(), dosis::Error::OK };
}

template <auto KEY>
dosis::Error setter([[maybe_unused]] const Type<KEY>& val)
{
    return dosis::Error::OK;
}

template <auto KEY>
dosis::Result<ResponseType<KEY>> doer(const Type<KEY>& request);
```

Listing 4.2: Getter, Setter, and Doer callbacks.

4.2.5 Daemon Implementation

The `Daemon` class provides the implementation of the DOSIS *Daemon*. It inherits all features from `Driver` and adds the additions according to section 3.4.2. Therefore, `Daemon` provides modification points for the *Daemon's* regular activation through virtual methods that a user may implement, methods to access the activation interval, and a way to specify if a specific *Daemon* requires an initial configuration message.

mainInit The `Daemon` activates this method after handling a configuration message. In contrast to the `init` method of a `Driver`, this may be called repeatedly and is delayed until a configuration message is received if the `Driver` requires an initial configuration message. The default implementation does not perform any operation. A user may overload this method.

mainStep The `Daemon` activates the `mainStep` method in a regular interval. The default implementation does not perform any operation. A user may overload this method.

set*Interval* The `setInterval` methods provides a way to modify the interval used for `mainStep` invocations. The initial value of the interval entirely disables the `mainStep` activations. Thus, a user should call this method either from `init`, `mainInit`, or an external location. It may also be called from within one of the `getter`, `setter`, or `doer` callbacks of the *Daemon*.

NEEDS_CONFIG The `NEEDS_CONFIG` constant is a static class member variable. If it is set to `true`, the *Daemon* delays activation of its `mainInit` and `mainStep` methods until it receives a configuration message. `NEEDS_CONFIG` defaults to `true`. A user may redefine it to `false`.

The handling of configuration messages is not part of the `Daemon` class. Instead, the *Daemon* contains a *Config Module*, which handles these messages. A `confirmConfig` callback within the `Daemon` class provides an interface for this module to notify the *Daemon* about an updated configuration.

4.2.6 DOSIS Message Handling

The communication between a *Component* and its *ComponentInterface* uses DOSIS messages as presented in section 3.6.3. The messages, originally transmitted via RODOS topics, are handled on a higher level that removes the burden of parsing raw bytes from users of the framework. The following section presents the C++ representation, identification, and forwarding of these messages to the respective *ModuleInterface* or *ModuleHandler*, and the general steps of the message handling process.

DOSIS Message

The `Message` class represents the DOSIS messages in C++. It provides access to access and modify the individual header fields. Additionally, it provides a template method to access the payload without

direct access to the raw buffer. Instead of raw bytes, it uses the target type for all accesses. While this hides type casts from the user, it is still the user's responsibility to only read data from a message with the type used to store said data. Other accesses may result in undefined behavior. Figure 4.5 depicts the `Message` class.

Channel and InterfaceChannel

The `Channel` and `InterfaceChannel` classes implement most of the logic to forward messages and similar calls to the appropriate *Module*. While the `Channel` class takes care of message forwarding in a *Component* to the respective *ModuleHandler*, the `InterfaceChannel` forwards messages in a *ComponentInterface* to the respective *ModuleInterface*. As the implementation details are similar, this section will focus on the `InterfaceChannel` and its capability to forward a received message to the appropriate `Interface` instance's `put` method. Nevertheless, the same mechanism also applies to a `Channel` class forwarding messages to the *ModuleHandler's* `handle` method and enables direct error replies to received messages. Figure 4.6 depicts the class diagram of the `InterfaceChannel` class. Figure D.1 in appendix D depicts the class diagram of the `Channel` class respectively.

As no type and method inference based on a key only available at runtime exists in C++, the DOSIS framework uses a custom mechanism instead. This mechanism forwards calls at runtime in three steps: First, a C-style array provides access to a method with the key as one of its template parameters. Afterward, this method performs the required type casting. Finally, it forwards the call to the respective target.

Redirect Call through Array of Function Pointers A C-style array of function pointers to a static template method with different template arguments identifies the appropriate forwarding method for a given message. At index *k* the array contains a pointer to the method with the `Interface` classe's type of the *Module* with key *k* as its template argument. As this type includes the *Module's* key, it enables a unique mapping even with multiple instances of a single *Module* only differentiated by their identifying key combined into a single *Component*. Although the referenced methods are static, i.e., only refer to the class but not a specific object, their first argument is a reference to a `InterfaceChannel` object. This enables a behavior similar to a non-static method while still enabling a call based on a regular function pointer. Overall, the array of function pointers to those methods provides a similar feature as virtual tables provide for hierarchies of virtual classes.

As specific versions of template methods are only instantiated during compilation if they are actually used, the array must be filled at compile-time. Listing 4.3 depicts the declaration of the `InterfaceChannel::putGeneric` method and the C-style array of function pointers initialized at compile-time.

```
1 template <typename Buffer>
2 static void put(InterfaceChannel& channel, DataMsg& dataMsg);
3
4 static constexpr void (*const PUTTERS[])(InterfaceChannel&, DataMsg&) = {
5     InterfaceChannel::put<Interfaces>...
6 };
```

Listing 4.3: Declaration of the `InterfaceChannel` class's `put` method and the array of pointers to forward received messages.

To assure the mapping of identifying key to array index in this implementation, the order of `Interfaces` is essential. These must be provided in the exact order of their respective keys. A static assertion assures this order as one of the limitations to the *ComponentInterfaces* as stated in section 4.2.3.

Accessing the Interface The `InterfaceChannel` inherits from all the *Modules'* `Interface` classes. Thus, a simple type-cast to the desired `Interface` class's type enables access to said `Interface`. As this type is the template argument of the static method called indirectly via the previously mentioned array,

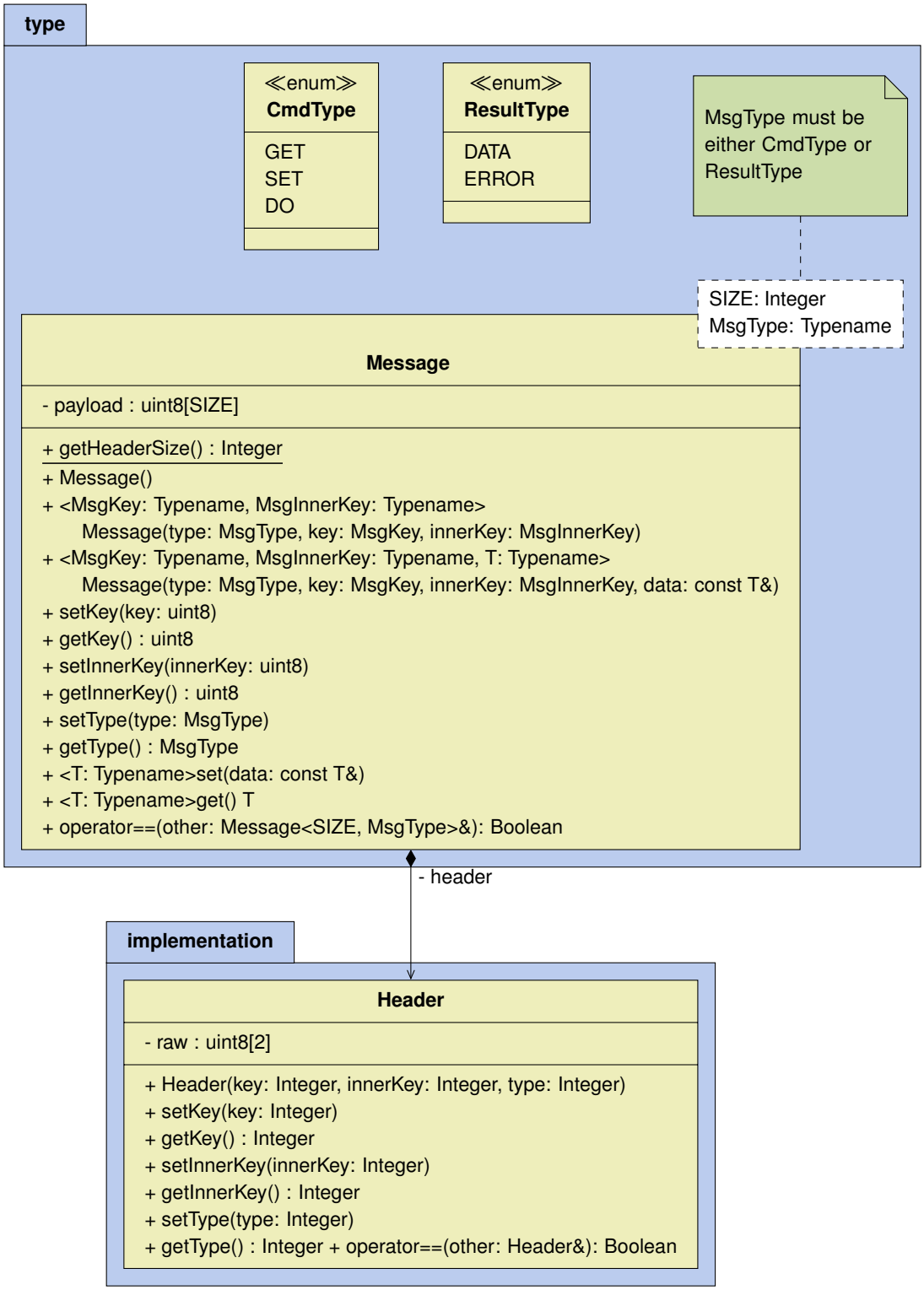


Figure 4.5: Class diagram of the DOSIS `Message`.

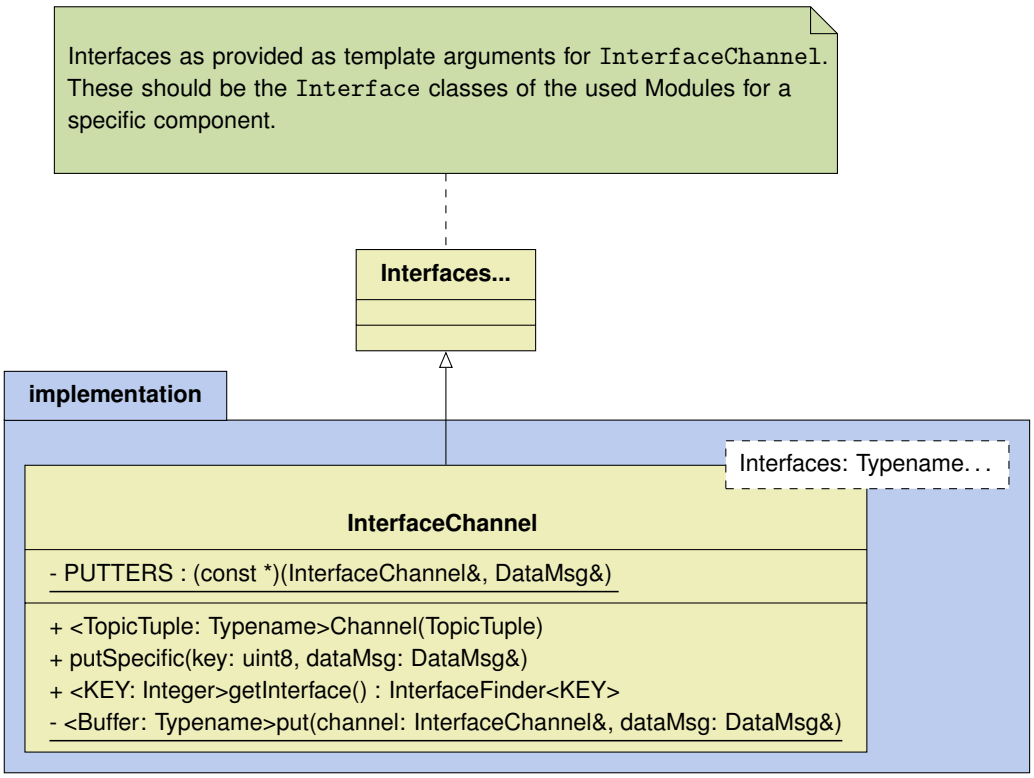


Figure 4.6: Class Diagram for a DOSIS `InterfaceChannel`. `InterfaceChannel` inherits from its variadic template parameter `Interfaces`.

the type of this `Interface` in a specific instance of said method is known at compile-time. Therefore, a simple `static_cast` enables the access to the desired `Interface`.

Forwarding Calls Once the `Interface` class can be directly accessed, the call can be forwarded using the designated methods within said `Interface`. While the `InterfaceChannel` class's `put` differentiates messages based on their type and forwards calls to either `handleValue` or `handleError`, the `Channel` does not further differentiate based on content but directly forwards the call to the `Handler` class's `handle` or `respondWithError` methods.

Sending Messages from a ComponentInterface

ModuleInterfaces directly handle sending of messages. In most cases, a *Module's* `Interface` class uses generator methods, e.g., `generateGet` or `generateSet` as shown in figure 4.2, to uniformly assemble its messages. The `Interface` classes contain a reference to the *Component's* command topic. Thus, they directly publish the generated message onto said topic.

Receiving Messages in a ComponentImplementation

A *ComponentImplementation* receives and handles messages in two separate steps: Receiving the message from a RODOS topic, and actually processing the message.

Receiving a Message A `CmdSubscriber` within the *Component* subscribes to the command topic. Thus, RODOS will forward all messages on said topic to a specific method of this subscriber. The `CmdSubscriber` stores the received message into a `DriverFifo`, which implements a simple FIFO. If this FIFO cannot store an additional message, the `CmdSubscriber` automatically replies with an appropriate error message. The `Channel` class's `respondWithError` method resolves the `Handler` instance that would normally handle this message. As the `Handle` class contains a reference to the data topic, it can directly generate and transmit an appropriate error message.

Processing a Message A *Component's* thread waits for messages in the `DriverFifo`. Whenever a message is available, it pops the message from this FIFO and triggers the message processing. The `Channel` class's `handle` method resolves the appropriate `Handler` instance based on the message's key and finally forwards it to the `Handler` class's `handle` method. The `Handler` class's `handle` method decides how to process the message based on the inner key and either internally handles the message (for parameters) or updates its register and calls the appropriate `setter`, `getter`, or `doer` callbacks. Afterward, the `Handler` generates and publishes a response to the data topic based on the returned value from these callbacks or its updated state (for parameter updates).

Sending Messages from a ComponentImplementation

In addition to the messages a `Handler` generates in response to received messages, a user or internal activity of a *Module*[4] may also send messages at arbitrary times. For this purpose, the *Driver's* `getModule` method[5] returns a reference to a *Module's* `Handler` instance identified by its key. The `Handler` in turn provides the `publish` and `publishError` methods that assemble a valid message and publish it to the data topic associated with the *Component*.

Receiving Messages in a ComponentInterface

In contrast to messages received in a *ComponentImplementation*, the *ComponentInterface* does not buffer the messages in a FIFO for later processing. Instead, messages received are directly handled by the `DriverInterface` provided `putGeneric` method. It forwards the call to the appropriate *ModuleInterface* using the `InterfaceChannel` class's `putGeneric` method previously presented. The *ModuleInterface* simply stores the content of the received messages. A user may later on retrieve this message via the `Interface` class's public interface.

4.3 DOSIS Time Handling

RODOS does not provide an interface to adapt the system clock used for scheduling purposes. Instead, it directly uses a hardware timer extended into a 64 bit integer representation of the local node's uptime in ns. To enable time synchronization, the DOSIS framework contains an internal time model. The following sections present the implementation of this time model, its limitations, and the available time synchronization implementations.

4.3.1 The DOSIS Time Model

The DOSIS time representation consists of two classes: `TimePoint` representing an absolute point in time, and `Duration` representing a certain time span. Their internal representation uses signed 64 bit integers representing the time in steps of 1 ns. To reduce the risk of misinterpretation, no conversion between `TimePoint`, `Duration`, and their respective integer representation exists. Instead, arithmetic operations and user defined C++ literals enable a user-friendly handling of the DOSIS time representation. A `TimeModel` class additionally provides the conversion to and from RODOS time representation for both DOSIS time types.

The `TimeModel` uses a linear time model with the RODOS uptime as uncorrected local clock. The `TimeModel` converts RODOS time representation into DOSIS time types and vice-versa using a section-wise defined linear model with parameters for skew and offset compensation. Listing 4.4 shows the simplified conversion methods. The variables `offset`, `skew`, and `syncTime` are the internal data of the `TimeModel`. The listing omits thread-safe access to those variables, boundary checks of input parameters, and type conversion operations for a better visualization of the conversion operation itself.

The internal state of the `TimeModel`, i.e., the `offset`, `skew`, and `syncTime`, are stored in a compound data structure, which is accessed via a pointer. Updating these values generates a new set of these

[4] E.g., an *Interval Module* regularly triggers the `getter` callback and transmits the respective data message.
[5] This method is also available for *Daemons*.

```
1  TimePoint rodosTimeToTimePoint(int64_t timePoint)
2  {
3    return timePoint + offset + skew * (timePoint - syncTime);
4  }
5
6  Duration rodosTimeToDuration(int64_t duration)
7  {
8    return duration * (1 + skew);
9  }
10
11 int64_t toRodosTime(TimePoint timePoint)
12 {
13   return (timePoint - offset + skew * syncTime) / (1 + skew);
14 }
15
16 int64_t toRodosTime(Duration duration)
17 {
18   return duration / (1 + skew);
19 }
```

Listing 4.4: RODOS to DOSIS and DOSIS to RODOS time conversion. Boundary condition checks, type conversion, and atomic access to `TimeModel` parameters omited for demonstration purposes.

values and swaps the pointer in an atomic operation as final step. The methods for time conversion will store a local copy of the pointer used for their calculation. This assures that they use the same internal state throughout the entire conversion operation.

The methods to set new `offset` or `skew` parameters are not multi-thread safe. Instead, the current implementation assumes that on a node only a single time synchronization mechanism is active at a time. Overall, the buffer for the internal state is safe to use with multiple readers and a single writer.

Time Usage in Components

All DOSIS *Components* and *Modules* only use the DOSIS `TimePoint` and `Duration` to represent time related values. Only when interacting with the scheduler, they convert the time to the required RODOS representation. To avoid the accumulation of errors when using intervals, these are not converted by incrementing the RODOS time representation accordingly. Instead, a new absolute time point represented by a DOSIS `TimePoint` is calculated based on an initial `TimePoint` and a `Duration`. Using this method for all repeatedly activated *Modules*, the *ComponentImplementation* calculates the next absolute point in time for any internal activity. Afterward, it waits for incoming messages until this point in time is reached. As this involves yielding until data is available, the time is converted to a RODOS time suitable for scheduling purposes at this point. If the *ComponentImplementation* receives a message, it handles the message and repeats the process. If the point in time for the next internal activity is reached beforehand, the *ComponentImplementation* triggers the internal activity and repeats the process for the next activation.

Limitations of Implementation

While this implementation yields good results for short time periods, a long time forward propagation may result in inaccuracies due to changes of the `TimeModel` instance's parameters during this period of time. Therefore, we advise not to use extended periods for time critical activities. If scheduling an activity a long time beforehand is necessary, separating the wait time into shorter intervals reduces the inaccuracies. Repeatedly resuming the corresponding *Component* during the extended wait time will reduce the inaccuracies, as the *ComponentImplementation* repeats the conversion from DOSIS to RODOS time during each activation using the updated time model parameters.

A modification of the RODOS time model could entirely mitigate this issue. Although it potentially solves the issue, modifying the scheduling process requires special care not to break other critical parts

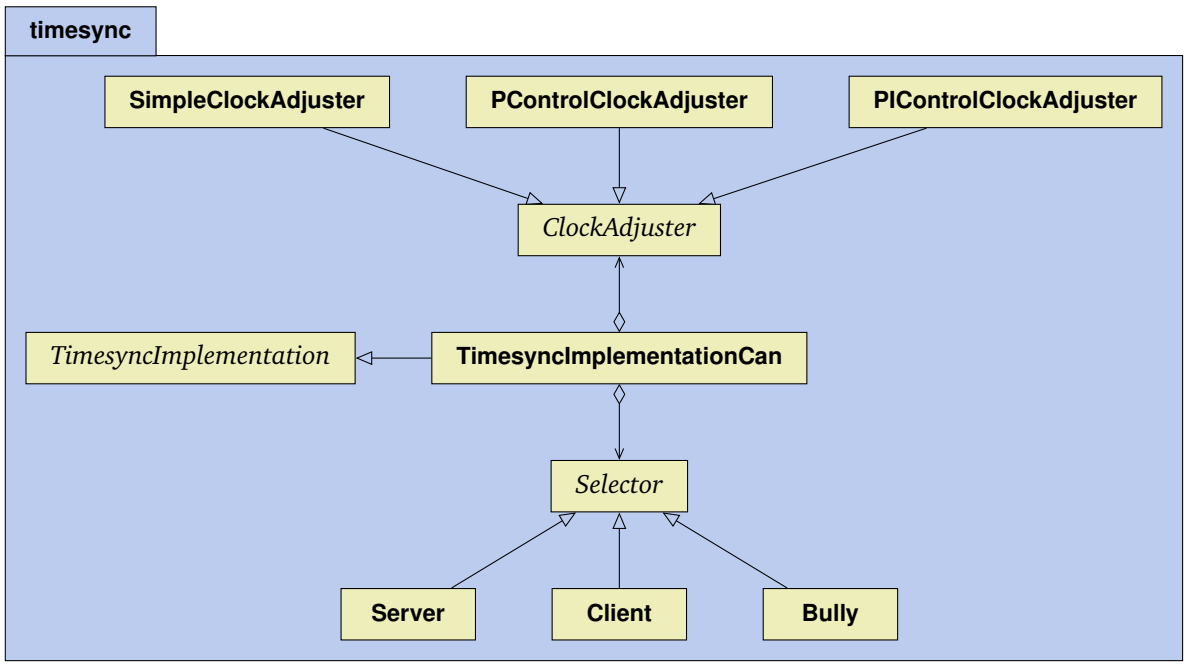


Figure 4.7: Overview of time synchronization classes.

of the system. As the current missions did not show any requirement for a stable long-term forward propagation, no change to the RODOS scheduling process is advised.

4.3.2 Time Synchronization Implementation

The time synchronization presented in section 3.8 is modular and consists of three main parts: selection of a reference node, transmitting the time, and update of the local clock. The `Selector`, `TimesyncImplementation`, and `ClockAdjuster` classes implement these parts. These classes are purely virtual and provide an interface to enable arbitrary combinations and thus to provide the flexibility to adapt the time synchronization to the needs of a specific mission. Out of these, the `TimesyncImplementation` not only implements the time transfer but also contains the connection logic of the three parts. I.e., the `TimesyncImplementation` uses the `Selector` to decide if it must behave as reference node and forwards all received time update messages to a `ClockAdjuster`. The DOSIS framework provides a CAN based implementation of the time synchronization. The other options suggested in section 3.8 were discarded due to their bad performance[6]. Figure 4.7 depicts these classes and the available realizations.

The time synchronization implementation uses the DOSIS `TimePoint` as its time representation. It uses a 64 bit signed integer format representing the time in ns since a user defined starting date.

Reference Node Selection

A `Selector` contains a method to check if the current node is the reference node and a method to update the decision, i.e., reevaluate the decision.

The DOSIS framework contains three implementations of a `Selector`. The `Server` and `Client` provide a statically defined behavior; the `Server` will always act as reference node, whereas the `Client` will never become a reference node. The `Bully` selector provides a dynamic reference-node selection according to the simplified bully presented in section 3.9.4.

The bully implementation's core is a state machine. It handles received messages based on the current state and the type of the received messages. Handling of multiple concurrent activities is possible as all wait statements from algorithms 3.1 to 3.4 are implemented as asynchronous wait. Thus, it handles a received message according to the respective algorithm; once it executes a wait statement,

[6]Chapter 5 presents more information on the performance of the various options.

it will instead start a timer and change the state. Once the timer activates, the wait time has finished and the algorithm continues. All instances communicate via a single RODOS topic. This ensures that the instances do not require a priori knowledge about the availability of other nodes.

Clock Update implementation

The DOSIS framework implements all time update versions suggested in section 3.8.4. Specifically, it implements a direct-set, P-controlled, and PI-controlled clock update mechanism. While the direct-set update modifies the `offset` parameter of the `TimeModel`, the other options modify the `skew` parameter.

Chapter 5

Time Synchronization Test

5.1 Time Synchronization Mechanisms

A first comparison of the time synchronization mechanisms proposed in section 3.8 as part of the author's own publication "Distributed Computing for Modular & Reliable Nanosatellites" [184]. This section presents these tests and provides a first insight into the performance of the different options and the overall system performance for a simple distributed control setup. Instead of the time calculations presented in section 4.3.2, these tests use a direct modification of the RODOS time representation and scheduling process.

5.1.1 Test Setup

A prototype consisting of three identical Nucleo-L496ZG evaluation boards provides the basic test infrastructure. A CAN bus using a transceiver-less setup [12] interconnects these nodes. Additionally, each node has a dedicated input and output general purpose input/output (GPIO) pin for external triggers and measurements. A control computer uses the integrated programmer of the evaluation boards to trigger individual test runs and record the UART output of all nodes.

Load Generator

A CAN load generator simulates the worst-case scenarios regarding the bus utilization. A RODOS thread implements this load generator and continuously publishes messages of the maximum allowed size (1300 B). Therefore, it simulates an almost 100 % bus utilization on the CAN bus. The used RODOS topic has the highest topic identifier in use. This topic identifier translates to the highest CAN identifier within the prototype. Thus, it has the lowest priority on the CAN bus.

Time Synchronization

The time synchronization test compares the pairwise time difference of all nodes. Therefore, it repeatedly records the local time on each node at an externally triggered time. For this purpose, an additional STM32F407G discovery provides a trigger signal. This trigger signal toggles its state every 100 ms and is directly connected to the input pin on all nodes. The input pin on each node triggers an interrupt on every state change of the trigger signal. The interrupt routine stores the current time. An asynchronous process prints the recorded time to UART, which the test control computer logs for later analysis. Figure 5.1 shows the full test setup.

A full comparison of all time synchronization options presented in section 3.8 is enabled by a set of individual tests. The test compares the time transfer based on

- DOSIS messages,
- RODOS messages,
- and raw CAN messages

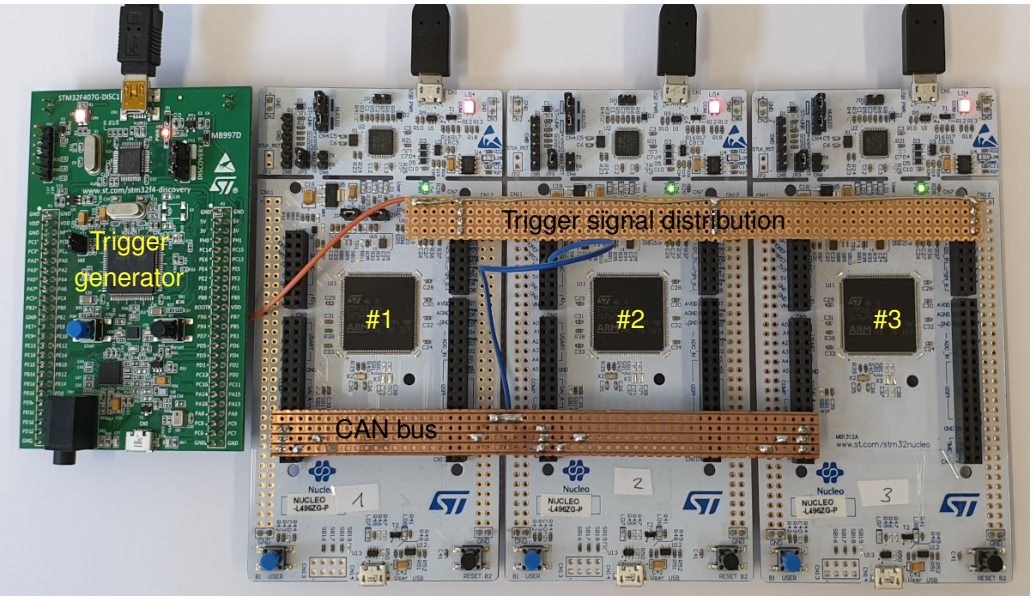


Figure 5.1: Experimental setup with three STM32L4 test nodes and an STM32F4 trigger generator node.

in combination with

- direct-set,
- and PI-controlled

clock update[1]. To account for the varying effects of the CAN load origin, the load generator is an optional part of the clock reference node's or one of the remaining nodes' firmware.

An individual test run is conducted for each combination of these mechanisms and the CAN load origin. The time synchronization interval, i.e., the timespan between consecutive updates, is 1 s for all test runs. The internal oscillator of all nodes is the reference oscillator for the local clock. These represent a reasonable worst-case scenario with a long time clock rate error of ∼1 % and a significant drift over temperature and input voltage [206]. The long time clock rate difference of the used MCU's local clocks was ∼0.5 %, the short time deviation was considerably higher.

Control Loop Setup

A simple control setup consisting of a sensor, controller, and actuator node demonstrates the capabilities of timely execution of the DOSIS framework. It measures the relative error compared to the expected actuation time point. The sensor and actuator nodes toggle their output GPIO pin whenever they are activated. External monitoring of these pins enables a comparison of the timespan between sensor readout and actuator actuation.

Sensor The sensor is a simple DOSIS *Driver* consisting of a single *Interval Module* representing the current state. Instead of actually sensing an external value, it inverts and returns the state of an internal boolean value. At the same time, the sensor sets the state of its GPIO pin accordingly and thus enables external observation. The *Interval Module* generates a readout every 500 ms in all test cases.

Actuator The actuator is also a simple DOSIS *Driver*. It consists of a single *Actuator Module* that enables direct and timed actuation. Similar to the sensor, it sets the state of its GPIO pin according to the commanded actuation. The `set` callback implements this update, thus timed actuation only affects the GPIO pin once it reaches the specified point in time.

[1] In contrast to the presented clock update in section 4.3.2, the experimental test setup uses a modified version of the RODOS local time model. The modified version contains a global offset and drift parameter as part of the time model used within the RODOS scheduling process.

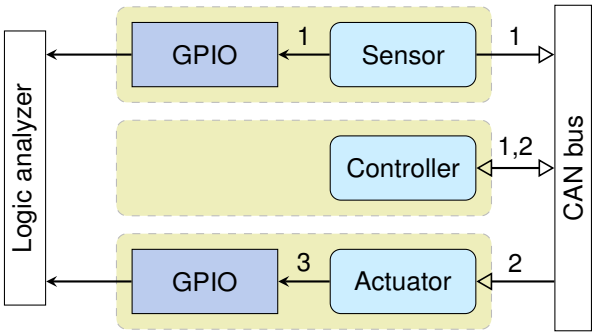


Figure 5.2: Control loop setup for time synchronization tests. Every 500 ms, the sensor generates a readout and toggles the corresponding GPIO pin (1). The controller receives the sensor readout and commands the actuator (2). Finally, the actuator toggles its corresponding GPIO pin after executing the command (3).

Controller The controller's only task is forwarding the received sensor data to the actuator. It knows two modes of operation: Either it directly forwards received data to the actuator for immediate actuation, or it schedules the actuation to 100 ms after sensor readout.

Used Time Synchronization A raw CAN based time transfer in combination with a PI-controlled clock update provides the time synchronization of the three nodes for all control loop tests.

External Observer A Saleae Logic Pro 16 logic analyzer monitors the GPIO pins of the sensor and actuator nodes at a sample rate of 1 MHz. Offline analysis of the recorded state changes of the GPIO pins provides the delay between sensor readout and actuator activity.

Similar to the plain time synchronization tests, a simulated CAN load originating from different parts of the system is part of the test scenarios. The CAN load for direct and scheduled actuation tests originates from:

- no part of the setup,
- the sensor node,
- the controller node,
- or the actuator node.

Figure 5.2 depicts the test components and their interconnections.

5.1.2 Time Synchronization Test Results

Each of the three basic time synchronization mechanisms was evaluated in combination with both time update mechanisms and different CAN load conditions. Table 5.1 lists the performed tests.

Each recording started 30 s after loading the firmware into the MCUs. Thus, the recordings do not include the initial synchronization that contains a hard set of the local time. Afterward, 180 s were recorded for each individual test run. Figure 5.3 depicts the distribution of the pairwise time deviation as a measure for the quality of the synchronization.

5.1.3 Control Loop Test Results

The control loop setup verifies the usability of the proposed system for distributed control applications.

Figure 5.4 depicts the distribution of the delay of the setup with immediate actuation for different CAN load scenarios over 350 samples. The sensor generates new data every 500 ms. The controller receives and forwards this data to the actuator for immediate execution. The actuator updates its output without any additional delay.

Table 5.1: Test cases for time synchronization evaluation for different clock transfer mechanisms, clock update mechanisms, and CAN load sources.

Id	Clock Transfer	Clock Update	CAN Load Source
a	DOSIS message based	direct-set	none
b	DOSIS message based	direct-set	client side
c	DOSIS message based	direct-set	server side
d	DOSIS message based	PI control	none
e	DOSIS message based	PI control	client side
f	DOSIS message based	PI control	server side
g	RODOS message based	direct-set	none
h	RODOS message based	direct-set	client side
i	RODOS message based	direct-set	server side
j	RODOS message based	PI control	none
k	RODOS message based	PI control	client side
l	RODOS message based	PI control	server side
m	CAN message based	direct-set	none
n	CAN message based	direct-set	client side
o	CAN message based	direct-set	server side
p	CAN message based	PI control	none
q	CAN message based	PI control	client side
r	CAN message based	PI control	server side

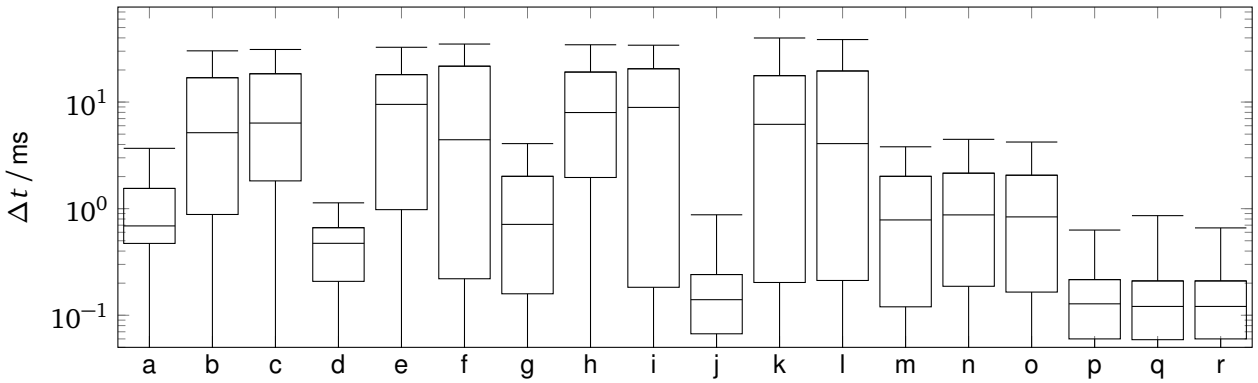


Figure 5.3: Pairwise time deviation for different time synchronization setups. The distribution over 180 s sampled every 100 ms is shown for each setup. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile. Table 5.1 lists the used clock transfer method, clock update mechanism, and CAN load source for test runs (a)–(r).

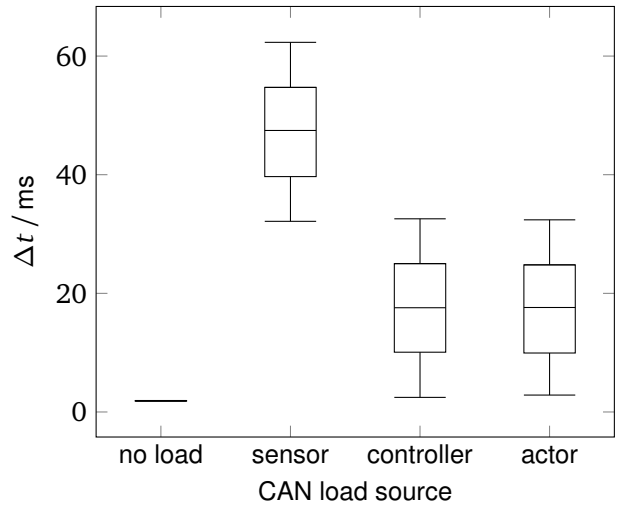


Figure 5.4: Distribution of 350 measurements of delay between sensor and actuator activity. All actions are executed immediately on arrival. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

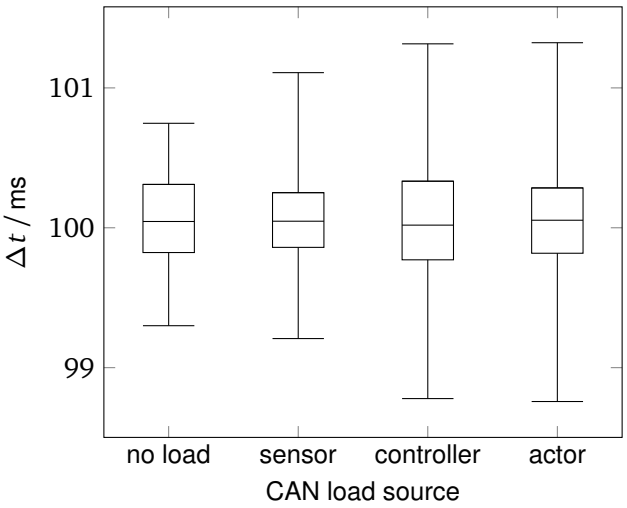


Figure 5.5: Distribution of 350 measurements of delay between sensor and actuator activity. The actuator is scheduled to activate 100 ms after the sensor readout. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

Figure 5.5 depicts the distribution of the delay of a setup with execution scheduled 100 ms after sensor readout over 350 samples. Again, it shows the different CAN load scenarios separately.

5.1.4 Time Synchronization Test Discussion

The relative error of DOSIS message or RODOS message based time transfer setups without load on the CAN bus is in the order of a few ms (test cases a, d, g, and j). These tests also show the advantage of PI-controlled clock update (test cases d and j). While the remaining error of about 1 ms for these setups are acceptable for most scenarios, the performance degrades rapidly once the CAN bus utilization rises (test cases b, c, e, f, h, i, k, and l). In those cases, a deviation of the local times of up to 40 ms can be observed. This is an expected consequence of the RODOS message handling over CAN. It forwards messages to higher layers in the order of their first CAN frame, which, as already stated in section 3.8.3, leads to large delays. These DOSIS or RODOS message based tests do not show a significant difference between direct-set and PI-controlled time update mechanisms under the influence of high CAN utilization. This can be explained as the expected offset after a 1 s time interval with a relative clock error of 0.5 % only leads to a time difference of 5 ms for a direct-set clock update. This is relatively small compared to the message transfer delay uncertainty and thus does not significantly change the result.

Using RODOS messages directly has no major impact compared to DOSIS messages. This can be explained by the fact that the expected additional software delay is small compared to the CAN delay and the error introduced by the local clocks themselves.

The direct usage of CAN circumvents this drawback of the RODOS message handling. Thus, the CAN load does not significantly influence the CAN message based approach (test cases m and p without CAN load, test cases n, o, q, and r with CAN load).

The main observable difference between the direct CAN based time synchronization setups is due to the selected clock update mechanism. The direct-set update suffers from the 5 ms error due to the different drift of the local clocks. The PI-controlled clock update mechanism on the other hand is capable of correcting the drift. This matches the expectations from section 3.8.4. The remaining error for a CAN based time synchronization with PI-controlled clock update mechanism thus stays below 1 ms.

Section 3.8 presents the limit for the remaining time synchronization error of less than 2 ms. Clearly, the CAN based time synchronization with PI-controlled clock update and the highest CAN priority for its own messages achieves this goal. Thus, the general time synchronization of the DOSIS framework is suitable for time-critical distributed control applications such as an ADCS system.

5.1.5 Control Loop Test Discussion

In the absence of additional load on the CAN bus, the direct actuation in a sensor-controller-actuator is as low as 2 ms. While this may be acceptable for some control applications, its performance significantly decreases with increased CAN bus utilization.

The main reason for this decreased performance is the RODOS message handling, as explained in section 3.8, which only forwards complete messages. RODOS processes messages interleaved on the CAN bus in order of their first respective CAN frame. In the worst-case, a message of 1300 B has just started and is currently in transmission. Including inter-frame spacing and bit-stuffing, such a message leads to approximately 30 ms delay independent of the CAN priority of the conflicting messages. Note that this only affects nodes receiving the unrelated CAN traffic. The node that originates the conflicting message receives the desired message without additional delay.

Therefore, a setup with CAN load originating from the sensor node experiences the worst-case delay. In this case, two message transmissions can be severely influenced by the generated traffic: Both, controller and actuator, have to wait for an ongoing artificial traffic message to complete before receiving and processing the actual data message. In this case, the expected worst-case delay is approximately two times the delay due to one 1300 B message, i.e., about 60 ms. If the traffic originates from the controller node, only the actuator has to wait (and vice versa). In this case the impact is only

about half as severe compared to the case with load generated at the sensor node. Figure 5.4 depicts these scenarios and clearly shows the expected worst-case delays of 60 ms and 30 ms respectively.

For scheduled actuation, we expect no timing uncertainty significantly larger than the remaining time synchronization error. The controller schedules the actuation at 100 ms after the initial sensor readout. As the worst-case delay of message propagation within the experimental setup is approximately 60 ms, the actuator receives the command before the scheduled point in time is reached. In this case, the main influence on the actuator's timing is the time synchronization itself and the concurrent processes and interrupts within the actuating node. Figure 5.5 depicts the behavior of the scheduled actuation. The measured delay between sensor readout and actuator activity matches the expected 100 ms with an error of less than ± 2 ms for all CAN load scenarios.

While the tests did show the impact of CAN traffic on direct actuation, they also verify that scheduled actuation is reliable in all tested scenarios. Therefore, not only the time synchronization itself, but also DOSIS *Components* are suitable for time critical control applications. A system designer must still take into account that the scheduled point in time is further in the future than the worst-case message transmission delay. This delay depends on the largest messages in a certain system and the number of hops required for a specific distributed controller.

5.2 Time Synchronization Verification

A second set of time synchronization tests verifies the final setup, including the local clock update mechanisms as presented in section 4.3.2. These tests verify the latest implementation of the DOSIS framework's features and confirm an actuation uncertainty below 2 ms, as demanded in section 3.8.

5.2.1 Setup

The general test setup is similar to the setup from section 5.1.1. Again, three identical Nucleo-L496ZG evaluation boards act as main test platform. A transceiver-less CAN setup interconnects these nodes and additional GPIO pins on each node provide an interface for external triggers and measurements.

Load Generator

The used CAN load generator is identical to the CAN load generator presented in section 5.1.1.

Time Synchronization

Similar to the previous tests, the time synchronization verification compares the pairwise time difference of all nodes. For this purpose, the control computer logs the times printed via UART on each node whenever a trigger signal toggles. A STM32F407G discovery board generates this signal, toggling every 100 ms, which is directly handled in interrupt handlers on all nodes. In contrast to the previous setup, the actual DOSIS time model as presented in section 4.3.2 is used for all tests. Node 1 is used as reference node for all time synchronization tests.

The verification tests use a direct CAN based time transfer for all test cases and compares the direct-set, P-controlled, and PI-controlled clock update mechanisms. Similar to the previous tests, either the node containing the reference clock or one of the other nodes may additionally generate CAN traffic using the CAN load generator.

Within each test case, the measurement start 30 s after reset of all nodes and observes the current time on each node every 100 ms for a total of 10 min. Similar to the previous tests, the internal oscillator provides the reference for all local clocks to represent worst-case conditions. Similar to the previous test, the long term relative clock rate difference is $\sim$0.5 %. Figure C.1 depicts the relative difference of the local clocks without additional synchronization for a time span of 10 min. Additionally, a repetition of all tests using a 32.768 kHz external oscillator[2] available on the Nucleo evaluation

[2] An NX3215SA crystal oscillator with a frequency tolerance of $\pm 20 \cdot 10^{-6}$ [157].

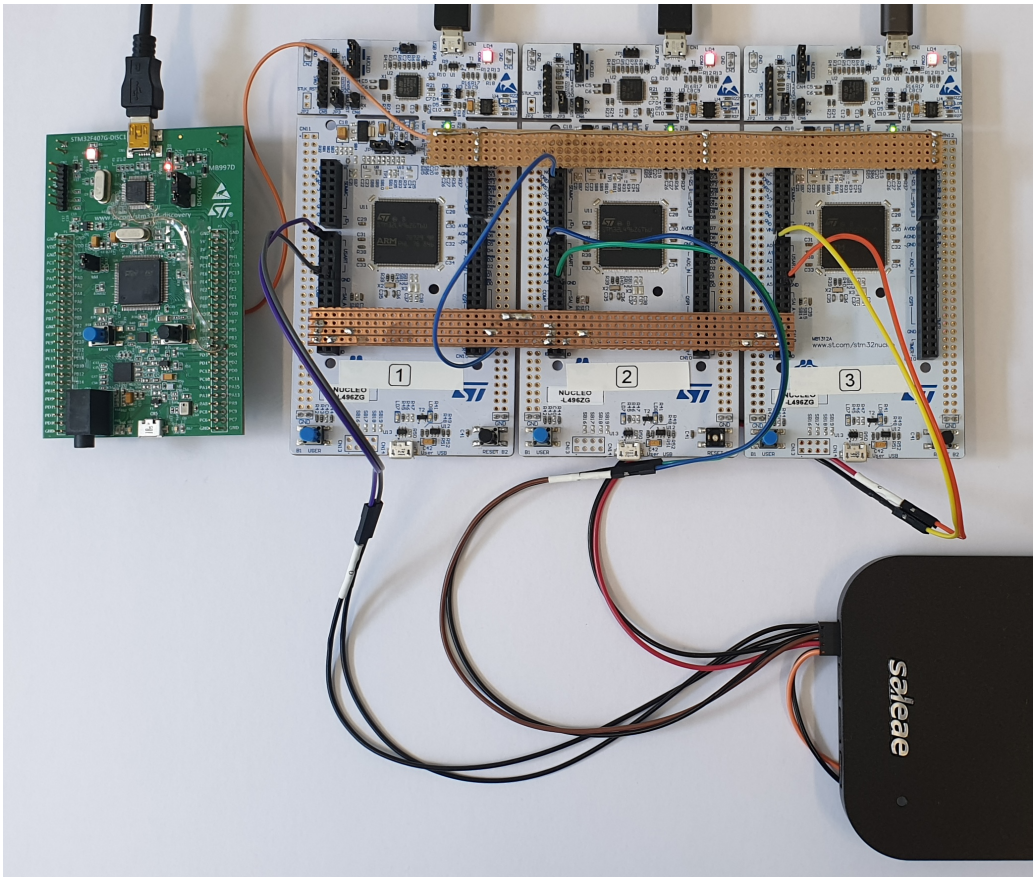


Figure 5.6: Verification setup of the DOSIS time synchronization. Shown are the three STM32L496ZG Nucleo nodes, the STM32F407G Discovery trigger generator connected to the input pin on all nodes and the Saleae logic analyzer connected to the output pin on each node individually. Note that the trigger generator is only used for time synchronization tests whereas the logic analyzer is only used for the control loop tests.

boards as source for the local clocks provides insight into the performance of the time synchronization in a setup closer to the expected setup in actual space applications.

Control Loop

The control loop test setup is again similar to the setup presented in section 5.1.1 but uses the latest DOSIS framework including the DOSIS time model as presented in section 4.3.2. Besides the used DOSIS version, the main difference between those setups is the controller implementation. To account for the computational effort of a real world distributed control system, the controller only forwards a command after 1 ms delay.

The control loop tests use the internal oscillator on all nodes, a CAN based time transfer, and a PI-controlled clock update. Similar to the previous tests, load generated at either sensor, controller, or actuator node optionally simulates a high CAN utilization.

Similar to the time synchronization test, each test case is repeated twice. The first repetition uses the internal oscillator with a large drift and short term fluctuations within this drift; the second repetition uses the external 32.768 kHz oscillator.

Figure 5.6 depicts the hardware setup used for the time synchronization and control loop verification tests.

5.2.2 Time Synchronization Test Results

The time synchronization verification tests compare the time synchronization performance of PI-controlled, P-controlled, and direct-set clock update with different CAN load conditions each. The

recording for each test case started 30 s after loading the firmware into all nodes and thus does not contain the initial synchronization including the potential hard set of the local time. A recording of each node's local time every 100 ms for a total of 600 s provides the raw data for each test. The pairwise time deviations calculated from these recordings are the indicator for the synchronization performance.

Figure 5.7 depicts the minimum, 25th percentile, median, 75th percentile, and maximum of these deviations for comparison of the time synchronization implementation of DOSIS using the internal oscillator. Node 1 — the node with the lowest skew factor of the three test nodes — is the reference node for these tests. Figures C.2 to C.10 in appendix C.4.1 depict the raw data of the pairwise time difference using the internal oscillator in greater detail.

Figure 5.8 depicts the second repetition of all tests using the external 32.768 kHz. Again, node 1 is the reference node for these tests. Likewise, figures C.12 to C.20 in appendix C.4.2 depict the raw data of the pairwise time difference using said external oscillator.

Figure 5.9 depicts the repetition of all tests using the external 32.768 kHz. For these tests, node 3 — the node with the highest skew factor of the three test nodes — is the reference node. Again figures C.12 to C.20 in appendix C.4.3 depict the raw data of the pairwise time difference for detailed reference.

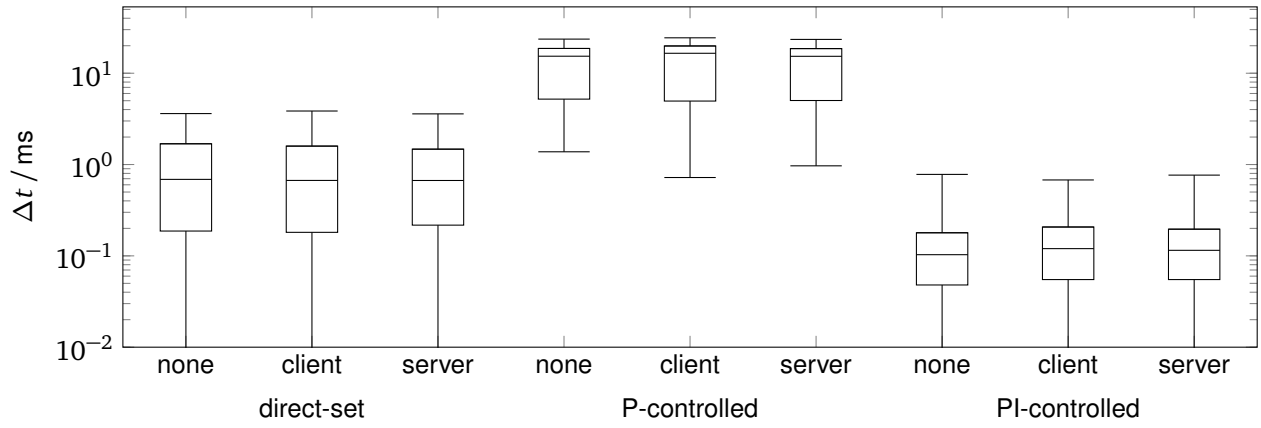


Figure 5.7: Pairwise time deviation Δt for different time synchronization setups using the internal oscillator as source for the local clocks. Shown is the distribution over 600 s sampled every 100 ms for PI-controlled, P-controlled, and direct-set clock update with different sources of CAN load. Node 1 is the time synchronization reference node. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

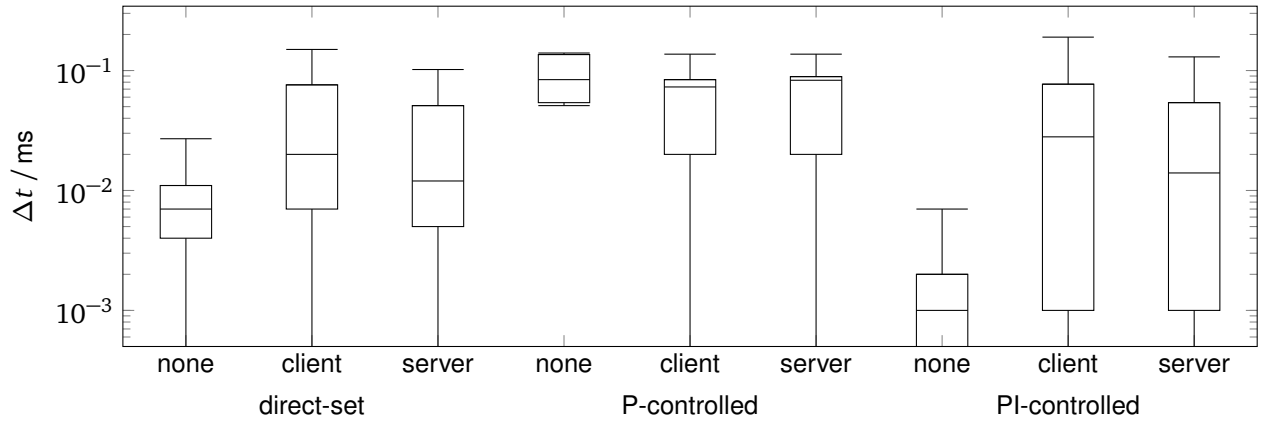


Figure 5.8: Pairwise time deviation Δt for different time synchronization setups using the external 32.768 kHz oscillator as source for the local clocks. Shown is the distribution over 600 s sampled every 100 ms for PI-controlled, P-controlled, and direct-set clock update with different sources of CAN load. Node 1 is the time synchronization reference node. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

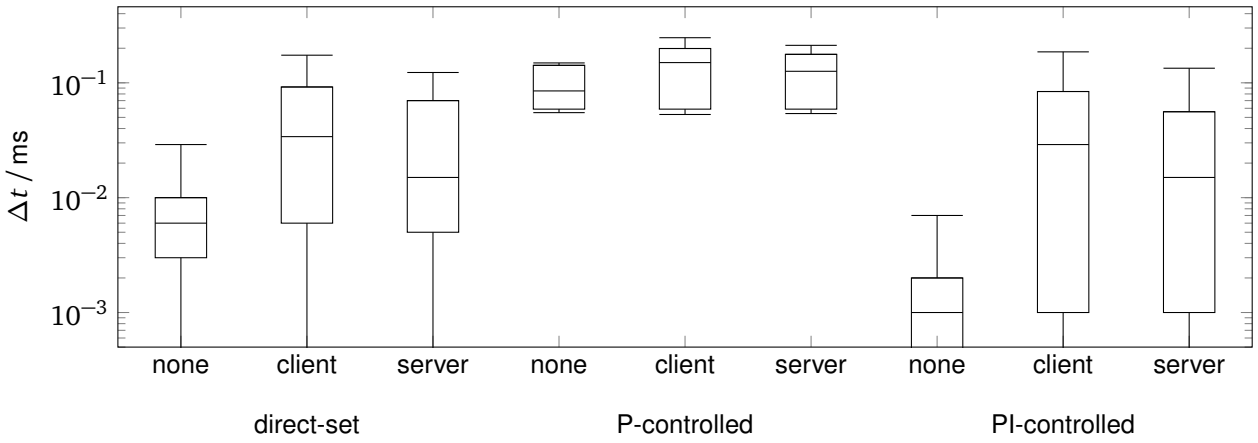


Figure 5.9: Pairwise time deviation Δt for different time synchronization setups using the external 32.768 kHz oscillator as source for the local clocks. Shown is the distribution over 600 s sampled every 100 ms for PI-controlled, P-controlled, and direct-set clock update with different sources of CAN load. Node 3 is the time synchronization reference node. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

5.2.3 Control Loop Test Results

Similar to the initial control loop tests, the sensor triggers a readout and thus one cycle every 500 ms. The first 30 s after loading the firmware may contain artifacts due to the initial time synchronization. The recording and analysis for each test case contains approximately 370 samples recorded after this initial delay. All figures depict the minimum, 25-perecentile, mean, 75-percentile, and maximum values for each test scenario.

The first set of control loop test uses the internal oscillator as source for each node's local clock. Figure 5.10 depicts the results of the direct actuation tests, i.e., using a controller that waits for 1 ms upon reception of a sensor readout and afterward commands the actuator for immediate action. Figure 5.11 depicts the results of the scheduled actuation at 100 ms after sensor readout respectively.

The second set of control loop tests uses an external 32.768 kHz oscillator as source for the local clocks. Figure 5.12 depicts the results of the direct actuation tests; Figure 5.13 depicts the results of the scheduled actuation at 100 ms after sensor readout.

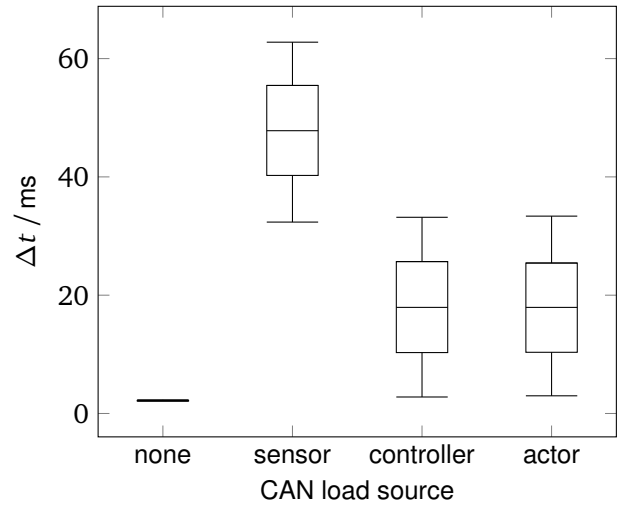


Figure 5.10: Results of the sensor-controller-actuator test with direct actuation using the internal oscillator as source for the local clocks. The controller simulates its computation time with a static 1 ms delay and commands the actuator to direct action. Δt is the time between sensor readout and actuator activation. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

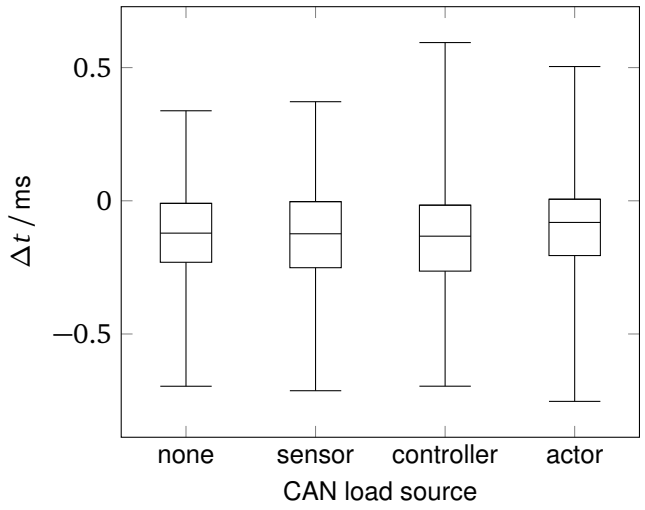


Figure 5.11: Results of the sensor-controller-actuator test with scheduled actuation using the internal oscillator as source for the local clocks. The controller simulates its computation time with a static 1 ms delay and command the actuator to a scheduled actuation 100 ms after sensor readout. Δt is the error between expected and actual actuation time. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

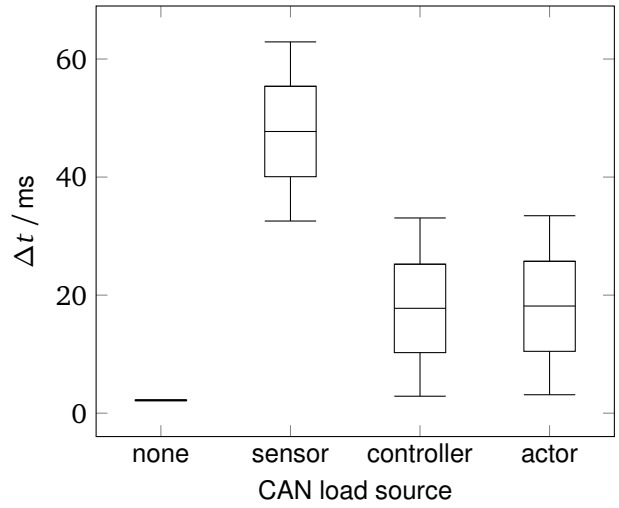


Figure 5.12: Results of the sensor-controller-actuator test with direct actuation using the external 32.768 kHz oscillator as source for the local clocks. The controller simulates its computation time with a static 1 ms delay and commands the actuator to direct action. Δt is the time between sensor readout and actuator activation. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

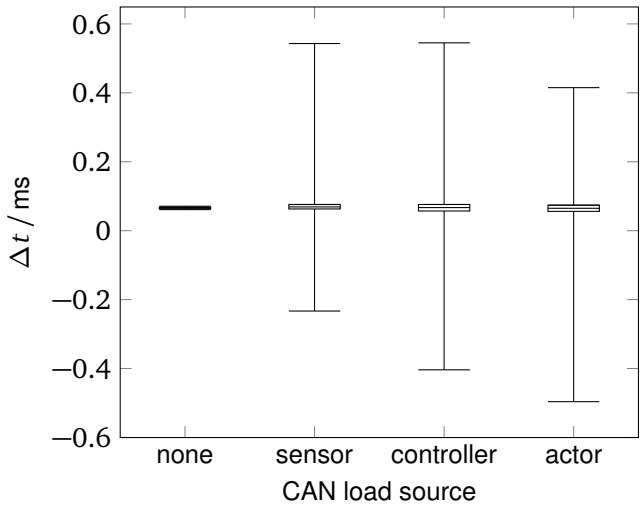


Figure 5.13: Results of the sensor-controller-actuator test with scheduled actuation using the external 32.768 kHz oscillator as source for the local clocks. The controller simulates its computation time with a static 1 ms delay and command the actuator to a scheduled actuation 100 ms after sensor readout. Δt is the error between expected and actual actuation time. The whiskers depict the minimum and maximum values; The boxes depict the first quartile, median, and third quartile.

5.2.4 Time Synchronization Test Discussion

The verification experiments for the implemented time synchronization mechanism using each node's internal oscillator confirm the previous result for a CAN based time transfer. Independent of the CAN load's origin, the direct-set update suffers from a remaining offset of up to 4 ms. This matches the expected result as the clocks of node 2 and node 3 will move apart about $4\,^{\text{ms}}/_{\text{s}}$ due to the skew difference of the local oscillators (see figure C.1) and the time synchronization interval, i.e., the time between consecutive updates, is set to 1 s.

Similarly, the P-controlled clock update confirms the previous result and again suffers from uncorrected offset. Although not clearly visible from figure 5.7, figures C.5 to C.7 clearly depict this static offset as the main reason for the large remaining error. This result is again independent of any additional CAN load.

The PI-controlled clock update yields the best result with a remaining error of less than 1 ms independent of the CAN load. Again, this confirms the previous experiments and thus confirms that the current implementation has a behavior similar to the version used for initial tests. Thus, using the DOSIS internal time model instead of a direct modification of the RODOS time representation does not significantly affect the overall time synchronization performance.

As expected, using each node's external 32.768 kHz oscillator as reference significantly improves the time synchronization performance. The direct-set clock update already yields good results as the relative skew is smaller (see figure C.11). Worst case relative skew of the used nodes is approximately $25 \cdot 10^{-3}\,^{\text{ms}}/_{\text{s}}$ for nodes 1 and 3. With a synchronization interval of 1 s we thus expect a worst-case error of approximately $25\,\mu s$. With any CAN load — which increases the time transfer delay by up to $T_{\text{CAN frame+}} = 160\,\mu s$ (see equation (3.4)) — we expect the worst-case error to increase up to $185\,\mu s$. The test runs with direct-set clock update depicted in figure 5.8 confirm these estimates.

Although the external oscillator reduces the remaining error for a P-controlled clock update, this update mechanism still suffers from a residual offset. While additional CAN load increases fluctuations of the remaining error, it also seems to improve the overall result. Such an improvement can be observed whenever the reference node's skew is smaller than the synchronized node's skew. In this case the added delay due to CAN collisions leads to an over-estimation of the error term. Due to the P-controller's nature, this will lead to a proportionally increased output and thus an increased control response. This increased response stirs the local clock of the synchronized node faster towards the reference clock and thus potentially reduces the remaining error. From figure C.11 we can see that the skew of the three test nodes are different and more specifically $s_3 > s_2 > s_1$. Node 1 — the node with the lowest skew of the used nodes — is the reference node for all test runs. Both of the synchronized nodes, nodes 2 and 3, thus overestimate the error term and show an improved result. As the additional delay due to CAN load affects both synchronized nodes similarly, the relative error between those nodes does not change significantly. Figures C.16 to C.17 better depict this difference in the individual nodes' response to CAN load. They also show that node 2 — which has a lower skew difference compared to the reference node — slightly overshoots. Similarly, the remaining error for nodes with a skew relatively smaller than the reference node's skew increases with CAN load. Figure 5.9 depicts the increased error in this scenario and Figures C.25 to C.27 show the effect on the individual nodes.

The PI-controlled clock update again yields the best result: the remaining error is $10\,\mu s$ without CAN load and less than 0.2 ms with CAN load on either side (see figure 5.8). While this is no significant improvement compared to the direct update, the PI-controller may even yield reasonable results with synchronization intervals of more than 1 s. Designers of a mission can use this result to find a trade-off between remaining error, clock update complexity, and synchronization interval matching their mission's requirements. Note that the controllers were not specially tuned for the environment, thus tuning a setup once the hardware and its parameters are known in detail may further improve the performance.

Overall, the repeated time synchronization tests confirm that the DOSIS framework and its implementation including the internal time representation without direct modifications of RODOS meet the requirement for a remaining relative time uncertainty of less than 2 ms from section 3.8. Thus, the DOSIS time synchronization is suitable for the intended purpose.

5.2.5 Control Loop Test Discussion

The control loop tests based on the sensor-controller-actuator setup with internal oscillator as clock reference yields results similar to the initial tests. Again, the direct actuation suffers from the expected delays while the timed actuation yields good results. The remaining error for those timed actuation test runs remains below 1 ms independent of the additional CAN load's source. For all control loop tests, the time synchronization's reference node is node 1. This node also has a skew error compared to the logging computer connected to the GPIO pin of all boards. Therefore, the results shown in figure 5.11 have a small remaining offset.

Using an external 32.768 kHz reference oscillator on each node has no effect on the observable delay of direct actuation tests as the CAN delay fluctuations are the major factor.

With timed actuation the average time does not exactly match the expected 100 ms, instead the average delay between sensor readout and actuator activity for scheduled actuation is about 100.07 ms. Again, the time synchronization only synchronizes local clocks, whereas it does not synchronize towards the recording computer. A static offset thus remains and reflects the skew difference between the recording computer's clock and the reference node. The remaining relative error is less than 0.01 ms in the absence of any additional CAN load, which matches the expected uncertainty due to the time synchronization itself. Additional CAN load generates a few outliers with a larger relative error of up to ± 0.5 ms. Multiple possible reasons for these outliers exist: interrupts due to additional CAN messages received on the actuator node, forward propagation of the time using temporary outliers of the skew due to time synchronization noise, or a remaining error in the time synchronization. As the outliers are still within reasonable margins, the author did not conduct a more detailed analysis of their exact reason.

Overall, these tests confirm that the major influence on the precision of any sensor readout or actuator activity is the time synchronization. As the time synchronization has a remaining relative error of about 0.1 ms using the external clock source and the sensor to actuator timing uncertainty is well below 1 ms, the DOSIS framework is suitable for timing critical applications such as an ADCS control loop.

Chapter 6

Radiation Test

6.1 Radiation Test Setup

The single event performance of the VA41620 and STM32L496ZG microcontrollers were evaluated at the πM1 beam line of Paul Scherrer Institute (PSI). The πM1 beam provides protons and pions at a momentum of 100 MeV c^{-1} to 500 MeV c^{-1} [165, 166].

A common hardware test setup for both microcontrollers provides a comparable setup. All test setups use evaluation boards provided by the respective vendors as carrier board for the respective unit under test (UUT). For the presented radiation tests, we use a PEB1-VA41620 by Vorago Technologies to evaluate the VA41620 microcontroller and a NUCLEO-L496ZG-P by STMicroelectronics for the STM32L496ZG microcontroller. These carrier PCBs provide the electrical and mechanic interface for all radiation tests.

6.1.1 Mechanical Setup

The hardware setup consists of an upstream trigger detector, a mount for the UUT, and a downstream trigger detector. The upstream detector counts the particles in a small area right before they reach the UUT. The active part of the upstream detector is a 10 mm × 10 mm × 5 mm Saint-Gobain BC-408 scintillator [187]. A 1 mm thick Al casing protects the scintillator and the photo-sensitive detector readout electronics from light. The UUT mount holding the carrier PCB aligns the UUT 47 mm behind the upstream detector. A second detector located 111 mm behind the front side of the carrier PCB observes particles after they pass the UUT. This downstream detectors active part is a 80 mm × 80 mm × 4 mm Saint-Gobain BC-408 scintillator[187]. Similar to the upstream detector, the downstream detector is shielded from light using a 1 mm Al casing. A XYZ table is the base for the whole setup. This XYZ table aligns the whole setup into the center of the particle beam. It finally positions the upstream detector at a distance of 122 cm from the beam window. Figure 6.1 provides an overview of the test setup and figure 6.2 visualizes the dimensions of this setup.

6.1.2 Electrical Setup

The electrical setup consists of two parts: the UUT setup and the detector readout electronics.

Unit Under Test

A carrier PCB provides the required electrical interfaces to the UUT. The debug USB port of the carrier PCB is connected to a sequencing and logging server. An additional UART connection between UUT and server enables logging of the UUT's console output. A USB to UART adapter provides the required UART interfaces for the server. Both USB devices are connected to the server via USB isolators. A Rhode & Schwarz HMP4040 power supply provides the necessary voltage for these isolators and enables logging of the electrical current drawn by the UUT and the UART converters.

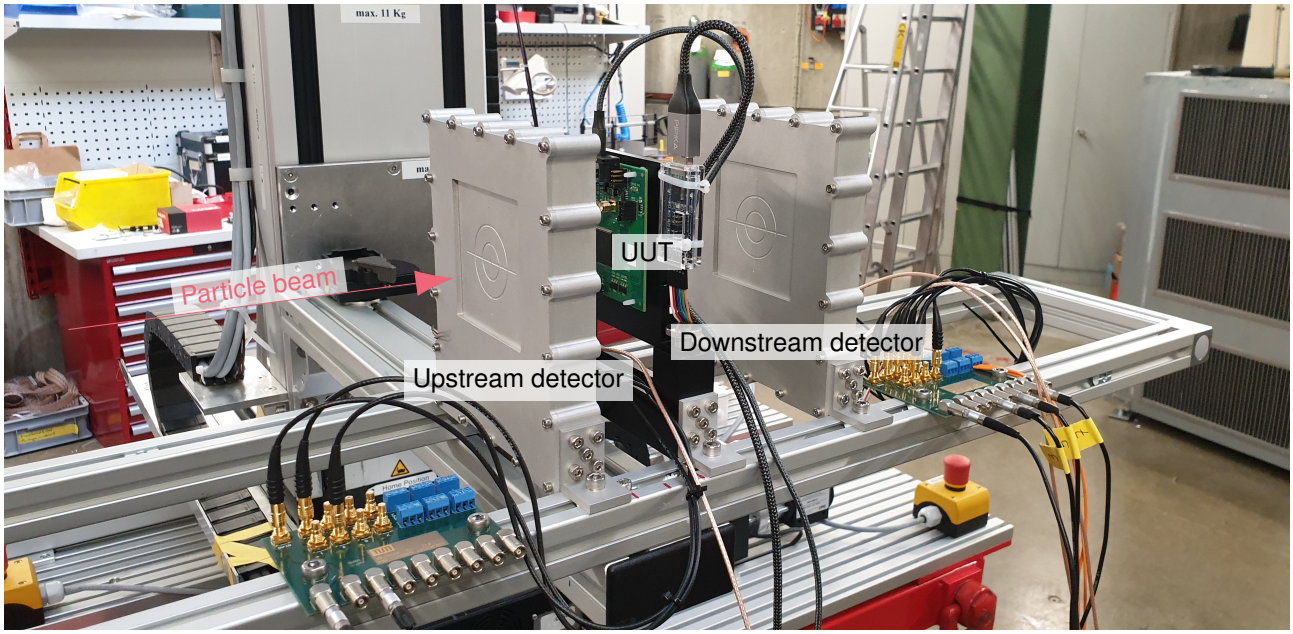


Figure 6.1: Overview of the radiation test setup at the πM1 beam line.

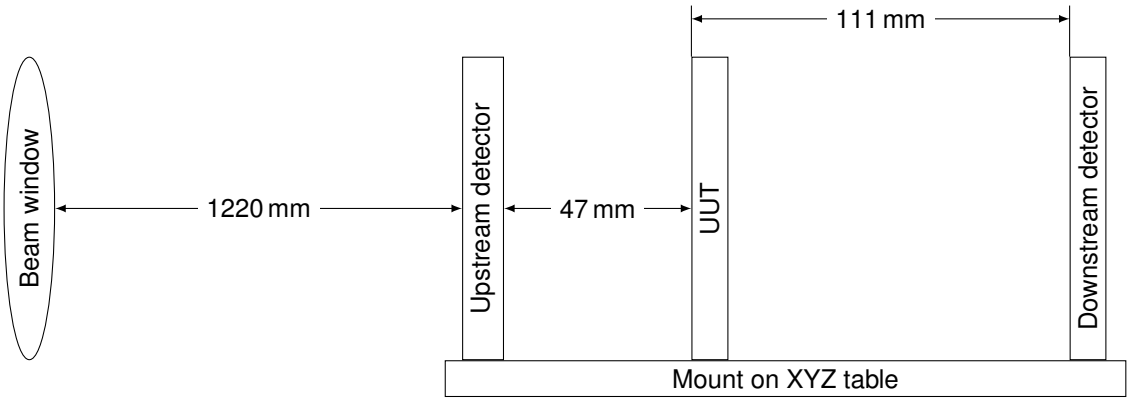


Figure 6.2: Mechanical dimensions of the setup at the πM1 beam line.

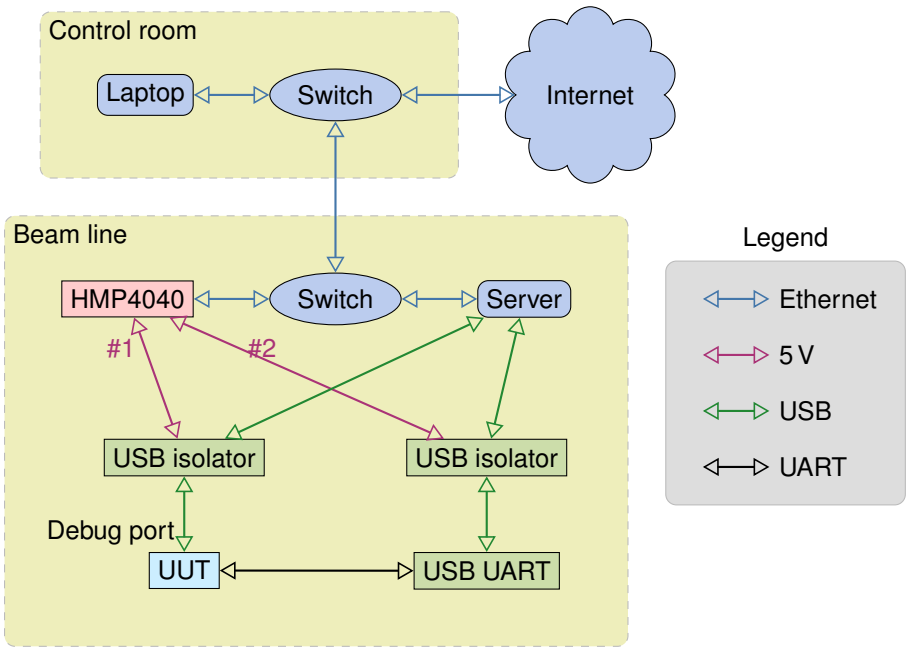


Figure 6.3: Electrical setup used for the UUT at the πM1 beam line.

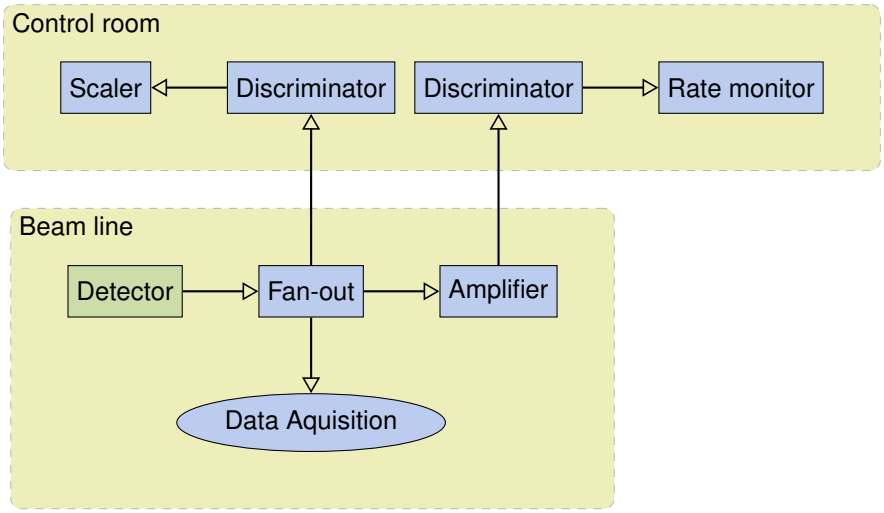


Figure 6.4: Electrical setup for analog readout of the detectors at the πM1 beam line.

A laptop located in the control room provides remote monitoring and control interface for the whole setup. This is necessary as no physical access to the server or any other equipment located near the beam line is possible while the beam line is active.

Figure 6.3 depicts the used electrical setup.

Detector Readout

Each detector consists of a scintillator surrounded by a number of silicon photomultipliers. A data acquisition hardware stores the signal of all detectors as individual channels based on a trigger signal. Additionally, an analog readout for one channel of each detector provides live information. Figure 6.4 depicts the analog setup. The discriminators suppress noise on the readout and create a signal suitable for the scaler and monitor. The scaler counts the signal pulses and thus the total amount of particles passing the upstream detector's surface of 1 cm^2 and thus provides a direct readout of the total fluence of particles ϕ of the current test run. The monitor counts the signal pulses for 1 s and resets itself every 2 s and thus provides a readout of the current flux Φ.

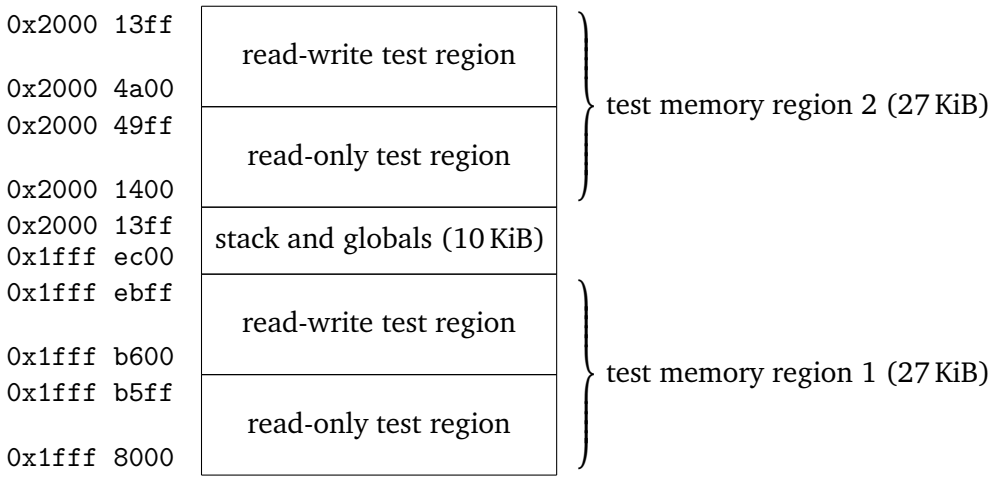


Figure 6.5: Memory layout of the VA41620 MCU used for radiation testing.

6.1.3 Test Software

A simple test software monitors the embedded RAM of the UUT for every test run. The software uses RODOS and implements a custom `MAIN` function. RODOS calls this function after system startup but prior to scheduler activation. Thus, the test software is a single-threaded application with minimal requirements regarding the surrounding system.

The test software first initializes the required hardware features and initializes the memory region used for testing. After initialization, an infinite loop performs repeated self-checks of the UUT's RAM. It performs a check of the read-write memory region every 50 ms, a check of the read-only memory region every 1 s; and generates a heartbeat every 5 s. The test software outputs any observations as well as the heartbeat via the UART interface which is finally logged on the previously mentioned server.

System Initialization

The test software initializes the overall system and the memory locations used for testing. The initialization fills a memory region dedicated for read-write testing and a memory region for read-only testing with a predefined pattern. This pattern is similar to a checkerboard pattern, but contains a dependency to the offset into the memory region. This dependency on the memory location avoids false negative test results if memory access fails and returns content of a different memory location instead. The content of each 32 bit memory location is `0xCCCC CCCC - offset`, where `offset` is the array index of the specific memory cell. 32 bit integers are used for all memory accesses.

The used memory layout of the available RAM can be seen in figure 6.5 and figure 6.6. For both microcontrollers, a region in the center of the available memory hold the stack and global variables required for the test software itself.

Additional Initialization of VA41620 The memory scrubbing and register refresh of the VA41620 is set up to regularly scan and restore the microcontroller's memory. Scrubbing of RAM and code RAM (called read-only memory (ROM) within the VA41620 documentation) is set to scan one memory word of 4 B every 32 clock cycles. With a system clock rate of 100 MHz a scrub engine will scan at a rate of

$$4\,\mathrm{B} \cdot \frac{100\,\mathrm{MHz}}{32} = 12.5\,\mathrm{B}\,\mu s^{-1} \approx 12.2\,\mathrm{KiB}\,\mathrm{ms}^{-1}. \tag{6.1}$$

The 64 KiB RAM are split into two regions with individual scrub engines. Therefore, the whole RAM is scrubbed every

$$\frac{64\,\mathrm{KiB}}{2 \cdot 12.5\,\mathrm{B}\,\mu s^{-1}} \approx 2.6\,\mathrm{ms}; \tag{6.2}$$

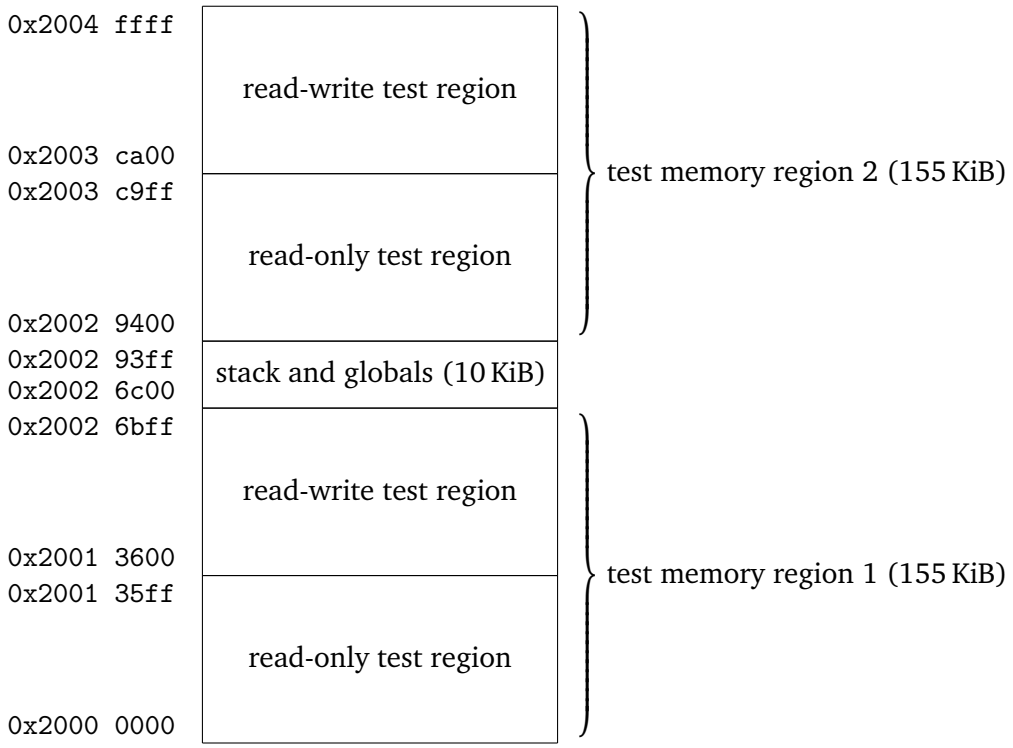


Figure 6.6: Memory layout of the STM32L496ZG MCU used for radiation testing.

the 256 KiB ROM every

$$\frac{256\,\text{KiB}}{12.5\,\text{B}\,\mu s^{-1}} \approx 21\,\text{ms}. \tag{6.3}$$

The TMR register refresh is set to refresh all registers every 256 clock cycles. With a system clock rate of 100 MHz, it refreshes each registers every

$$\frac{256}{100\,\text{MHz}} = 2.56\,\mu s. \tag{6.4}$$

Memory Test

Besides the different memory size of the VA41620 and the STM32L496ZG the memory test routine is identical. The read-only memory test compares each memory location with the predefined pattern used for initialization. The read-write memory test verifies the memory in three steps. First, it compares the memory with the predefined pattern similar to the read-only memory test. Afterward, it overwrites the memory with the bitwise negated value (i.e., it flips every bit) and reads the memory location again to verify a successful write operation. Finally, it repeats the second step with the original pattern. This leaves the memory with the exact same content it initially had for the next iteration.

If any memory test reads an unexpected value, it generates an output with all relevant information. This output contains the affected memory location, the expected value and the actual value and is printed on the previously mentioned UART port.

Heartbeat

The heartbeat of both microcontrollers generates an additional UART output to verify that the software is alive. The VA41620 test software additionally monitors the hardware counters for bit errors in RAM and ROM corrected by the hardware's built-in memory scrubbing. If it detects a change of any hardware counter, the software generates an output containing the affected memory region, the old value, and the new value and prints this information as part of the heartbeat via the UART port.

6.2 Data Analysis Method

The data analysis is a multistep process. First, a preprocessing step converts the raw recordings into a better understandable format. Afterwards, a statistical analysis provides insight into the observed bit-flips. Additionally, the deposited energy and an estimate for the flux and fluence errors are derived from the acquired recordings and instrument readouts.

6.2.1 Raw Data Preprocessing

The test software generates an output contains:

- a heartbeat in a regular interval to detect test software malfunction,
- the memory address, expected content, and actual content whenever the test software detects a bit-flip, and
- hardware counters with every heartbeat if their values have changed (VA41620 only).

During logging of this output, the logging-computer prepends every line with the current timestamp.

A first processing step summarizes any observed events into 5 s intervals. For each 5 s interval, it stores the amount of heartbeats, reboots, hardware corrected bit-flips on the VA41620, uncorrected bit-flips of read-only memory tests, and uncorrected bit-flips of read-write memory tests. Additionally, information on the test run itself is gathered and combined with the recording. Each processed data contains information on the particle type (proton, pion+, pion-), the fluence (cm^{-2}), flux ($\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$), and momentum of the particles at the beam window ($\mathrm{MeV}\,c^{-1}$) noted during the specific test run.

6.2.2 Bit-Flip Analysis

A second analysis step estimates the expected bit-flip probability in a 5 s interval and the uncertainty of said probability using bootstrap statistics [60]. The bootstrap method does not require knowledge about the probability distribution of the random variable, but instead enables a statistical analysis based on observed data. The used bootstrap analysis using the Matlab `bootci` method resamples the observed data 10'000 times and calculates the mean value on each resampled set. Afterward, it estimates the 68.2 %, i.e., 1-σ, confidence interval and the overall mean value. Multiple test runs with identical MCU, particle type, and momentum — excluding test runs with instable beam power, a flux not in $120 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ to $130 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$, or unexpected system freezes — are combined into a single data set for this analysis. Test runs with instable beam power are excluded due to the increased flux and fluence uncertainty. Exclusion of test runs with a flux outside the given range reduces the uncertainty regarding the pile-up estimation (see section 6.2.4). Finally, test runs with an unexpected freeze generate incomplete observations only and are thus not suitable for direct comparison. This result represents the likelihood and 1-σ confidence interval for the likelihood of a bit-flip in a 5 s interval for a specific MCU, particle type, and particle momentum.

As a final analysis step, the bit-flip analysis normalizes the result to monitored memory size and fluence of the specific test run into a bit-flip cross-section $\sigma_{\text{bit-flip}}$:

$$\sigma_{\text{bit-flip}} = \frac{N_{\text{bit-flip}}}{S_{\text{memory}} \cdot \text{bit}^{-1} \cdot \phi} \tag{6.5}$$

with

$\sigma_{\text{bit-flip}}$	:	bit error cross section,
$N_{\text{bit-flip}}$	:	number of observed bit-flips,
S_{memory}	:	size of monitored memory, and
ϕ	:	particle fluence.

6.2.3 Deposited Energy

The deposited energy and LET in the UUT's active volume enables normalized comparison of different SEE tests. A simulation based on the observed deposited energy in the upstream and downstream detector estimates the LET for protons with different initial momenta. For this purpose, a simulation containing the physical test setup — i.e., the upstream detector, UUT, downstream detector, and their respective relative distances — traces particles along their path. The initial particle momentum is tuned to match the simulation result and the observation of the deposited energy in the upstream and downstream detector[1]. In contrast to the physical setup, the simulation directly monitors the deposited energy of the traced particles within the UUT's active volume, i.e., the MCU's silicon die. Thus, after tuning of the initial particle momentum, the simulation generates the deposited energy in the UUT's active volume. Finally, the LET as a linearized estimate and the 1-σ interval for the deposited energy per unit path length calculates from the simulated deposited energy in the UUT and the thickness of the UUT's active volume.

6.2.4 Particle Flux and Fluence

While the scaler and rate monitor provide a direct readout of flux Φ and fluence ϕ, the beam power was not fully stable during test runs, thus readouts of flux and fluence may not match. To estimate the uncertainty, the flux is also estimated based on the fluence and the active beam time, i.e., the time between initially activating the beam for a test run and the end of the test run. The further analysis uses the mean value. The deviations of the two values provide an estimate for the remaining uncertainty. In extreme cases the mean value deviates by about 10 % from the two values.

Additionally, the pileup — i.e., the proportion of multiple hits within a single recording frame — is estimated from upstream detector recordings. The observed pileup ratio is (5 ± 1) % for all test runs with a flux readout of between $120 \cdot 10^3\,\mathrm{cm^{-2}\,s^{-1}}$ to $130 \cdot 10^3\,\mathrm{cm^{-2}\,s^{-1}}$. Detailed effect of this pileup on analog readouts of flux and fluence is unknown as the precise behavior of the analog setup, e.g., of amplifier and discriminators, is unknown. As the analog readout is potentially faster than the digital recording (where a single recording has a duration of 800 ns), the estimated pileup from detector recordings provides a worst-case estimate for the error introduced to the analog readouts.

We will thus assume a worst-case error of ±10 % due to beam fluctuations and the unknown behavior of the analog instruments, and +5 % due to potential pileup. All analysis will thus include a total error of −10 % to +15 % on the flux and fluence values.

6.3 VA41620 Radiation Test Results

A total of 21 test runs for two VA41620 MCUs provide the basic data for the further analysis. Table 6.1 depicts the basic information for these test runs. Identifiers are assigned in chronological order. No software-observable bit-error occurred during these test runs, but a number of hardware-corrected single-bit errors were reported by the corresponding hardware counters. The further analysis thus presents the bit-flip cross-section of hardware corrected bit-flips for different scenarios. Figure E.1 in appendix E visualizes the raw analysis without exclusions for each individual test run.

As presented in section 6.2.2, the further analysis combines all test runs with identical particle type, used MCU, and particle momentum into a single result. Excluded from this analysis are:

- test runs 1, 2, and 3 due to a deviating flux;
- test runs 3, 18, and 21 due to beam instabilities;
- and test run 8 due to a freeze of the MCU.

[1] These simulations and the tuning of parameters where conducted by T. Pöschl and M. Losekamm at the TUM department of physics.

A separate analysis and discussion of the reasons for the freeze of the MCU in test run 8 is necessary. Figure 6.7 visualizes the combined data of the hardware-corrected bit-flip cross-section in the remaining test runs for various particle momenta.

The LET analysis focuses on proton tests as these are the main source of radiation in a LEO environment (see section 2.3). Figure 6.8 depicts the combined result of the test runs for different proton LET.

Table 6.1: Overview of VA41620 radiation test runs sorted by used MCU, particle type, and momentum. Flux and fluence from scaler and rate monitor readout as presented in section 6.1. Bit errors are hardware-corrected single-bit errors as reported by hardware counters.

Id	Timestamp	Used MCU	Particle Type	Momentum p / MeV c^{-1}	Fluence ϕ / cm^{-2}	Flux Φ / cm^{-2} s^{-1}	Bit Errors $N_{\text{bit-flip}}$	Comment
11	2021-06-30 16:37	1	proton	240	$84 \cdot 10^6$	$126 \cdot 10^3$	0	
12	2021-06-30 17:08	1	proton	295	$101 \cdot 10^6$	$123 \cdot 10^3$	22	
1	2021-06-30 08:17	1	proton	310	$21 \cdot 10^6$	$23 \cdot 10^3$	11	low flux for setup purposes
2	2021-06-30 09:30	1	proton	310	$30 \cdot 10^6$	$32 \cdot 10^3$	27	low flux for setup purposes
3	2021-06-30 10:55	1	proton	310	$270 \cdot 10^6$	$230 \cdot 10^3$	56	beam power unstable
4	2021-06-30 11:55	1	proton	310	$545 \cdot 10^6$	$126 \cdot 10^3$	191	
21	2021-07-01 10:58	1	proton	310	$106 \cdot 10^6$		27	beam power unstable
8	2021-06-30 15:20	1	proton	325	$<77 \cdot 10^6$	$126 \cdot 10^3$	6	freeze after ∼2 min
9	2021-06-30 15:34	1	proton	325	$101 \cdot 10^6$	$129 \cdot 10^3$	24	
7	2021-06-30 14:57	1	proton	330	$104 \cdot 10^6$	$129 \cdot 10^3$	35	
5	2021-06-30 13:31	1	proton	345	$101 \cdot 10^6$	$125 \cdot 10^3$	44	
10	2021-06-30 16:01	1	proton	360	$102 \cdot 10^6$	$128 \cdot 10^3$	33	
6	2021-06-30 14:03	1	proton	375	$271 \cdot 10^6$	$128 \cdot 10^3$	74	
14	2021-06-30 18:24	1	pion−	375	$100 \cdot 10^6$	$32 \cdot 10^3$	78	
13	2021-06-30 17:34	1	pion+	375	$104 \cdot 10^6$	$122 \cdot 10^3$	59	unexpected reboot
19	2021-07-01 10:02	2	proton	310	$102 \cdot 10^6$	$123 \cdot 10^3$	21	
20	2021-07-01 10:27	2	proton	330	$101 \cdot 10^6$	$121 \cdot 10^3$	29	
18	2021-07-01 09:00	2	proton	345	$136 \cdot 10^6$	$122 \cdot 10^3$	46	flux of $\Phi = 15 \cdot 10^3$ cm^{-2} s^{-1} for first $40 \cdot 10^6$ cm^{-2}
17	2021-07-01 08:33	2	proton	375	$122 \cdot 10^6$	$123 \cdot 10^3$	39	
15	2021-06-30 23:39	2	pion−	375	$522 \cdot 10^6$	$32 \cdot 10^3$	302	2 unexpected reboots
16	2021-07-01 08:03	2	pion+	375	$102 \cdot 10^6$	$123 \cdot 10^3$	54	

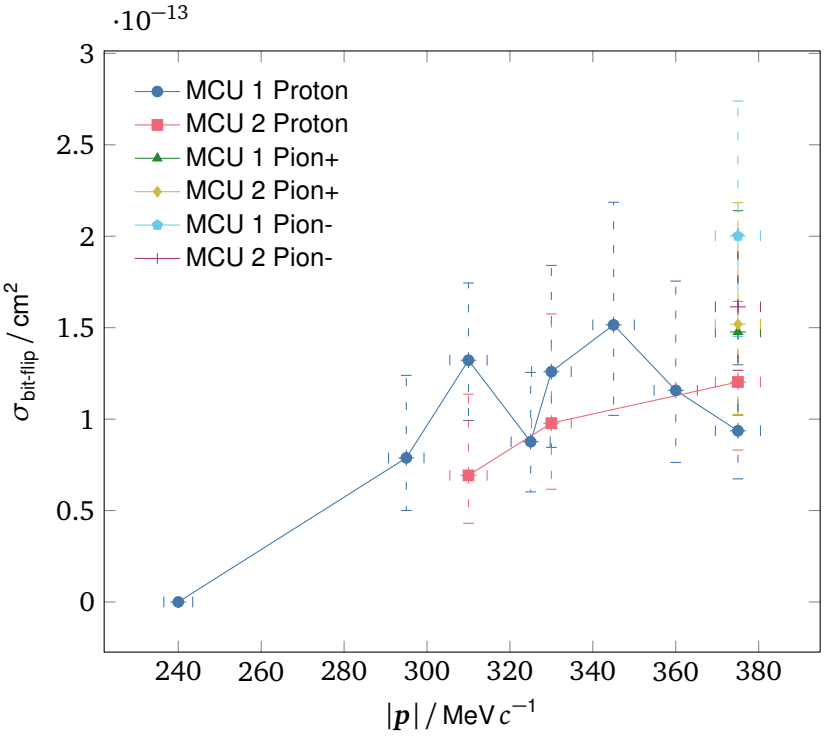


Figure 6.7: Hardware corrected bit-flip cross-section $\sigma_{\text{bit-flip}}$ of the VA41620 RAM and ROM memory regions for different particles and momenta. Each data-point represents the combined and normalized result for all test runs with identical particle type, momentum, and test hardware including 1-σ error bars for $\sigma_{\text{bit-flip}}$. Impulse error bars show the πM1 beam line's full width at half maximum according to [166].

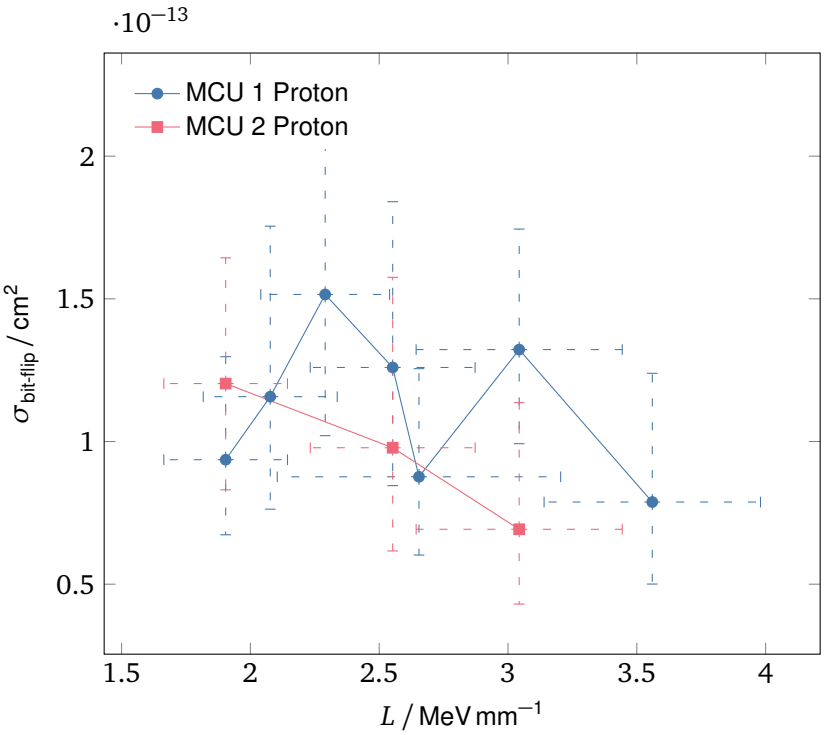


Figure 6.8: Hardware corrected bit-flip cross-section $\sigma_{\text{bit-flip}}$ of the VA41620 RAM and ROM memory regions due to proton irradiation for different LET. Each data-point represents the combined and normalized result for all test runs with identical LET and test hardware including 1-σ error bars.

6.4 STM32L4 Radiation Test Results

The STM32L4 radiation test data consists of a total of nine individual test runs. Table 6.2 depicts the basic information on these test runs. In contrast to the VA41620 MCU, the STM32L496ZG used for these tests does not feature any error correction for its RAM. Thus, the reported bit errors are bit errors detected by the test software itself. Again, some test runs are not suitable for direct comparison: Test run 1 was excluded from the future analysis due to the low flux of only $23 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. Additionally, test runs 4 and 6 are excluded due to off-nominal behavior of the test itself. While the MCU entirely stopped its output in test run 4, the node did print excessive output in test run 6. The excessive output affects the data as it prevents the test loop from its regular operation as it has to wait for the relatively slow UART.

The data of only a few STM32L4 radiation test runs is available and does not include any duplicate runs for the same momentum and particle type. Thus, the analysis does not combine multiple test runs into a single data point but instead analyses each test run individually. Figure 6.9 visualizes the observed bit-flip cross-section for different particle types and momenta. Similar to the VA41620 test results, the proton tests were further processed to depict the bit-flip cross-section for different LET. Figure 6.10 visualizes this additionally processed data.

Although no significant difference between read-only and read-write test patterns could be observed, figures E.2 and E.3 in appendix E depict this analysis for reference.

Table 6.2: Overview of STM32L496ZG radiation test runs sorted by used MCU, particle type, and momentum. Flux and fluence from scaler and rate monitor readout as presented in section 6.1. The bit errors column shows the sum of software detected bit errors in read-only and read-write memory regions.

Id	**Timestamp**	**Used MCU**	**Particle Type**	**Momentum** p / MeV c^{-1}	**Fluence** ϕ / cm^{-2}	**Flux** Φ / cm^{-2} s^{-1}	**Bit Errors** $N_{\text{bit-flip}}$	**Comment**
1	2021-06-29 22:05	1	proton	310	$15 \cdot 10^6$	$23 \cdot 10^3$	6	low flux for setup purposes
2	2021-06-30 20:27	2	proton	310	$103 \cdot 10^6$	$125 \cdot 10^3$	18	flux of $\Phi = 4 \cdot 10^3$ cm^{-2} s^{-1} for first 6 min
3	2021-06-30 20:58	2	proton	330	$103 \cdot 10^6$	$123 \cdot 10^3$	19	
4	2021-06-30 21:21	2	proton	345	$30 \cdot 10^6$	$128 \cdot 10^3$	2	freeze of MCU
5	2021-06-30 21:29	2	proton	345	$70 \cdot 10^6$	$128 \cdot 10^3$	18	
6	2021-06-30 21:48	2	proton	375	$37 \cdot 10^6$	$123 \cdot 10^3$	55 565	run-away output / failed software
7	2021-06-30 21:54	2	proton	375	$70 \cdot 10^6$	$123 \cdot 10^3$	8	
9	2021-06-30 22:55	2	pion−	375	$30 \cdot 10^6$	$31 \cdot 10^3$	1	
8	2021-06-30 22:15	2	pion+	375	$101 \cdot 10^6$	$124 \cdot 10^3$	21	

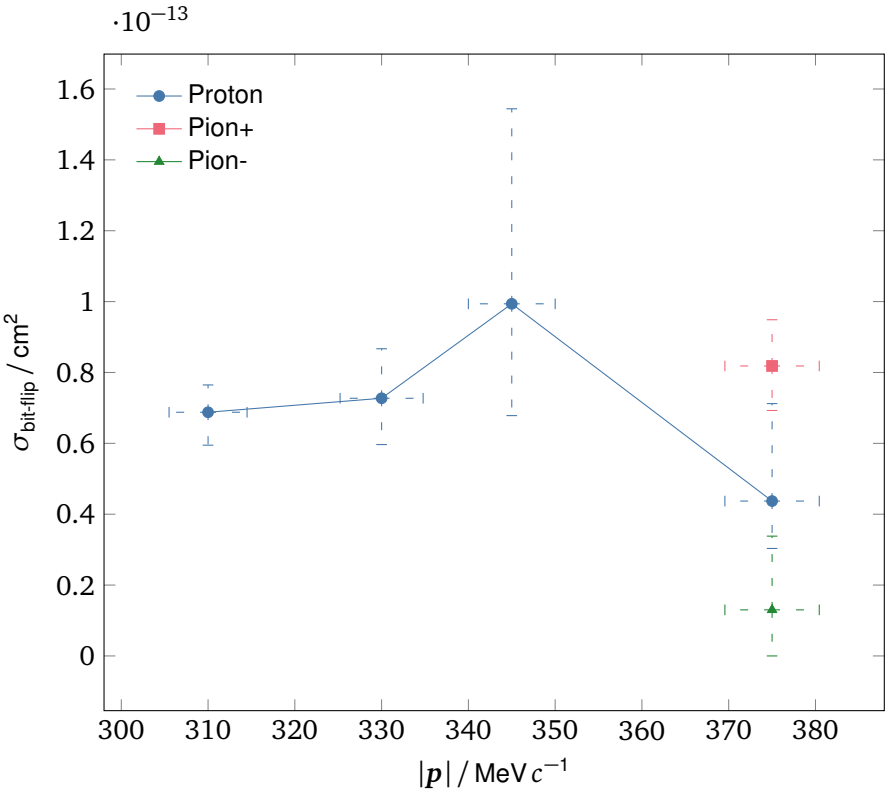


Figure 6.9: Observed bit-flip cross-section $\sigma_{\text{bit-flip}}$ in the monitored STM32L4 memory region of MCU 2 for different particles and momenta including 1-σ error bars for $\sigma_{\text{bit-flip}}$. Impulse error bars show the πM1 beam line's full width at half maximum according to [166].

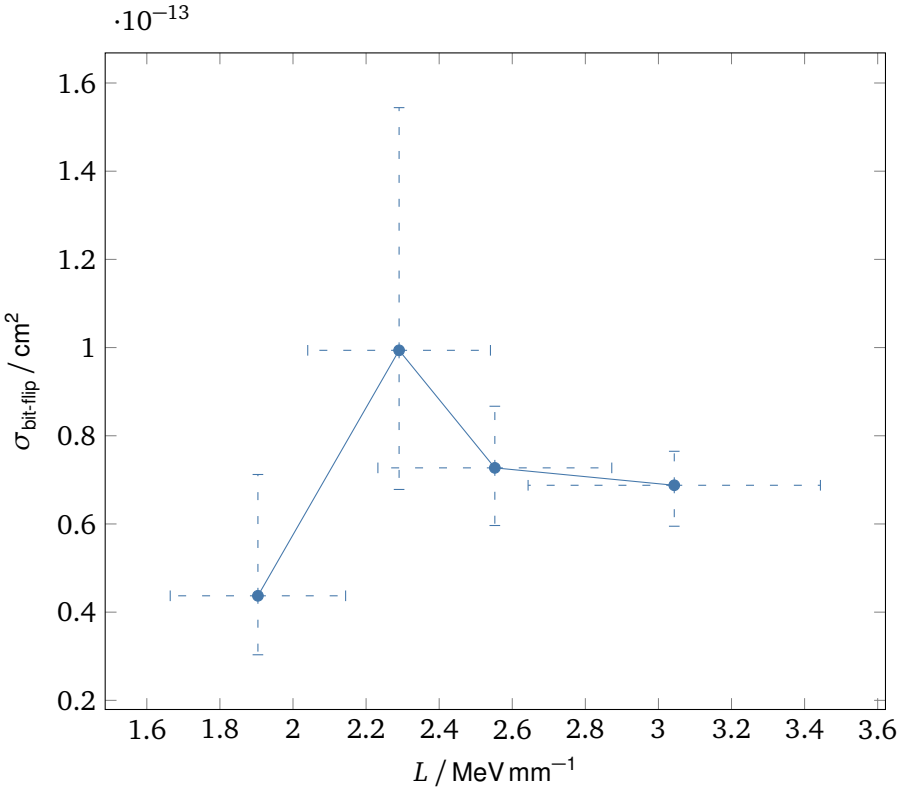


Figure 6.10: Observed bit-flip cross-section in the monitored STM32L4 memory region of MCU 2 due to proton irradiation for different LET including 1-σ error bars.

6.5 VA41620 Radiation Test Discussion

Previous proton radiation tests of the VA10820 MCU by Wilcox and Seidleck [230] verified the radiation tolerance and the built-in hardware error-correction for the MCU's internal memory. For these tests, the authors of [230] use protons with 200 MeV kinetic energy at a flux of $4.8 \cdot 10^7\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ to $7.8 \cdot 10^7\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. Their measurements show an average RAM and ROM bit-error cross-section per bit of memory of $3.2 \cdot 10^{-14}\,\mathrm{cm}^2$ to $1.2 \cdot 10^{-13}\,\mathrm{cm}^2$. The built-in hardware error-correction successfully corrected all the observed bit errors during these tests [230].

Vorago uses the same HARDSIL technology for the VA41620 previously used for the VA10820. Due to the same technology and their similar specifications we expect them to also behave similarly regarding proton radiation. The results presented in section 6.3 confirm this expectation with a bit-flip cross-section of $3.8 \cdot 10^{-14}\,\mathrm{cm}^2$ to $2.3 \cdot 10^{-13}\,\mathrm{cm}^2$. All bit-flips were corrected in hardware and thus no single bit-error could be seen by memory accesses of the software.

Test run 11 with an initial proton momentum of $240\,\mathrm{MeV}\,c^{-1}$ acts as a reference to verify that the observed bit-flips originate from radiation effects. With this relatively low momentum, the protons barely reach the upstream detector — enabling a flux and fluence estimate — but can not reach the actual UUT. As no bit-flips occurred during test run 11, the effects observable during the remaining test runs are related to the radiation and do not originate from other parts of the test setup itself. While a worst-case estimate for the bit-error cross-section of $2.3 \cdot 10^{-13}\,\mathrm{cm}^2$ ($2.7 \cdot 10^{-13}\,\mathrm{cm}^2$ for pions) can be given from the test result, no clear trend regarding different particle momenta or LET (for protons) is visible.

A few anomalies, excluded from the general analysis are present in the test results. During test run 8 — a proton test run with an initial proton momentum of $325\,\mathrm{MeV}\,c^{-1}$ corresponding to a LET of $2.7\,\mathrm{MeV}\,\mathrm{mm}^{-1}$ — the UUT froze after approximately two minutes and did not generate any additional UART output. At the same time, the power consumption via the UUT's debug port did grow significantly from 128 mA to 154 mA. The reason for this behavior is either an interruption of the MCU's functionality, a software error triggered by an otherwise unobserved event or series of events, or an interruption of the functionality of a peripheral device on the evaluation board. An interruption of the voltage regulators' functionality due to a SEE caused by a proton would be an example for a periphery failure explaining the observed behavior. While the exact reason for this failure is unknown, a power cycle of the entire UUT could resolve the failure. A detailed analysis and careful selection of peripheral components can mitigate or reduce the risk of such a failure due to a critical external component. As even a careful selection is unlikely to guarantee uninterrupted operation, the possibility of a full power cycle should be embedded in a concrete satellite design.

Additionally, a total of three unexpected resets of the VA41620 occurred during two of the pion test runs: two during test run 15 and one during test run 13. Unfortunately, the test software's output does not include VA41620's register indicating the previous reset reason. Possible reasons for such a reset are multi-bit errors in the VA41620's memory or SEEs leading to a temporary interruption of the periphery's functionality — for example, a malfunction of a voltage converter causing a brown-out reset of the MCU. Monitoring this register in future radiation tests may provide further insight into these reset's origin. Similar to the previous anomaly, a careful selection of the surrounding components is suggested. As pions are not one of the main sources of radiation in LEO, the probability of them causing a mission failure is lower compared to other sources of radiation. Still, certain missions may require a detailed understanding of this issue and thus further investigation of the root cause of the unexpected resets.

6.5.1 Implications for Radiation Environment in LEO

As mentioned in section 2.3, we expect the worst-case conditions in the SAA. Assuming shielding of 1.7 mm Al equivalent, only protons with a kinetic energy larger than 20 MeV are relevant. Within the SAA, we expect a peak flux of approximately $3 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ for these protons (see figure 2.4).

The VA41620's radiation tests show a worst-case estimate for the expected bit-flip cross-section of $2.3 \cdot 10^{-13}\,\mathrm{cm}^2$. With a proton flux of $3 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ we thus expect a bit-flip rate of $6.3 \cdot 10^{-9}\,\mathrm{B}^{-1}\,\mathrm{s}^{-1}$, which corresponds to one bit-flip within the entire internal memory of the VA4162 approximately every

7.43 h within the SAA. The VA41620's internal scrubbing and TMR registers will correct these bit-flips, thus the bit-flips will not affect the firmware's operation. Therefore, the proton environment of the SAA does not affect the reliable operation of the VA41620.

While pion tests were conducted, the bit-error cross-section due to pions is not relevant for missions in LEO since the pions flux is not significant in this environment.

6.6 STM32L4 Radiation Test Discussion

The STM32L4 MCU acts as reference during development of the DOSIS framework and is a cheap alternative to the VA41620. The radiation test, especially the proton test, provides insight for designers that may choose the STM32L4 for a specific mission. Similar to the VA41620 radiation test, the STM32L4 test does not reveal a trend in the expected number of bit-flips for the tested momentum or LET range. Still, the test results provide an estimated bit-flip cross-section of $9.8 \cdot 10^{-15}\,\mathrm{cm}^2$ to $3.1 \cdot 10^{-13}\,\mathrm{cm}^2$.

Additionally, the observed anomalies indicate general instabilities in the STM32L4's functionality. Due to the missing monitoring of the reset reason register of the STM32L4, a special register indicating why the previous reset occurred, the reason for the unresponsive MCU in test run 4 is unclear. The excessive output during test run 6, on the other hand, enables some insight into potential causes of the observed effect. The output reports numerous memory errors within memory region 1 (see figure 6.6) for both, the read-only and the read-write test patterns. More precisely, the reported errors are in memory region `0x2000 8704` to `0x2001 FFFC` and follow a common pattern: In all reported errors, the same bit — which is part of the most significant nibble — is affected. The observable changes in the most significant nibble change after some time to affect a different bit. Table 6.3 shows the observed bit errors in the beginning and after a short time into the test run. Additionally, sporadic single-bit errors (or multi-bit errors if occurring in combination with the regular pattern) or different changes to the most significant nibble show up in the output. Especially during the transition of the affected bit from the third bit to the first bit a larger number of different changes to the most significant nibble appear.

Table 6.3: Pattern of bit errors in the most-significant nibble during test run 6.

Expected	**Actual**	**Comment**
`0xC` ≡ `0b1100`	`0xE` ≡ `0b1110`	third bit flipped to `1`
`0x3` ≡ `0b0011`	`0x1` ≡ `0b0001`	third bit flipped to `0`
`0xC` ≡ `0b1100`	`0x4` ≡ `0b0100`	first bit flipped to `0`
`0x3` ≡ `0b0011`	`0xB` ≡ `0b1011`	first bit flipped to `1`

It is unlikely that the observed errors are actual bit-flips in memory as this would require the radiation to specifically affect a set of bits spread systematically over a large portion of memory. A functional fault of a component between the MCU's core and the memory better matches the observation. For example, an unreliable data line between memory and core could explain why a certain bit returns what appears to have a random value instead of the actual content of the memory. A full power cycle did resolve the observed issue and the MCU returned to its nominal behavior.

While the exact origin of the observed behavior during test run 6 is unclear, a functional interruption of the controller itself seems more likely than individual bit-flips in memory. Therefore, a mission designer must take this behavior into account. While it only appeared once and thus no reasonable estimate for the likelihood of such an event exists, the effects are severe and may affect a wide range of functionalities. The side effects are arbitrary and may even involve periphery — e.g., due to an unwanted commanding of an actuator or an unwanted memory write operation to persistent memory — which could lead to a permanent impact on the satellite's functionality.

Thus, even if the risk of functional interruption with potential permanent damage is acceptable, every design should include a power cycle mechanism capable of resolving stuck or malfunctioning nodes such as an external watchdog. A software mechanism within the node is not feasible as it cannot

reliably detect those functional interrupts, similar as the test firmware could not distinguish individual bit errors from a malfunctioning memory interface.

6.6.1 Implications for Radiation Environment in LEO

The proton flux in the peak of the SAA in a LEO is approximately $3 \cdot 10^3\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. With the worst-case bit-flip cross-section of $3.1 \cdot 10^{-13}\,\mathrm{cm}^2$ — this is the likelihood observed during the conducted proton tests — we expect a bit-flip rate of $8.6 \cdot 10^{-9}\,\mathrm{B}^{-1}\,\mathrm{s}^{-1}$ for an STM32L4. For the entire RAM of the STM32L496ZG this leads to an uncorrected bit-flip every 1.0 h within the SAA.

Additionally, functional interruptions that could cause effects in other parts of the satellite system were observed (test run 6). A watchdog or other reset circuitry capable of power-cycling the entire node may resolve some of those issues, especially the unresponsive cases. Nevertheless, it cannot avoid permanent effects due to unwanted commands to periphery or memory that may be issued as a consequence of the malfunction. Thus, the STM32L4 is not suggested for LEO missions with required reliable operation inside the SAA. If a mission does not require reliable operation within the SAA, the STM32L4 is still a cheap and power-saving alternative. Still, further tests should provide estimates for the likelihood of such events and their detailed effects.

6.7 Conclusion

The radiation tests demonstrate that the radiation tolerance of the VA41620 is suitable to neglect sporadic bit errors. Due to the observed anomalies, an external watchdog should still be embedded in hardware designs. It should be capable of a full power cycle of all components to mitigate stuck software due to various reasons.

The STM32L4's radiation test, on the other hand, did not provide promising results for operation of the MCU in the SAA. In other regions, especially for missions that do not require operation within the SAA it may still be a reasonable choice assuming an external reset to recover from anomalies exists.

Chapter 7

Discussion

7.1 Fulfilled Objectives

A comparison of this thesis' objectives introduced in section 1.5 and the capabilities of the DOSIS framework and the tested hardware platform enables a better understanding of this thesis' achievements. This comparison utilizes the initially defined design goals and highlights not only the implementation's features but also the future perspectives for its application on real satellite missions.

7.1.1 Distributed System Framework

Objective 1 — Framework Design and Implementation — demands a framework that enables distributed control on a CubeSat based on small nodes without enhanced hardware capabilities. The DOSIS framework designed in this thesis provides the required features with its *Components* and their respective *ComponentInterfaces* that can be used on arbitrary nodes of the satellite. A basic set of *Modules* enables pragmatic declarations and definitions of new *Components* and thus simplifies the development of individual parts of a CubeSat's firmware. Finally, the orchestration of those *Components* to different nodes and their pragmatic interconnection with so-called *deviceDefs* enables a quick setup of distributed CubeSat control applications. As the framework does not rely on enhanced hardware features, the resulting firmware can be deployed on a wide range of low-power nodes as long as these nodes use a 32 bit platform. Currently, the available ports of RODOS, the used OS for the DOSIS framework, limit the use cases to a number of available MCUs.

The design goals presented in section 2.4 act as a guideline for the DOSIS framework's design. Design goal 1 — Component Availability — demands that all components are accessible, at least for CubeSat missions at TUM but preferably even for other academic and commercial applications. The initial selection uses the availability as one of its exclusion criteria (see section 3.1.1) and finally chooses RODOS, which is published under Apache License version 2.0. This license allows the usage of RODOS for arbitrary applications free-of-charge and without any restrictions on the custom additions or modifications made [211]. As the DOSIS framework does not depend on other sources, it is generally available for any academic or commercial application and is itself published under Apache License version 2.0.

Due to the lack of detailed knowledge about the MOVE-III and LRSM mission hard- and software requirements, design goal 2 — Expandability and Flexibility — demands a flexible and expandable system that can be adapted to the specific mission's needs. The DOSIS framework provides this flexibility at all stages of the development. First, it enables development of *Components* independent of the remaining system. Therefore, it is possible to start early with *Component* development and testing. At the same time, the relatively independent *Components* also enable a late change to one *Component* without affecting other parts of the system. A mission's firmware consists of a combination of these *Components* orchestrated to the available nodes. As this orchestration does not require any changes to the *Components* themselves — the *Components* are fully independent of their node assignment unless they require direct hardware access — it also enables simplified redistribution at any time during the development process. Finally, the DOSIS framework does not require a specific number of nodes and

thus can be used in a wide range of systems. A DOSIS setup with a single node and a setup with more than ten nodes are equally possible. Scaling of the node number can not only scale the available computational power or available physical interfaces for external components, but also enables scaling the software to a required level of redundancy. Thus, the DOSIS framework provides all the flexibility required for adaptations at any stage of the development cycle and encourages the reuse of components available from other missions.

The limited electrical power available on a CubeSat is a scarce resource. Thus, design goal 3 — Power Consumption — demands a reasonable use of this resource and encourages power-saving. The RODOS operating system currently supports a few types of MCU, for example ARM M based controllers. Power consumption of ARM M based microcontrollers is typically in the range of a few $10 \cdot 10^1$ mW to $10 \cdot 10^2$ mW, for example, the STM32L496 with a power consumption of typically less than 30 mW [206], which is acceptable for CubeSat and small satellite missions. While the use of RODOS and the low-power ARM M MCUs satisfy design goal 3, the power saving features in RODOS could be increased in the future, e.g., by better using the available sleep and power saving features of the MCUs.

For use in harsh environments, design goal 4 — Reliability — demands embedded measures to improve a system's reliability or at least the support for user-defined mechanisms. Section 3.9 presents a few countermeasures for failed components. Out of the suggested mechanisms, only the fail-over mechanism for time synchronization based on a custom version of the Bully algorithm (see section 3.9.4) is currently implemented. Developers can add other features, such as TMR features for *Components* based on the existing DOSIS infrastructure. Due to the location independent messaging, such a feature can be added without modifying any *Component* but the one requiring said feature. Thus, the DOSIS framework embeds some features, but more importantly enables the development of custom features where required.

The final design goal 5 — System Load — demands a reduction of the load of all resources where possible. As we never experienced high load on the MCU or the CAN bus, no additional effort is required so far. Especially the load introduced due to the DOSIS framework itself is minimal as the framework does not require heavy computation.

Overall, the DOSIS framework provides the required features to support and encourage the development of distributed applications on CubeSats. It provides the flexibility to enable its use on different missions and encourages the reuse of components. As the DOSIS framework's design and implementation is according to the initially formulated design goals, the objective 1 — Framework Design and Implementation — is fully achieved.

7.1.2 ADCS Capability

The second objective of this thesis — Demonstrate ADCS Capability — demands the demonstration of all features required for a successful implementation of a timing sensitive control application based on the DOSIS framework. The suggested example, an ADCS system, is one of the most critical real-time applications for CubeSat software. It requires a reliable sensor readout, a potentially computing intense calculation of the next actuation, and the control of the actuators themselves. In DOSIS, each of these parts would be implemented as an individual *Component*. While they could be deployed to a single node, it only becomes a distributed control system once deployed to a set of different nodes. To still guarantee a synchronized behavior — i.e., a regular readout of all sensor values at a pre-defined point in time, an update of the control output, and a precisely timed actuation of the actuators — DOSIS contains a time synchronization mechanism.

Section 3.8 estimates that an ADCS system requires a time uncertainty of less than 2 ms to avoid significant disturbance of the ADCS system. While section 3.8 presents a few options for such a time synchronization, only a direct CAN message based synchronization can actually satisfy this requirement as otherwise delays will affect the synchronization beyond the acceptable limits (see section 3.8.3). A detailed evaluation and verification of the time synchronization mechanism in chapter 5 shows that this message transfer has the expected behavior. The tests in sections 5.1 and 5.2 show that this time transfer scheme in combination with a PI-controlled local time update satisfies the requirement even for a relatively unstable clock source. Additionally, the verification tests in section 5.2 also show that for

better local clock sources an even simpler direct update mechanism already satisfies the 2 ms constraint. Furthermore, we could show that the timing of DOSIS *Modules* like the *Interval* or *Actuator Modules* meet the 2 ms constraint. As additional traffic on the CAN bus does not affect the 2 ms constraint, more complex control loops or multiple control loops in a single system are also possible unless the local resources of the individual nodes are sufficient for this workload. If the timing constraints are violated due to a high temporary load on a single node, e.g., as many sensors attached to said nodes should be queried at the same time, additional nodes can be added to distribute the load. Thus, ADCS developers can fully rely on the DOSIS built-in *Modules* to read out sensor values in specific intervals and actuate any actuator at a precisely determined point in time.

Overall, the DOSIS framework has shown its capability of synchronizing the local clock of all nodes with a remaining uncertainty of 2 ms. In combination with timed DOSIS *Modules*, an ADCS system can be developed similar to other timing insensitive *Components*. The final orchestration of the ADCS system's DOSIS *Components* to different nodes of the overall system does not negatively affect the overall ADCS system's performance. Thus, the DOSIS framework also fulfills objective 2.

7.1.3 Hardware Platform

The final objective — Demonstrate on Target Hardware — aims at the availability of a hardware platform suitable for CubeSat missions in harsh environments. Of the upcoming missions at TUM, the reliable operation of AFIS within the SAA is the most critical constraint. To fulfil objective 3, a platform that can tolerate the radiation environment in LEO and specifically in the SAA (see section 2.3) is required. This thesis suggests the VA41620 by Vorago Technologies, which provides this capability, as a candidate MCU for AFIS and beyond. The critical design goals — availability, power consumption, and reliability — again provide a better insight into the VA41620's capabilities in contrast to the low-power STM32L4.

The hardware availability is critical for a successful mission. The VA41620 is a COTS ARM M4 based MCU available via common distributors, e.g., Mouser electronics. In contrast to other components suitable for use in space, it is readily available and, with a cost of about 4000 €, reasonable for CubeSat projects at TUM.

Typical low-power ARM M MCUs consume about 10^1 mW to 10^2 mW, for example the STM32L4 with a consumption of less than 30 mW [206]. Due to the used HARDSIL process and the radiation hardening, the VA41620 consumes significantly more power, depending on the load pattern and interface activity, about 140 mW [63]. While this may considerably change a CubeSats power budget, it is still reasonable, especially for a larger 3 U or 6 U CubeSat. Thus, while not optimal regarding the power consumption, the VA41620 is a valid candidate for designers to find a trade-off between power consumption and radiation tolerance.

The most critical design goal for objective 3 is design goal 4 — Reliability. An MCU requires a considerable TID and SEE tolerance for reliable operation in the SAA. The VA41620 tolerates a TID of 3 kGy [227] and can thus operate reliably for extended periods in LEO and beyond. Additionally, it has a high latch-up immunity and provides a bit-error correction in all registers and the embedded RAM. Chapter 6 presents a radiation test conducted at the PSI that verifies the error correction capabilities of the VA41620. During this test no single bit-error could be observed from software. Thus, the reliable operation of the VA41620 is not affected by the radiation environment of the SAA. Additional thermal tests by Biswas [21] suggest reliable operation and startup of the VA41620 at temperatures as low as $-109\,^{\circ}\mathrm{C}$.

Overall, the VA41620 is not a power-saving MCU, but is a viable alternative for applications with an extended demand for reliability. Reliable operation in radiation environments and at extreme temperatures has been demonstrated. Thus, objective 3 is also fulfilled with the VA41620, an ARM M4 based MCU that is well supported by RODOS and the DOSIS framework.

7.2 Simplified and Modular Component Development

As initially presented in section 1.2, the missing modularity of the MOVE-II hardware and software is one of the main reasons why the MOVE-II platform cannot be used for the upcoming missions. The LRSM needs a system that can be easily adapted and scaled for a specific mission while keeping the required workload, and thus the number of required persons, as low as possible. Thus, the DOSIS framework has to be modular to simplify sharing of components between missions while at the same time simplifying the development of individual components.

The DOSIS framework is inherently modular due to its strong separation of interface and implementation of each part. The declaration of a new *Component* is a declaration of its *ComponentInterface*. The *ComponentImplementation*, including the default behavior of the *Component's Modules*, is derived from this *ComponentInterface*. It guarantees that the implementation always matches the interface and assures proper interpretation of messages. With this guarantee, the interface and implementation can be used arbitrarily on different nodes. As all communication uses the *ComponentInterface* only, a *Component's* internal behavior can be adapted to a mission's specific needs without modifications to any other part of the system. Multiple copies of a *Component*, possibly using different versions of the *Component's* implementation, can co-exist on a single mission. These are individually connected to the respective interface instances at compile-time using *deviceDefs*. Overall, this provides a modular architecture that enables the design and implementation based on a combination of available *Components* from other missions — possibly modified to fit the specific mission — and mission specific *Components*.

The effort of developing a DOSIS *Component* or orchestrating *Components* into multiple nodes, especially the complexity managed by developers, is reduced in multiple ways. The declaration of a *Component* combines multiple *Modules* and their respective data types into one *Component*. This directly defines the *Component's* interface and the default behavior thereof. Thus, developers do not have to repeat commonly used patterns, including the communication between a *Component's* interface and its implementation. Extensive unit tests of the DOSIS framework itself assure this networking and default functionalities behavior. Thus, developers do not have to repeatedly test the commonly used functionality and can focus on their *Component's* actual functionality. As the use of C++ templates strongly connects a *Component's* interface and its implementation — the interface is a template parameter of the implementation — the compiler guarantees that they actually match and have a common understanding of the internal messaging. Additional names assigned to all *Modules* used within a *Component's* interface enable the access using proper names and thus increases the readability of code. In combination with strongly typed interface methods, DOSIS *Components* reduce risk of misinterpretation of data. C++ types for physical units — an already available extension to the DOSIS framework (see section 7.5.1) — can help to further reduce the risk of misinterpretation. Specifically, they help to avoid misinterpretation of a value due to a wrongly assumed unit — for example interpreting a current given in mA as A and thus resulting in an error of 10^3.

Callbacks provide the modification point for developers to change the behavior of a *Component's* implementation. Due to the used C++ templates, the compiler assures that all required callbacks actually exist, all implemented callbacks use the expected data type in their interfaces, and only callbacks that are actually used are implemented. This avoids a number of software bugs where a developer accidentally implements a callback that is never called instead of the callback actually intended. As the interface of any *Component* can be used even before the *Component's* implementation exists, the development of individual *Components* is relatively independent. This independence gives the developers freedom to implement and test their *Components* even though other parts of the system are still missing.

The final step of creating the firmware for each node on a satellite is the orchestration of the *Components*. This step also connects the *Component* interface instances to the respective *Component* implementation instances. *DeviceDefs* provide the required information for this connection, including the data type of the *Component* interface. Using this data type, the compiler verifies that no wrongful connection — i.e., connection of a *Component* interface to the implementation of a different *Component* — exists within the system. This again avoids potentially hard-to-detect bugs that may result in unexpected behavior due to misinterpreted messages.

An example application demonstrating the simple steps of declaring a *Component*, implementing

this *Component's* specific behavior, and accessing the *Component* via its interface can be found in appendix D.1.

For these reasons, we see that the DOSIS framework simplifies the firmware development. It also simplifies and encourages the reuse of software. Nevertheless, no real CubeSat firmware based on the DOSIS framework was developed so far. Thus, monitoring the impact of the DOSIS framework on the development process is necessary to reveal potential for future improvement.

7.3 DOSIS on ORIGINS LRSM Missions

Because AFIS and ComPol rely on a number of commercial subsystems, only two computational nodes are developed in-house: the central CDH system that controls and gathers telemetry of the satellite and a PDP that controls the scientific payload and processes its data. The remaining primary subsystems — i.e., ADCS, EPS, or communications subsystem — are off-the-shelf components that provide a CAN interface. Figure 7.1 shows this setup and all major data connections. A CAN bus provides the main interconnection to monitor and control the activities of all subsystems. Additional direct connections via Ethernet and TIA/EIA-422 between the communication system and the PDP and CDH, respectively, provide means for the transmission of data at higher rates than the CAN bus is capable of. An additional benefit of this architecture is that it offers redundant paths for the transmission of science data: Nominally, data is transmitted at high rates directly from the PDP to the communications system via Ethernet; in case of problems with this link, data can also be sent at lower rates through an TIA/EIA-422 connection and routed via the CDH.

The IOV-1 experiment as already presented in section 2.1.3 gives us the opportunity to test prototypes of the CDH and PDP nodes. The experiment will have a separate, flight-proven computer that controls and monitors all systems, thus it provides greater freedom in testing our new systems.

The CDH's primary task is the monitoring and control of all other subsystems of the satellite bus. A VA41620 equipped with radiation-tolerant memory for housekeeping data collected from all subsystems, for firmware updates, and for configuration parameters for both itself and the PDP is the core of the CDH. In addition, it has its own housekeeping sensors, such as temperature sensors, and provides interfaces for additional periphery — e.g., a receiver for a GNSS receiver or an external real-time clock source. To pave the way for future distributed satellite architectures, two identical copies of the CDH in the IOV-1 experiment verify various soft-redundancy and the distributed-control capabilities of the DOSIS framework.

The primary reason for developing a custom PDP is our scientific payloads' requirement for highly parallel interfaces. An FPGA on the PDP provides this interface. An additional VA41620 MCU handles instrument control, telemetry, and the interface to update firmware from the CDH.

7.3.1 On-Board Software

The CDH and PDP each contain a VA41620 MCU executing a RODOS and DOSIS based firmware. As many parts of the software's tasks and the distribution of the responsibilities between the first two nodes is still undefined, the development relies on the DOSIS framework's capabilities of independent development. So far, a CubeSat Space Protocol (CSP) implementation and a DOSIS *Driver* for the SatLab SRS-4 transceiver were developed as part of an interdisciplinary project supervised by the author of this thesis [147]. Additional components — e.g., in-orbit update capabilities of the VA41620 or access to external MRAM and a file system to simplify on-board telemetry processing and storage of generic data — are beeing actively developed.

In future missions, the CDH shall also be able to run the complex control algorithms required for a distributed EPS or ADCS. The DOSIS frameworks capability to gradually replace other components or entire subsystems as needed enables an iterative development towards these missions. Multiple new missions within the LRSM and its successor Space Missions Laboratory (SML) are expected in the near future. Thus, the development of DOSIS and firmware components based on DOSIS will continue and is expected to further increase the capabilities of the TUM CubeSat bus.

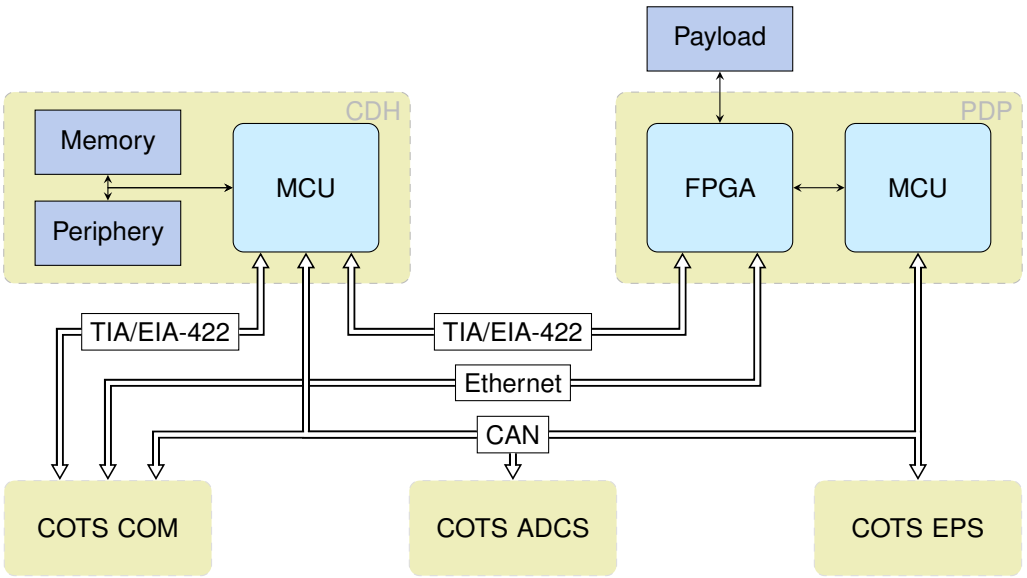


Figure 7.1: Schematic hardware overview of the LRSM satellite bus including core subsystems used for the AFIS and ComPol missions.

7.4 Limitations of Implementation

Although presented in chapter 3, the current implementation of DOSIS does not yet implement all presented features. Especially the support for redundancy management is rudimentary. Multiple commanding entities for a single *Component* are possible, but the developer is fully responsible to take care of any additional synchronization step. While this may be manageable for simple cases, the developer must take special care for contradicting commands in situations where multiple *Daemons* are responsible for a single *Driver*. For more complex cases, e.g., for TMR schemes where the *Driver* would receive an individual command from each *Daemon*, this will be a rather complex and thus error-prone task. Therefore, it is currently suggested to implement a strict chain of command where each DOSIS *Component* receives commands only from a single other entity at a time. If a setup requires multiple commanding entities, e.g., for a TMR setup, a dedicated voting *Component* should be used as an intermediate message end-point that receives all commands, pre-processes them, and only forwards them to the actually commanded DOSIS *Component* thereafter.

Another limitation of the current implementation is the lack of proper access to historic data. All currently available *Modules* return an error code indicating the lack of data if the currently available data was accessed before. Future versions of DOSIS should update this behavior to match the defined behavior of returning the currently available data and tagging this data as historic.

Besides the unimplemented features, two additional issues, revealed during the DOSIS framework's development, must be addressed. First, the current implementation uses a time model independent of the hardware's actual time. Due to the possible changes to the time model's parameters for clock synchronization purposes, a long-term forward propagated value may lead to imprecise timing. This is a result of the one-time conversion of the DOSIS time to a RODOS time for scheduling purposes. Neither DOSIS nor RODOS update the time used for scheduling if the clock parameter changes, e.g., due to a time update message and an updated clock offset and skew.

The second issue is the coexistence with other protocols on a shared CAN bus. While RODOS messages use extended CAN identifiers with a certain prefix, the CSP uses the entire extended CAN identifiers. Thus, RODOS messages will be interpreted in CSP implementations as potentially invalid data and vice versa.

Finally, special care is needed when using the DOSIS framework for purposes other than satellite control. The entire framework is intended for relatively small messages transmitted in regular intervals. DOSIS *Modules* transferring large messages, e.g., for large payload measurements, may lead to unexpected delays of the messaging and may thus lead to timing issues. This is a direct result of the way RODOS fragments messages on the CAN bus. More details about expected delays with RODOS

messages are presented in section 3.8.3. Thus, developers using the DOSIS framework should avoid large data types for any of the currently available DOSIS *Modules*.

7.5 Ongoing Effort and Future Extensions

While this thesis presents the general DOSIS framework at its current state, the active development is still ongoing. A wide range of extensions increase the freedom of designers and developers using the DOSIS framework. While the design and implementation of these extensions may be incomplete, they show the wide range of the ongoing effort and the potential for ongoing use of DOSIS at TUM.

7.5.1 Ongoing Development

All extensions presented in this section are part of student theses supervised by the author of this work.

Checkpoint and Restore

Checkpoint and restore mechanisms provide a way to resume work after a critical failure and may thus be used to increase a system's reliability and reduce the impact of a failed node on the overall system (see section 3.9.5). For this purpose, checkpoints of a running application act as a starting point to resume the application at a known state. A cooperative thread-migration system implemented as an extension to RODOS provides the capabilities of runtime migration of RODOS threads [65, 96] as a first step towards a checkpoint and restore mechanism. While the support for interfaces used by such threads is still limited, e.g., hardware resources are not available for such threads, the communication via RODOS topics is possible for these threads independent of the used node. Currently, this thread-migration system does not support DOSIS *Components* and lacks a mechanism to autonomously detect failed tasks and trigger a migration or restore from a previously stored checkpoint accordingly.

Scripting Language

Scripting languages enable the execution of code without intermediate compilation steps. On small MCUs without MMU they are especially useful as the support for runtime linkage is rare and thus each change of a script would require an update of the entire firmware.

μPython, an open-source[1] Python 3 implementation for embedded devices, enables scripting even on resource constrained MCUs without MMU. First efforts to execute μPython in RODOS were successful [90]. Currently, Python extensions to enable the access to DOSIS *Components* are not available thus no direct communication between Python and other RODOS or DOSIS applications is possible.

Benchmarking

Benchmarking an application enables insight into the resource utilization and potentially reveals performance issues in code. Within a DOSIS system, two resources are critical: the CAN bus utilization and the MCU utilization. Initial benchmark setups based on the built-in performance counters of the ARM M-4 core and the external observation of CAN messages did not reveal any unexpected performance issues [88, 138, 195]. Nevertheless, the DOSIS framework and application development could benefit from a continuous benchmarking. Such a benchmarking, potentially integrated into regular build pipelines, could track the changes over time and reveal unconsidered side effects of any changes to the framework's code.

[1] μPython is available at https://micropython.org/.

Physical Units

The C++ language does not contain direct support for International System of Units (SI) dimensions and units. Instead, sensor readouts or other values calculated from raw measurements use regular data types, i.e., integer or floating point types. While this does not affect the mathematical operations per se, it removes information about the units of these values. Static type checks could use this information to avoid physically illegal operations, such as adding values of different types. Thus, DOSIS includes a physical unit system providing increased type safety[2]. Additional literals increase the readability of the source code and enable direct use of physical constants. While the current implementation already adds a significant benefit to DOSIS, additional support for less common units and added flexibility regarding the C++ data type internally would further encourage developers to use the physical unit type system.

Firmware Updates

Firmware updates are an essential functionality for most CubeSat missions. An update may range from just a small fix or adding a minor script up to an entire update of the firmware image. A safe and secure software update mechanism is currently in development as part of an ongoing Master's thesis [146]. It will provide safe updates that aim at reducing the risk of rendering the system unresponsive. At the same time basic security measurements assure that only software updates from a known source are accepted.

7.5.2 Additional DOSIS Modules

First user applications and their respective DOSIS *Components* show the need for additional DOSIS *Modules* specialized for specific tasks. While the initial set of *Modules* presented in section 3.5 targets sensors, actuators, and the infrastructure required for distributed control, they do not cover all scenarios for payload handling, access to historic housekeeping data, or commanding. They could especially benefit from three additional *Modules*:

- A *Module* enabling larger messages. As previously mentioned, DOSIS currently focuses on short messages, e.g., sensor readouts. Payload configuration or access to time-series of historic housekeeping data often provide or require larger chunks of data at a time. Therefore, a *Module* that enables fragmentation of larger messages simplifies the development of such applications.
- A *Module* with variable length messages. Commands issued remotely by satellite controllers do not have a uniform size. Especially scripts or sequences of commands as often used for satellite control have a widely varying size. As padding them to the maximum possible size is unreasonable — not only regarding the CAN bus and memory utilization but also the involved communication link between ground station and satellite — a special *Module* is required. Additionally, on-board error reporting and debugging messages could greatly benefit from such a *Module*.
- A *Module* with enumerating and/or identifying commands and their responses. While most communication is stateless, some commanding requires a way to identify the response due to a certain command. This identification benefits commanding via a radio link that may lose some messages where the identification of the received responses is essential. It would avoid execution of duplicate messages and enable the access to the previous returned value if it did not reach its destination.

The missing *Modules* show that the DOSIS framework is not complete for all applications. Once an application requires features beyond sensor and actuator access for distributed control, the provided support is limited. While the three suggested additional *Modules* would certainly solve the remaining issues for some applications, future satellite missions and their specific requirements will reveal more applications that require specialized *Modules* or other supportive features. Thus, the ongoing development of the DOSIS framework is essential for its future use in a multitude of space missions.

[2] Developed as part of an Interdisciplinary Project [180].

7.6 Summary

The three main objectives of this thesis are reached: The DOSIS framework provides the necessary support for distributed applications on a resource constrained CubeSat. Additionally, the frameworks can provide suitable timing guarantees for an ADCS control loop. Furthermore, this thesis shows that the VA41620 controller is a radiation tolerant alternative for the LRSM missions, which is also compatible with the DOSIS framework.

Nevertheless, the verification in a real environment is still pending. A verification will be conducted as one of the next setups as part of the IOV-1 LRSM mission. While DOSIS provides the basic infrastructure for distributed control, additional effort is required in the future to widen the applicability of DOSIS — especially for CDH software and a flexible payload interface.

Chapter 8

Conclusion

8.1 Summary

A few CubeSat missions at TUM will be developed and launched in the upcoming years. Not only MOVE-III, a new satellite of the MOVE series, but also the AFIS and ComPol satellites of the ORIGINS LRSM are developed concurrently. With the tight time-line of all of these projects, we identify the lack of a flexible and easy to use software framework encouraging the reuse and sharing between those missions as one of the main problems. We suggest a distributed system that also supports distributed real-time control applications as a potential solution (see section 1.2). As no such distributed system exists in literature, and especially none of the available frameworks directly enables the suggested system, the author suggests the design and development of a custom framework, namely the DOSIS framework (see section 1.4). Thus, the main objective of this thesis is the design and implementation of the DOSIS framework, the verification of its applicability to distributed control applications, and the demonstration of hardware for the target environment (see section 1.5).

Initial design goals based on the upcoming missions — AFIS, ComPol, and MOVE-III — provide a guideline for the design and implementation of the DOSIS framework (see section 2.4). A comparison of existing OSs and on-board software frameworks identifies RODOS in combination with a CAN based network of nodes as the best candidate to provide the basic infrastructure for the DOSIS framework (see sections 3.1 and 3.2). The following design of the DOSIS framework itself is the main achievement of this thesis (see sections 3.3 to 3.7). It presents DOSIS *Components* as the largest building blocks, which in turn are a combination of DOSIS *Modules*. These *Modules* provide pre-defined behavior and enable simplified development of Custom components. While developers can change the behavior via optional callbacks, the strict separation of a *Component's* interface and its implementation enable early testing and modular assembly of the final firmware. It not only enables a modular assembly within a single node, but an orchestration of the *Components* to different nodes. At the same time this modular and flexible design encourages sharing of *Components* between different missions and simplifies the development due to its well-defined *Component* interfaces.

In addition to the DOSIS *Components*, the framework's design also contains various alternatives for a time synchronization of all nodes (see section 3.8). This time synchronization is an essential part of the framework and enables precisely timed and synchronized activities spread over the network of nodes. A few alternatives to further increase the reliability of the overall system are available and different solutions are available for a satellite's designer (see section 3.9).

The implementation (see chapter 4) provides the DOSIS core — i.e., the *Components*, *Modules*, and infrastructure required for their usage — a time synchronization based on a DOSIS internal time model, and a Bully based fail-over for this time synchronization. Additionally, the implementation contains some partially finished features that are still in development and not directly part of this thesis (see section 7.5).

Afterward, a number of tests verify the functionality of the framework and the selected target hardware. A first set of tests specifically verifies the time synchronization and the DOSIS framework's capabilities to guarantee timing for sensor readouts and actuator activations (see chapter 5). These tests verify that the remaining time uncertainty is below $\pm 2\,\text{ms}$ even for a low-quality local clock

reference on each node. Thus, these tests demonstrate the framework's capability to provide the required infrastructure for a distributed time-critical control system. As the AFIS mission requires reliable operation in the SAA, the VA41620 is suggested as microcontroller of the distributed system. A radiation test campaign (see chapter 6) could verify the built-in error correction of said controller. All bit-flips were successfully handled in hardware and no bit error could be directly observed from software. In the SAA, the VA41620's underlying memory is expected to generate about 1 bit-flip every few hours. The built-in error correction will automatically correct this bit-flip and thus no additional software mechanisms recovering corrupted data are required.

8.2 Conclusion

The presented DOSIS framework has been designed and implemented to support the firmware development for the LRSM missions — specifically AFIS and ComPol — and the MOVE-III CubeSat. Furthermore, the framework and the radiation tolerant VA41620 platform fully reach the objectives of this thesis. Achieving these objectives advances the current state of the art with respect to the identified research gaps presented in section 1.4:

Research Gap 1 *Distributed satellite bus control for a resource constrained CubeSat platform.*

Research Gap 2 *Reliable time synchronization on a resource constrained platform.*

Research gap 1 is addressed with the DOSIS framework itself. It demonstrates a distributed control setup based on a number of nodes. The suggested platform is based on ARM M-4 MCUs with a relatively low power consumption and thus suitable for use in a regular CubeSat. With the added time synchronization, implemented entirely in software, the DOSIS framework also advances research gap 2. While this synchronization is not optimized, it shows that such a synchronization is clearly possible. In combination with the failover mechanism based on a simplified Bully algorithm, a reasonable reliable synchronization mechanism is established.

Advancing the state of the art regarding those gaps also supports the hypothesis of this thesis:

Hypothesis 1 *A system based on a network of nodes can improve the flexibility and adaptability of the overall system and allows simple extensions to meet different mission criteria.*

The DOSIS framework provides a simple interface for application development. At the same time, it encourages the reuse of software components between different missions and enables the scaling of the network of nodes based on a specific mission's requirements. In combination with the distributed control advancements, we have shown that a system based on a network of nodes simplifies extensions, modifications, and reuse of existing software. While the DOSIS framework may not be optimal for all applications, it certainly improves the reuse of software between LRSM and MOVE missions and simplifies the adaptation of the software to the specific mission. Thus, hypothesis 1 is confirmed.

Hypothesis 2 *Such a system can meet the timing requirements to implement timing sensitive systems such as a distributed attitude determination and control system.*

We designed the entire DOSIS framework presented in this thesis to support distributed real-time control applications. With objective 2, achieving the required time synchronization accuracy and precision for an ADCS system was one of the main goals. The time synchronization design thus aims at a remaining time uncertainty of less than 2 ms to support an ADCS controller with minimal distortion due to imprecise timing. In chapter 5 we could demonstrate that the DOSIS framework and the implemented time synchronization well achieves this goal. With these tests, we could also demonstrate that the DOSIS framework can meet the timing requirements of an ADCS and scheduled sensor readouts as well as actuator activations can be reliably scheduled within the acceptable remaining uncertainty of 2 ms. Thus, the time synchronization of the DOSIS framework confirms hypothesis 2.

Overall, we successfully demonstrated the viability of a distributed system for CubeSat control. The DOSIS framework at this point not only acts as a demonstrator of a principle, but has its direct

application in the LRSM and MOVE-III. Although the framework requires some additions for a better support of applications other than distributed control, developers can already use the framework to implement first *Components* for later satellite missions. Therefore, this thesis and the current DOSIS framework are only the first step towards a full-featured toolbox for CubeSats with distributed software at TUM.

8.3 Outlook

While this thesis advances the state of the art regarding distributed real-time control on CubeSats, we have not yet reached a state where the DOSIS framework can be used for arbitrary applications. The strong focus on real-time control, which certainly is one of the tasks with strict requirements regarding the system's performance, has lead to a reduced support for other applications, such as generic housekeeping or payload configuration and data management. In future efforts, we will thus extend the framework's capabilities to not only support control applications but also enable simplified development of payload interfaces and advanced CDH systems. Therefore, the implementation of the additions presented in section 7.5 is one of the next steps. Especially the suggested *Modules* for large messages and variable sized messages are critical for the use of DOSIS to implement a CDH system's data handling capabilities.

At the same time, the DOSIS framework will find its first real-world application — other than experimental tests on MOVE-BEYOND high altitude balloons — on the ISS based IOV-1 experiment. The included verification of DOSIS on a VA41620 based platform will provide a better insight into the need for additional features and provide a test platform for distributed reliability measures. At the same time, the active development of the AFIS and ComPol missions will drive the extension of the DOSIS framework's capabilities as previously suggested. Finally, those experiments will verify the framework and its application for generic CubeSat control tasks.

Recent publications show that the general idea of a distributed system on a CubeSat is viable. For example, the CubeSat based Experiment for Space Radiation Analysis [128] also utilizes a distributed system of VA41630 — a version of the VA41620 with embedded code memory — nodes. Therefore, we assume that the DOSIS framework will find its application in various future CubeSat missions at TUM.

References

[1] AAC Clyde Space. *KRYTEN-M3. Command & Data Handling*. Datasheet. July 28, 2020. URL: https://www.aac-clyde.space/wp-content/uploads/2021/10/AAC_DataSheet_Kryten.pdf (visited on 10/26/2022).

[2] AAC Clyde Space. *SIRIUS OBC LEON3FT. Commany & Data Handling*. Datasheet. July 28, 2020. URL: https://www.aac-clyde.space/wp-content/uploads/2021/10/AAC_DataSheet_Sirius-OBC.pdf (visited on 10/26/2022).

[3] AAC Clyde Space. *SIRIUS TCM LEON3FT. Commany & Data Handling*. Datasheet. July 28, 2020. URL: https://www.aac-clyde.space/wp-content/uploads/2021/10/AAC_DataSheet_Sirius-TCM.pdf (visited on 10/26/2022).

[4] Abdullah, M., Al-Kohali, I., and Othman, M. "An Adaptive Bully Algorithm for Leader Elections in Distributed Systems". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 373–384. DOI: 10.1007/978-3-030-25636-4_29.

[5] Adriani, O., Barbarino, G. C., Bazilevskaya, G. A., Bellotti, R., Boezio, M., Bogomolov, E. A., Bongi, M., Bonvicini, V., Borisov, S., Bottai, S., Bruno, A., Cafagna, F., Campana, D., Carbone, R., Carlson, P., Casolino, M., Castellini, G., Consiglio, L., Pascale, M. P. D., Santis, C. D., Simone, N. D., Felice, V. D., Galper, A. M., Gillard, W., Grishantseva, L., Jerse, G., Karelin, A. V., Kheymits, M. D., Koldashov, S. V., Krutkov, S. Y., Kvashnin, A. N., Leonov, A., Malakhov, V., Marcelli, L., Mayorov, A. G., Menn, W., Mikhailov, V. V., Mocchiutti, E., Monaco, A., Mori, N., Nikonov, N., Osteria, G., Palma, F., Papini, P., Pearce, M., Picozza, P., Pizzolotto, C., Ricci, M., Ricciarini, S. B., Rossetto, L., Sarkar, R., Simon, M., Sparvoli, R., Spillantini, P., Stozhkov, Y. I., Vacchi, A., Vannuccini, E., Vasilyev, G., Voronov, S. A., Yurkin, Y. T., Wu, J., Zampa, G., Zampa, N., and Zverev, V. G. "The Discovery of Geomagnetically Trapped Cosmic-Ray Antiprotons". In: *The Astrophysical Journal* 737.2 (July 2011), p. L29. DOI: 10.1088/2041-8205/737/2/L29.

[6] Alaña, E., Carmen Lomba, M. del, Jung, A., Grenham, A., and Fowell, S. "The Avionics SOIS Services of COrDeT On-Board Software Architecture". In: *Proceedings of Data Systems in Aerospace (DASIA) 2013* (May 2013). Porto, Portugal, Aug. 2013.

[7] Amazon Web Services. *The FreeRTOS™ Reference Manual. API Functions and Configuration Options*. Reference Manual. Version 10.0.0 issue 1. 2017. URL: https://www.freertos.org/Documentation/RTOS_book.html (visited on 01/07/2022).

[8] Amazon Web Services, Inc. *FreeRTOS - free professionally developed and robust real time operating system for small embedded systems development*. Oct. 22, 2020. URL: https://www.freertos.org/RTOS.html (visited on 01/10/2022).

[9] Amazon Web Services, Inc. *FreeRTOS Core - FreeRTOS*. Sept. 29, 2021. URL: https://freertos.org/freertos-core/overview.html (visited on 01/10/2022).

[10] Amazon Web Services, Inc. *FreeRTOS Plus TCP - A free thread aware TCP/IP stack for FreeRTOS*. Sept. 12, 2021. URL: https://freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP (visited on 01/10/2022).

[11] Amazon Web Services, Inc. *IoT Libraries - FreeRTOS*. Dec. 20, 2021. URL: https://freertos.org/iot-libraries.html (visited on 01/10/2022).

[12] Barrenscheen, J. *On-Board Communication via CAN without Transceiver*. ApNote 2921. Siemens, Dec. 1996.

[13] Badhwar, G. and O'Neill, P. "Long-term modulation of galactic cosmic radiation and its model for space exploration". In: *Advances in Space Research* 14.10 (Oct. 1994), pp. 749–757. DOI: 10.1016/0273-1177(94)90537-1.

[14] Bannatyne, R., Gifford, D., Klein, K., McCarville, K., Merritt, C., and Neddermeyer, S. "Creation of an ARM® Cortex®-M0 microcontroller for high temperature embedded systems". In: *Additional Conferences (Device Packaging, HiTEC, HiTEN, and CICMT)* 2017.HiTEN (July 2017), pp. 000031–000035. DOI: 10.4071/2380-4491.2017.HITEN.31.

[15] Bannatyne, R., Gifford, D., Klein, K., and Merritt, C. "High temperature / radiation hardened capable ARM® Cortex®-M0 microcontrollers". In: *IMAPS International Conference and Exhibition on High Temperature Electronics, HiTEC 2016* 2016.HiTEC (Jan. 2016), pp. 1–5. DOI: 10.4071/2016-HITEC-46.

[16] Baş, M. E. and Karataş, M. *n-ART OBCOMMS. On Board Computer System*. Interface Control Document Version: 1.1. GUMUSH AeroSpace Ltd., Feb. 18, 2021.

[17] Baş, M. E. and Karataş, M. *N-ART SMC. Sensor and Mechanical Control Board*. Interface Control Document Version: 1.1. GUMUSH AeroSpace Ltd., Feb. 18, 2021.

[18] Bätz, B. "Design and implementation of a framework for spacecraft flight software". en. PhD thesis. Universität Stuttgart, 2020. DOI: 10.18419/OPUS-11205.

[19] Baun, C. *Computer Networks / Computernetze*. Springer Fachmedien Wiesbaden, 2019. DOI: 10.1007/978-3-658-26356-0.

[20] Benninghoff, H., Borchers, K., Börner, A., Fey, G., Gerndt, A., Höflinger, K., Lüdtke, D., Maibaum, O., Peng, T., Schwenk, K., Weps, B., and Westerdorff, K. *OBC-NG Concept and Implementation*. Forschungsbericht. Deutsches Zentrum für Luft- und Raumfahrt, Simulations- und Softwaretechnik, Jan. 20, 2016.

[21] Biswas, J. "Rerating of Electrnic Components for improved Lunar Mission Night Survival". PhD thesis. Technical University of Munich, 2022. unpublished.

[22] Bocchino, R. L. J., Canham, T. K., Watney, G. J., Reder, L. J., and Levison, J. W. "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems". In: *32nd Annual AIAA/USU Conference on Small Satellites*. Logan, UT, USA, 2018, pp. 1–19. URL: https://digitalcommons.usu.edu/smallsat/2018/all2018/328/.

[23] Bök, P.-B., Noack, A., Müller, M., and Behnke, D. *Computernetze und Internet of Things*. Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-29408-3. DOI: 10.1007/978-3-658-29409-0.

[24] Bouwmeester, J., Langer, M., and Gill, E. "Survey on the implementation and reliability of CubeSat electrical bus interfaces". In: *CEAS Space Journal* 9.2 (Sept. 2016), pp. 163–173. DOI: 10.1007/S12567-016-0138-0.

[25] Brandon, C., Chapin, P., Farnsworth, C., and Klink, S. "CubedOS: A Verified CubeSat Operating System". In: *ADA USER* 38.3 (2017), pp. 151–156.

[26] Brandon, C., Chapin, P., Farnsworth, C., and Klink, S. "From Physicist to Rocket Scientist and How to Make a CubeSat That Works". In: *ADA USER* 41.1 (Mar. 2020), pp. 36–42.

[27] Braun, V., Oikonomidou, X., Sanvido, S., and Lemmens, S. "Fostering Collaborative Concepts in Space Debris Mitigation". In: *8th European Conference on Space Debris*. Ed. by Flohrer, T., Lemmens, S., and Schmitz, F. Vol. 8. as. ESA Space Debris Office, 2021. URL: https://conference.sdo.esoc.esa.int/proceedings/sdc8/paper/29.

[28] Bruhn, F. C., Selin, P., Kalnins, I., Lyke, J. C., Rosengren-Calixte, J., and Nordenberg, R. "QuadSat/PnP: A space-plug-and-play architecture (SPA) compliant nanosatellite". In: *AIAA Infotech at Aerospace Conference and Exhibit 2011*. March. Reston, Virigina: American Institute of Aeronautics and Astronautics, Mar. 2011. ISBN: 9781600869440. DOI: 10.2514/6.2011-1575.

[29] Busch, S., Bangert, P., Dombrovski, S., and Schilling, K. "UWE-3, in-orbit performance and lessons learned of a modular and flexible satellite bus for future pico-satellite formations". In: *Acta Astronautica* 117 (Dec. 2015), pp. 73–89. ISSN: 0094-5765. DOI: 10.1016/J.ACTAASTRO.2015.08.002.

[30] Bustos, F. P., Calia, D. B., Budker, D., Centrone, M., Hellemeier, J., Hickson, P., Holzlöhner, R., and Rochester, S. "Remote sensing of geomagnetic fields and atomic collisions in the mesosphere". In: *Nature Communications* 9.1 (Sept. 2018). DOI: 10.1038/S41467-018-06396-7.

[31] California Institute of Technology. *A Quick Look at the Hub Pattern. F Prime User's Guide*. 2020. URL: https://nasa.github.io/fprime/UsersGuide/best/hub-pattern.html (visited on 11/26/2021).

[32] Campola, M. J., Cochran, D. J., Boutte, A. J., Chen, D., Gigliuto, R. A., LaBel, K. A., Pellish, J. A., Ladbury, R. L., Casey, M. C., Wilcox, E. P., O'Bryan, M. V., Lauenstein, J.-M., Violette, D., and Xapsos, M. A. "Compendium of Current Total Ionizing Dose and Displacement Damage for Candidate Spacecraft Electronics for NASA". In: *2015 IEEE Radiation Effects Data Workshop (REDW)*. IEEE, July 2015. DOI: 10.1109/REDW.2015.7336705.

[33] Caroll, K. *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*. Tech. rep. 2RDU00001 Rev C. Lockheed Martin Corporation, Dec. 2015.

[34] Caulfield, J. "Application of Redundant Processing to Space Shuttle". In: *IFAC Proceedings Volumes* 14.2 (Aug. 1981), pp. 2461–2466. ISSN: 1474-6670. DOI: 10.1016/S1474-6670(17)63836-9.

[35] CCSDS 301.0-B-4. *Time Code Formats. Recommended Standard*. Blue Book. Version Issue 4. The Consultative Committee for Space Data Systems, Nov. 2010. URL: https://public.ccsds.org/Pubs/301x0b4e1.pdf.

[36] CCSDS TBD.0-O-0. *CAST Flight Software as a CCSDS Onboard Reference Architecture. Draft Experimental Specification*. Draft Orange Book Draft A. The Consultative Committee for Space Data Systems, Oct. 2016. URL: https://cwe.ccsds.org/sea/docs/SEA-SA/Meeting%20Materials/2016/Fall%202016%20Rome/CAST%20ORANGE%20BOOK.pdf (visited on 11/24/2021).

[37] Cena, G., Bertolotti, I. C., Hu, T., and Valenzano, A. "Performance comparison of mechanisms to reduce bit stuffing jitters in controller area networks". In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE, Sept. 2012. DOI: 10.1109/ETFA.2012.6489559.

[38] Cena, G., Bertolotti, I. C., Hu, T., and Valenzano, A. "A Mechanism to Prevent Stuff Bits in CAN for Achieving Jitterless Communication". In: *IEEE Transactions on Industrial Informatics* 11.1 (Feb. 2015), pp. 83–93. DOI: 10.1109/TII.2014.2365153.

[39] Center, K. B., Fronterhouse, D. C., and Martin, M. "The software strategy for SPA Plug and play spacecraft". In: *2010 IEEE Aerospace Conference*. IEEE, Mar. 2010. DOI: 10.1109/AERO.2010.5446806.

[40] Christensen, J. H. "Space Plug-and-Play Architecture Networking: A Self-Configuring Heterogeneous Network Architecture". PhD thesis. Utah State University, Dec. 2012. DOI: 10.26076/C888-977B.

[41] Chulliat, A., Alken, P., and Nair, M. *The US/UK World Magnetic Model for 2020-2025*. Tech. rep. National Oceanic and Atmoshperic Administration, 2020. DOI: 10.25923/YTK1-YX35.

[42] Chupp, E. L. "Evolution of our understanding of solar flare particle acceleration: (1942–1995)". In: *AIP Conference Proceedings*. AIP, 1996. DOI: 10.1063/1.50997.

[43] CiA 301. *CANopen. Application layer and communication profile*. Tech. rep. CAN in Automation (CiA) e.V., Feb. 2002.

[44] Coelho, C. “A Software Framework for Nanosatellites based on CCSDS Mission Operations Services with Reference Implementation for ESA’s OPS-SAT Mission”. PhD thesis. Graz University of Technology, Nov. 2017. URL: https://diglib.tugraz.at/a-software-framework-for-nanosatellites-based-on-ccsds-mission-operations-services-with-reference-implementation-for-esas-ops-sat-mission-2017.

[45] Coelho, C., Koudelka, O., and Merri, M. “NanoSat MO framework: When OBSW turns into apps”. In: *2017 IEEE Aerospace Conference*. IEEE, Mar. 2017. DOI: 10.1109/AERO.2017.7943951.

[46] Coelho, C., Koudelka, O., and Merri, M. “NanoSat MO Framework: Achieving On-board Software Portability”. In: *SpaceOps 2016 Conference*. American Institute of Aeronautics and Astronautics, May 2016. DOI: 10.2514/6.2016-2624.

[47] Coelho, C., Merri, M., Koudelka, O., and Sarkarati, M. “OPS-SAT Experiments’ Software Management with the NanoSat MO Framework”. In: *AIAA SPACE 2016*. American Institute of Aeronautics and Astronautics, Sept. 2016. DOI: 10.2514/6.2016-5301.

[48] Corley, B. and Steimle, C. “New Bartolomeo Payload Platform on the International Space Station”. In: *AIAA SCITECH 2022 Forum*. American Institute of Aeronautics and Astronautics, Jan. 2022. DOI: 10.2514/6.2022-0860.

[49] Corporation, A. *SAM9260. SMART ARM-based Embedded MPU*. Datasheet. San Jose, CA, USA, Jan. 13, 2016.

[50] Cudmore, A. *Porting the Core Flight System to the Dellingr Cubesat*. Flight Software Workshop 2017. Presentation. Laurel, MD, USA, Dec. 6, 2017. URL: https://ntrs.nasa.gov/citations/20170011566 (visited on 11/24/2021).

[51] Cypress. *S70FL01GS. 1 Gbit (128 Mbyte) 3.0V SPI Flash*. Datasheet. Mar. 21, 2018.

[52] Czech, M., Fleischner, A., and Walter, U. “A First-MOVE in Satellite Development at the TU-München”. In: *Small Satellite Missions for Earth Observation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 235–245. DOI: 10.1007/978-3-642-03501-2_22.

[53] Di Mascio, S., Menicucci, A., Furano, G., Szewczyk, T., Campajola, L., Capua, F. D., Lucaroni, A., and Ottavi, M. “Towards defining a simplified procedure for COTS system-on-chip TID testing”. In: *Nuclear Engineering and Technology* 50.8 (Dec. 2018), pp. 1298–1305. DOI: 10.1016/J.NET.2018.07.010.

[54] Dorman, L. I. and Venkatesan, D. “Solar cosmic rays”. In: *Space Science Reviews* 64.3-4 (1993), pp. 183–362. DOI: 10.1007/BF00750737.

[55] Dubos, G. F., Castet, J.-F., and Saleh, J. H. “Statistical reliability analysis of satellites by mass category: Does spacecraft size matter?” In: *Acta Astronautica* 67.5-6 (Sept. 2010), pp. 584–595. DOI: 10.1016/J.ACTAASTRO.2010.04.017.

[56] ECSS-E-HB-10-12A. *Space engineering - Calculation of radiation and its effects and margin policy handbook*. Handbook. Nordwijk, Netherlands: ESA, Dec. 17, 2010. URL: https://ecss.nl/hbstms/ecss-e-hb-10-12a-calculation-of-radiation-and-its-effects-and-margin-policy-handbook/.

[57] ECSS-E-ST-10-04C. *Space engineering - Space environment*. Standard. Version Rev. 1. Nordwijk, Netherlands: ESA, June 15, 2020. URL: https://ecss.nl/standard/ecss-e-st-10-04c-rev-1-space-environment-15-june-2020/.

[58] ECSS-E-ST-10-12C. *Space engineering - Methods for the calculation of radiation received and its effects, and a policy for design margins*. Standard. Nordwijk, Netherlands: ESA, Nov. 15, 2008. URL: https://ecss.nl/standard/ecss-e-st-10-12c-methods-for-the-calculation-of-radiation-received-and-its-effects-and-a-policy-for-design-margins/.

[59] ECSS-E-ST-50-15C. *Space engineering - CANbus extension protocol*. Standard. Nordwijk, Netherlands: ESA, May 1, 2015. URL: https://ecss.nl/standard/ecss-e-st-50-15c-space-engineering-canbus-extension-protocol-1-may-2015/.

[60] Efron, B. “Bootstrap Methods: Another Look at the Jackknife”. In: *The Annals of Statistics* 7.1 (Jan. 1979). DOI: 10.1214/AOS/1176344552.

[61] EnduroSat. *ONBOARD COMPUTER (OBC)*. 2021. URL: https://www.endurosat.com/cubesat-store/cubesat-obc/onboard-computer-obc/ (visited on 11/17/2021).

[62] Excellence Cluster ORIGINS. *LRSM - The Laboratory for Rapid Space Missions*. 2021. URL: https://www.origins-cluster.de/en/infrastructure/lrsm (visited on 04/20/2022).

[63] Faehling, M. "Development and Evaluation of the MOVE on-board Data Handling System on a Space Qualified Microcontroller". Bachelor's Thesis. Technical University of Munich, Mar. 1, 2021.

[64] Faure, P., Tanaka, A., and Cho, M. "Toward lean satellites reliability improvement using HORYU-IV project as case study". In: *Acta Astronautica* 133 (Apr. 2017), pp. 33–49. DOI: 10.1016/J.ACTAASTRO.2016.12.030.

[65] Föger, M. "Run-Time Migration of Communication enabled Processes in the RODOS Operating System". Bachelor's Thesis. Technical University of Munich, Feb. 15, 2021.

[66] Fontanelli, D. and Macii, D. "Towards master-less wsn clock synchronization with a light communication protocol". In: *2010 IEEE Instrumentation & Measurement Technology Conference Proceedings*. IEEE, 2010. DOI: 10.1109/IMTC.2010.5488010.

[67] Francis, N., Collier, C., and Lyke, J. "Optical Networking for Aerospace Systems Provisioned Through Plug and Play Avionics". In: *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, Apr. 2010. DOI: 10.2514/6.2010-3476.

[68] Fronterhouse, D., Lyke, J., and Achramowicz, S. "Plug-and-play Satellite (PnPSat)". In: *AIAA Infotech@Aerospace 2007 Conference and Exhibit*. American Institute of Aeronautics and Astronautics, May 2007. DOI: 10.2514/6.2007-2914.

[69] Garcia-Molina. "Elections in a Distributed Computing System". In: *IEEE Transactions on Computers* C-31.1 (Jan. 1982), pp. 48–59. ISSN: 0018-9340. DOI: 10.1109/TC.1982.1675885.

[70] Gholipour, M., Kordafshari, M., Jahanshahi, M., and Rahmani, A. "A New Approach for Election Algorithm in Distributed Systems". In: *2009 Second International Conference on Communication Theory, Reliability, and Quality of Service*. IEEE, July 2009, pp. 70–74. ISBN: 978-1-4244-4423-6. DOI: 10.1109/CTRQ.2009.32.

[71] Gonzalez-Velo, Y., Barnaby, H. J., Kozicki, M. N., Gopalan, C., and Holbert, K. "Total Ionizing Dose Retention Capability of Conductive Bridging Random Access Memory". In: *IEEE Electron Device Letters* 35.2 (Feb. 2014), pp. 205–207. DOI: 10.1109/LED.2013.2295801.

[72] Gumush. *N-ART OBCOMMS*. 2021. URL: https://gumush.com.tr/product/n-art-obcomms/ (visited on 11/19/2021).

[73] Hanaway, J. F. and Moorehead, R. W. *Space Shuttle Avionics System*. NASA-SP-504. Washington DC: National Aironatical, Space Administration; Office of Management; Scientific, and Technical Information Division, 1989. URL: https://ntrs.nasa.gov/citations/19900015844.

[74] Haynes, W. M., Lide, D. R., and Bruno, T. J., eds. *CRC Handbook of Chemistry and Physics*. 97th. CRC Press, June 2016. ISBN: 978-1-4987-5429-3. DOI: 10.1201/9781315380476.

[75] Hess, W. "Energetic particles in the inner Van Allen belt". In: *Space Science Reviews* 1.2 (Oct. 1962). DOI: 10.1007/BF00240580.

[76] Igenbergs, E., Hüdepohl, A., Uesugi, K., Hayashi, T., Svedhem, H., Iglseder, H., Koller, G., Glasmachers, A., Grün, E., Schwehm, G., Mizutani, H., Yamamoto, T., Fujimura, A., Ishii, N., Araki, H., Yamakoshi, K., and Nogami, K. "The Munich Dust Counter — A Cosmic Dust Experiment on Board of the Muses-A Mission of Japan". In: *Origin and Evolution of Interplanetary Dust*. Springer Netherlands, 1991, pp. 45–48. DOI: 10.1007/978-94-011-3640-2_9.

[77] Iglseder, H., Münzenmayer, R., Svedhem, H., and Grün, E. "Cosmic dust and space debris measurements with the Munich dust counter on board the satellites hiten and brem-sat". In: *Advances in Space Research* 13.8 (Aug. 1993), pp. 129–132. DOI: 10.1016/0273-1177(93)90579-Z.

[78] Ingalls, J. D., Gadlage, M. J., Wang, J., Williams, A. M., Bruce, D. I., and Ranjan, R. Y. "Total Dose and Heavy Ion Radiation Response of 55 nm Avalanche Technology Spin Transfer Torque MRAM". In: *2019 IEEE Radiation Effects Data Workshop*. IEEE, July 2019. DOI: 10.1109/REDW.2019.8906645.

[79] Innoflight. *CHAMPS Flight Computer (MPSoC CFC-400)*. Datasheet. Jan. 26, 2019.

[80] ISISPACE Group. *ISIS On Board Computer*. 2021. URL: https://www.isispace.nl/product/on-board-computer/ (visited on 11/17/2021).

[81] ISO 11898-1:2015(E). *Road vehicles — controller area network (CAN) — Part 1: Data link layer and physical signalling*. International Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2015.

[82] ISO 11898-2:2016(E). *Road vehicles — controller area network (CAN) — Part 2: High-speed medium access unit*. International Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 15, 2016.

[83] ISO 11898-3:2006(E). *Road vehicles — controller area network (CAN) — Part 3: Low-speed, fault-tolerant, medium-dependent interface*. International Standard. Geneva, Switzerland: International Organization for Standardization, June 1, 2006.

[84] ISO 11898-3:2006/Cor.1:2006(E). *Road vehicles — controller area network (CAN) — Part 3: Low-speed, fault-tolerant, medium-dependent interface — technical corrigendum 1*. International Standard. Geneva, Switzerland: International Organization for Standardization, Dec. 1, 2016.

[85] ISO 11898-4:2004(E). *Road vehicles — controller area network (CAN) — Part 4: Time-triggered communication*. International Standard. Geneva, Switzerland: International Organization for Standardization, Aug. 1, 2004.

[86] ISO/IEC 7498-1:1994(E). *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*. International Standard. International Organization for Standardization, Nov. 1994.

[87] ISO/IEC/IEEE 24765:2017(E). *Systems and software engineering–Vocabulary*. International Standard. International Organization for Standardization / Institute of Electrical and Electronicas Engineers, Sept. 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[88] Johne, N. "Automated Performancetracking of Software for Embedded Systems". Bachelor's Thesis. Technical University of Munich, June 15, 2021.

[89] Kabashi, Q., Zeqiri, A., and Zabeli, M. "The Reduction of Number Messages in Election Bully Algorithm". In: *International Journal of Computers and Communication* 10 (2016), pp. 53–57.

[90] Kale, T. "Dependable and Modular Command and Data Handling Platform for Small Spacecraft using MicroPython on RODOS". Master's Thesis. Technical University of Munich, Sept. 15, 2019. URL: https://mediatum.ub.tum.de/1519584.

[91] KERA. *NanoMind A3200*. Datasheet 1006901. Version 1.17. GomSpace A/S, Jan. 29, 2021. URL: https://gomspace.com/shop/subsystems/command-and-data-handling/nanomind-a3200.aspx (visited on 11/17/2021).

[92] Kief, C. J., Zufelt, B., Cannon, S. R., Lyke, J., and Mee, J. K. "The advent of the PnP Cube satellite". In: *2012 IEEE Aerospace Conference*. IEEE, Mar. 2012. DOI: 10.1109/AERO.2012.6187237.

[93] Kiesbye, J., Messmann, D., Preisinger, M., Reina, G., Nagy, D., Schummer, F., Mostad, M., Kale, T., and Langer, M. "Hardware-In-The-Loop and Software-In-The-Loop Testing of the MOVE-II CubeSat". In: *Aerospace* 6.12 (Dec. 2019), p. 130. DOI: 10.3390/AEROSPACE6120130.

[94] Komori, S. and Sakamoto, Y. "Development Trend of Epoxy Molding Compound for Encapsulating Semiconductor Chips". In: *Materials for Advanced Packaging*. Springer US, 2009, pp. 339–363. DOI: 10.1007/978-0-387-78219-5_10.

[95] Königsmann, H. J., Oelze, H. W., and Rath, H. J. "BREM-SAT - First Flight Results". In: *8th Annual AIAA/USU Conference on Small Satellites*. Sept. 1994. URL: https://digitalcommons.usu.edu/smallsat/1994/all1994/5/.

[96] Konlechner, R. "Design of a Run-Time Migration System for Processes within the RODOS Operating System". Bachelor's Thesis. Technical University of Munich, Sept. 15, 2020.

[97] Kordafshari, M. S., Gholipour, M., Mosakhani, M., Haghighat, A. T., Dehghan, M., Kordafshari, M. S., Gholipour, M., Mosakhani, M., Haghighat, A. T., and Dehghan, M. "Modified bully election algorithm in distributed systems". In: *Proceedings of the 9th WSEAS International Conference on Computers*. ICCOMP'05. Athens, Greece: World Scientific, Engineering Academy, and Society (WSEAS), 2005. ISBN: 9608457297.

[98] KPLabs. *Antelope OBC*. Datasheet. 2021. URL: https://kplabs.space/wp-content/uploads/Anteleope-technical-sheet.pdf (visited on 11/17/2021).

[99] Krasowski, M. J., Prokop, N. F., Flatico, J. M., Greer, L. C., Jenkins, P. P., Neudeck, P. G., Chen, L., and Spina, D. C. "CIB: An Improved Communication Architecture for Real-Time Monitoring of Aerospace Materials, Instruments, and Sensors on the ISS". In: *The Scientific World Journal* 2013 (2013), pp. 1–12. DOI: 10.1155/2013/185769.

[100] Kubos Corporation. *KubOS Design. Kubos 1.21.0 documentation*. 2020. URL: https://docs.kubos.com/1.21.0/kubos-design.html (visited on 11/26/2021).

[101] Kulu, E. *Nanosats Database - Nanosatellite launches with forecast*. Jan. 1, 2022. URL: https://www.nanosats.eu/img/fig/Nanosats_years_forecasts_2022-01-01.pdf (visited on 04/06/2022).

[102] Kuwahara, T., Yoshida, K., Sakamoto, Y., Fukuda, K., Fukuyama, M., and Shibuya, Y. "International Scientific Micro-satellite RISESAT based on Space Plug and Play Avionics". In: *26th Annual AIAA/USU Conference on Small Satellites*. 2012. URL: https://digitalcommons.usu.edu/smallsat/2012/all2012/8/.

[103] Kuwahara, T., Yoshida, K., Sakamoto, Y., Tomioka, Y., and Fukuda, K. "Satellite system integration based on Space Plug and Play Avionics". In: *2011 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, Dec. 2011. DOI: 10.1109/SII.2011.6147568.

[104] Langer, M., Olthoff, C., Harder, J., Fuchs, C., Dziura, M., Hoehn, A., and Walter, U. "Results and lessons learned from the cubesat mission first-move". In: *Small Satellite Missions for Earth Observation*. Berlin: Springer, 2015.

[105] Langer, M. "Reliability Assessment and Reliability Prediction of CubeSats through System Level Testing and Reliability Growth Modelling". PhD thesis. Technische Universität München, 2018. URN: urn:nbn:de:bvb:91-diss-20181008-1446237-1-2. URL: https://mediatum.ub.tum.de/1446237.

[106] Langer, M., Olthoff, C., and Walter, U. *First-MOVE Lessons Learned Report*. Tech. rep. TUM, Lehrstuhl für Raumfahrttechnik, 2014.

[107] Langer, M., Schummer, F., Appel, N., Gruebler, T., Janzer, K., Kiesbye, J., Krempel, L., Lill, A., Messmann, D., Rückerl, S., and Weisgerber, M. "MOVE-II the Munich Orbital Verification Experiment II". In: *Proceedings of the 4th IAA Conference on University Satellite Missions & CubeSat Workshop*. Vol. 163. IAA-AAS-CU-17-06-05. Dec. 2017, pp. 441–459. ISBN: 9780877036470.

[108] Langer, M., Schummer, F., Rückerl, S., Vogel, D., Lill, A., Amann, R., Kale, T., Krempel, L., Kiesbye, J., Lux, P., and Meßmann, D. *MOVE-II System Documentation*. Tech. rep. LRT/WARR, 2019.

[109] Lanza, D., Vick, R., and Lyke, J. "The Space Plug-and-Play Avionics Common Data Dictionary – Constructing the Language of SPA". In: *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, Apr. 2010. DOI: 10.2514/6.2010-3496.

[110] Lapeyrere, V., Lacour, S., David, L., Nowak, M., Crouzier, A., Schworer, G., Perrot, P., and Rayane, S. "PicSat: a Cubesat mission for exoplanetary transit detection in 2017". In: *31st Annual AIAA/USU Conference on Small Satellites*. Logan, UT, USA, 2017. URL: https://digitalcommons.usu.edu/smallsat/2017/all2017/86/.

[111] Lee, S.-H. and Choi, H. “The Fast Bully Algorithm: For Electing a Coordinator Process in Distributed Systems”. In: *Information Networking: Wireless Communications Technologies and Network Applications*. Springer Berlin Heidelberg, 2002, pp. 609–622. DOI: 10.1007/3-540-45801-8_58.

[112] Lehmberg, D. *Press Release: MOVE student group launches stratospheric balloon with new satellite prototype*. Oct. 28, 2021. URL: https://www.move2space.de/blog/press-release-move-student-group-launches-stratospheric-balloon-with-new-satellite-prototype/ (visited on 04/08/2022).

[113] Leppinen, H. “Current use of linux in spacecraft flight software”. In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (Oct. 2017), pp. 4–13. DOI: 10.1109/MAES.2017.160182.

[114] Lill, A., Messmann, D., and Langer, M. “Agile Software Development for Space Applications”. In: *Deutscher Luft- und Raumfahrtkongress*. Deutsche Gesellschaft für Luft- und Raumfahrt, 2017.

[115] Losekamm, M. J., Milde, M., Pöschl, T., Greenwald, D., and Paul, S. “Real-Time Omnidirectional Radiation Monitoring on Spacecraft”. In: *AIAA SPACE 2016*. Long Beach, California: American Institute of Aeronautics and Astronautics, Sept. 9, 2016. DOI: 10.2514/6.2016-5532.

[116] Losekamm, M. J., Milde, M., Pöschl, T., Greenwald, D., and Paul, S. “A new analysis method using Bragg curve spectroscopy for a Multi-purpose Active-target Particle Telescope for radiation monitoring”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 845 (Feb. 11, 2017), pp. 520–523. DOI: 10.1016/J.NIMA.2016.05.029.

[117] Losekamm, M. J., Paul, S., Poschl, T., and Zachrau, H. J. “The RadMap Telescope on the International Space Station”. In: *2021 IEEE Aerospace Conference*. IEEE, Mar. 2021. DOI: 10.1109/AERO50100.2021.9438435.

[118] Lüdtke, D., Westerdorff, K., Stohlmann, K., Borner, A., Maibaum, O., Peng, T., Weps, B., Fey, G., and Gerndt, A. “OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft”. In: *2014 IEEE Aerospace Conference*. IEEE, Mar. 2014, pp. 1–13. ISBN: 978-1-4799-1622-1. DOI: 10.1109/AERO.2014.6836179.

[119] Lund, A., Haj Hammadeh, Z. A., Kenny, P., Vishav, V., Wovalov, A., and Gerndt, A. “A fault-tolerant scalable and distributed middleware for future space missions”. In: *Deutscher Luft- und Raumfahrtkongress 2020* (online). Sept. 1, 2020. URL: https://elib.dlr.de/136450/.

[120] Lunze, P. D. J. *Regelungstechnik 1*. Springer Berlin Heidelberg, 2020. DOI: 10.1007/978-3-662-60746-6.

[121] Lyke, J., Young, Q., Christensen, J., and Anderson, D. “Lessons Learned: Our Decade in Plug-and-play for Spacecraft”. In: *28th Annual AIAA/USU Conference on Small Satellites*. 28. Logan, UT, USA, 2014. URL: https://digitalcommons.usu.edu/smallsat/2014/StandMod/2/.

[122] Lyke, J. “Space-Plug-and-Play Avionics (SPA): A Three-Year Progress Report”. In: *AIAA Infotech@Aerospace 2007 Conference and Exhibit*. American Institute of Aeronautics and Astronautics, May 2007. DOI: 10.2514/6.2007-2928.

[123] Lyke, J., Cannon, S., Fronterhouse, D., Lanza, D., and Byers, T. “A Plug-and-Play System for Spacecraft Components based on the USB Standard”. In: *19th Annual AIAA/USU Conference on Small Satellites*. 28. Logan, UT, USA, 2005. URL: https://digitalcommons.usu.edu/smallsat/2005/all2005/9/.

[124] Lyke, J., Fronterhouse, D., Cannon, S., Lanza, D., and Byers, W. (“Space Plug-and-Play Avionics”. In: *3rd Responsive Space Conference*. Los Angeles, CA, USA: AIAA, Apr. 2005, pp. 1–12.

[125] Mahadeo, D. M., Rohwer, L. E. S., Martinez, M., and Nowlin, R. N. *Assessment of Commercial-Off-The-Shelf Electronics for use in a Short-Term Geostationary Satellite*. Tech. rep. Oct. 2018. DOI: 10.2172/1481565.

[126] Maibaum, O. and Heidecker, A. “Software Evolution from TET-1 to Eu: CROPIS”. In: *10th IAA International Symposium on Small Satellites for Earth Observation*. 10. Berlin: Wissenschaft und Technik Verlag, Apr. 2015, pp. 195–198. ISBN: 978-3-89685-575-6. URL: https://elib.dlr.de/100859/.

[127] "Networking Concepts". In: *Satellite Technology*. Ed. by Maini, A. K. and Agrawal, V. Third Edition. John Wiley & Sons Ltd, Apr. 2014. Chap. 9, pp. 433–470. DOI: 10.1002/9781118636459.CH09.

[128] Maldonado, C. A., Deming, J., Mosley, B. N., Morgan, K. S., McGlown, J., Nelson, A., Fernandes, P. A., Kroupa, M., Katko, K., Hehlen, M. P., Arnold, D., Barney, J., Safi, C., Pyle, M., Schultz, T., Reisenfeld, D., Skoug, R., Guider, A., Holloway, M., Morning, H., Krause, E., Sandoval, B., Beckman, D., Miller, Z., Merl, R., Graham, P. S., White, T. P., Tripp, Z., Hoose, B., Roecker, C., Klimenko, A., Dutch, R., Kaufeld, K., Cox, E., Cole, Q., Clanton, C., Bloser, P., Larsen, B. A., Fairbanks, T., George, J., Michel, J., Alpine, E. L., Kelby, C., and Abbott, B. F. "The Experiment for Space Radiation Analysis: A 12U CubeSat to Explore the Earth's Radiation Belts". In: *2022 IEEE Aerospace Conference*. IEEE, Mar. 2022. DOI: 10.1109/AERO53065.2022.9843239.

[129] Mamun, Q. E. K., Masum, S. M., and Mustafa, M. A. R. "Modified bully algorithm for electing coordinator in distributed systems." In: *WSEAS Transaction on Computers* 3.4 (2004), pp. 948–953.

[130] Maróti, M., Kusy, B., Simon, G., and Lédeczi, Á. "The flooding time synchronization protocol". In: *Proceedings of the 2nd international conference on Embedded networked sensor systems - SenSys '04*. Ed. by Stankovic, J. A., Arora, A., and Govindan, R. ACM Press, 2004, pp. 39–49. ISBN: 1581138792. DOI: 10.1145/1031495.1031501. URL: http://portal.acm.org/citation.cfm?doid=1031495.

[131] McCarthy, D. D. "The Julian and Modified Julian Dates". In: *Journal for the History of Astronomy* 29.4 (Nov. 1998), pp. 327–330. DOI: 10.1177/002182869802900402.

[132] McComas, D. "NASA / GSFC's Flight Software Core Flight System". In: *Flight Software Workshop*. 11. 2012, pp. 1–32. URL: http://docplayer.net/41145319-Nasa-gsfc-s-flight-software-core-flight-system.html%20http://flightsoftware.jhuapl.edu/files/2012/FSW12_McComas.pdf.

[133] McComas, D., Wilmot, J., and Cudmore, A. "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft". In: *30th Annual AIAA/USU Conference on Small Satellites*. 2016. URL: https://digitalcommons.usu.edu/smallsat/2016/TS4AdvTech1/1.

[134] McNutt, C. J., Vick, R., Whiting, H., and Lyke, J. "Modular Nanosatellites: Plug-and-Play (PnP) CubeSat". In: *AIAA 7th Responsive Space Conference*. 2009, pp. 1–14.

[135] Meadows, P. and Cong. *Introduction to Azure RTOS ThreadX*. Mircosoft. Jan. 4, 2021. URL: https://docs.microsoft.com/en-us/azure/rtos/threadx/chapter1 (visited on 01/13/2022).

[136] Meadows, P., Conger, D., and Teebken, T. *Understand Azure RTOS ThreadX*. Microsoft. Aug. 3, 2021. URL: https://docs.microsoft.com/en-us/azure/rtos/threadx/overview-threadx (visited on 01/13/2022).

[137] Meier, M. "TRISTAN Fly High - Design Studies for a CubeSat Compton Telescope". MA thesis. Technical University of Munich, Dec. 2, 2019. URL: https://publications.mppmu.mpg.de/?action=search&mpi=MPP-2019-355.

[138] Melzer, Y. "Benchmarking Utilization of Microcontrollers in Distributed Satellite On-Board Computers". Bachelor's Thesis. Technical University of Munich, Sept. 15, 2020.

[139] Messmann, D., Gruebler, T., Coelho, F., Ohlenforst, T., Bruegge, J. V., Mauracher, F., Doetterl, M., Plamauer, S., Schnierle, P., Kale, T., Seifert, M., Fuhrmann, A., Karagiannis, E., Ulanowski, A., Lausenhammer, T., Meraner, A., and Langer, M. "Advances in the Development of the Attitude Determination and Control System of the CubeSat MOVE-II". In: *Proceedings of the 7th European Conference for Aeronautics and Space Sciences*. 7. Milan, Italy: EUCASS Association, Aug. 2017, pp. 1–15. DOI: 10.13009/EUCASS2017-660.

[140] Mewaldt, R. "Galactic cosmic ray composition and energy spectra". In: *Advances in Space Research* 14.10 (Oct. 1994), pp. 737–747. DOI: 10.1016/0273-1177(94)90536-3.

[141] Microchip Technology Inc. *SAM5D2 Series. SAMA5D21 /22 /23 /24 /26 /27 /28*. Datasheet. 2017.

[142] Microsoft. *Azure RTOS ThreadX Documentation*. Microsoft. 2022. URL: https://docs.microsoft.com/en-us/azure/rtos/threadx/ (visited on 01/13/2022).

[143] Miranda, D. J. F., Ferreira, M., Kucinskis, F., and McComas, D. "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions". In: *Journal of Aerospace Technology and Management* 11 (Oct. 2019). ISSN: 2175-9146. DOI: 10.5028/JATM.V11.1081.

[144] Miroshnichenko, L. *Solar Cosmic Rays*. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-09429-8.

[145] Miroshnichenko, L. I. *Radiation Hazard in Space*. 2003. ISBN: 978-94-017-0301-7. DOI: 10.1007/978-94-017-0301-7.

[146] Molz, P. "Reliable On-Orbit Software Updates for CubeSats". Master's Thesis. Technical University of Munich, Oct. 14, 2022.

[147] Molz, P. "Software Components for the ORIGINS ISS Mission's On-board Computer". Interdisciplinary Project. Technical University of Munich, June 8, 2022.

[148] Montenegro, S. and Aumann, A. *RODOS - Introduction and Documentation*. Jan. 28, 2020. URL: https://gitlab.com/rodos/rodos/-/blob/master/doc/detailed-doc.pdf (visited on 02/03/2022).

[149] Montenegro, S. and Dannemann, F. "RODOS real time kernel design for dependability". In: *Proceedings of Data Systems in Aerospace (DASIA) 2009*. Istanbul: European Space Agency, 2009. URL: https://elib.dlr.de/112377/.

[150] Montenegro, S., Hilgarth, A., Mikschl, T., Ruffer, M., and Walter, T. "VIDANA - An Extremly Fault Tolerant Data Management System for Satellites". In: *International Workshop on Fractional Spacecrafts (IWFS 2014)*. Delft, Netherlands, 2014.

[151] Montenegro, S. and Richardson, J. "RODOS operating system for Network Centric Core Avionics". In: *Proceedings of the First International Conference on Advances in Satellite and Space Communications*. Colmar, France, 2009. URL: https://www.researchgate.net/publication/228955862.

[152] Montenegro, S. and Walter, T. "VIDANA : data management system for nanosatellites". In: *Deutscher Luft- und Raumfahrtkongress*. Deutsche Gesellschaft für Luft- und Raumfahrt, 2013.

[153] Mori, K. "Autonomous decentralized systems: Concept, data field architecture and future trends". In: *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*. IEE Comput. Soc. Press, 1993. DOI: 10.1109/ISADS.1993.262725.

[154] Müller, R. *Operating System Abstraction Layer (OSAL)*. Jan. 13, 2021. URL: https://egit.irs.uni-stuttgart.de/fsfw/fsfw/src/commit/e6a71086141c3bb9602f7afa843402e3f61adcee/doc/README-osal.md (visited on 11/30/2021).

[155] Murshed, M. G. and Allen, A. R. "Enhanced Bully Algorithm for Leader Node Election in Synchronous Distributed Systems". In: *Computers* 1.1 (June 2012), pp. 3–23. DOI: 10.3390/COMPUTERS1010003.

[156] NanoAvionics. *CubeSat On-Board Computer - Main Bus Unit SatBus 3C2*. 2021. URL: https://nanoavionics.com/cubesat-components/cubesat-on-board-computer-main-bus-unit-satbus-3c2/ (visited on 11/17/2021).

[157] Nihon Dempa Kogyo Co. Ltd. *NX3215SA. Crystal Units*. Datasheet. 2003.

[158] Nolte, T., Hansson, H., and Norstrom, C. "Minimizing CAN response-time jitter by message manipulation". In: *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Comput. Soc, Aug. 27, 2002. DOI: 10.1109/RTTAS.2002.1137394.

[159] Oikonomidou, X., Karagiannis, E., Schlaak, M., Hacker, A., Pucknus, P., Pretsch, L., and Iorgulescu, S. "DEDRA On-board MOVE-III: An in-situ detector to support the validation of MASTER". In: *MASTER Modelling Workshop* (Mar. 2, 2021). Ed. by Oikonomidou, X., Braun, V., and Clormann, M. Vol. 1. ESA, Mar. 2, 2021. URL: https://indico.esa.int/event/370/contributions/5911/.

[160] Oikonomidou, X., Braun, V., Pail, R., Gruber, T., Schummer, F., Meßmann, D., Hacker, A., Heapy, J., Martin, L., and Strasser, F. "MOVE-III - An in-situ detector to support space debris model validation". In: *8th European Conference on Space Debris*. Ed. by Flohrer, T., Lemmens, S., and Schmitz, F. Vol. 8. as. ESA Space Debris Office, 2021. URL: https://conference.sdo.esoc.esa.int/proceedings/sdc8/paper/17.

[161] Oikonomidou, X., Karagiannis, E., Still, D., Strasser, F., Firmbach, F. S., Hettwer, J., Schweinfurth, A. G., Pucknus, P., Menekay, D., You, T., Vovk, M., Weber, S., and Zhu, Z. "MOVE-III: A CubeSat for the detection of sub-millimetre space debris and meteoroids in Low Earth Orbit". In: *Frontiers in Space Technologies* 3 (Oct. 2022). DOI: 10.3389/FRSPT.2022.933988.

[162] OrbAstro. *TELOS OBC*. 2021. URL: https://orbastro.com/subsystems/telos-obc/ (visited on 11/19/2021).

[163] Pasetti, A. *The COrDeT Framework*. User Manual. Version 1.0. PP-UM-COR-0002. P&P Software GmbH, Mar. 22, 2019. URL: https://github.com/pnp-software/cordetfw/blob/master/doc/um/UserManual.pdf (visited on 11/25/2021).

[164] Pasetti, A. *The COrDeT Framework*. Definition. Version 2.1. PP-DF-COR-0002. P&P Software GmbH, May 25, 2021. URL: https://github.com/pnp-software/cordetfw/blob/master/doc/cordetfw/cordetfw.pdf (visited on 11/25/2021).

[165] Paul Scherrer Institut. *Secondary beam lines at PSI*. URL: https://www.psi.ch/en/sbl/secondary-beamlines (visited on 08/30/2021).

[166] Paul Scherrer Institut. *πM1 Beam Line*. URL: https://www.psi.ch/en/sbl/pim1-beamline (visited on 08/30/2021).

[167] Peng, T., Hoflinger, K., Weps, B., Maibaum, O., Schwenk, K., Ludtke, D., and Gerndt, A. "A Component-Based Middleware for a Reliable Distributed and Reconfigurable Spacecraft Onboard Computer". In: *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. 35. IEEE, Sept. 2016, pp. 337–342. ISBN: 978-1-5090-3513-7. DOI: 10.1109/SRDS.2016.051. URL: http://ieeexplore.ieee.org/document/7794363/.

[168] Petersen, E. *Single Event Effects in Aerospace*. Ed. by Hanzo, L. Hoboken, New Jersey, USA: John Wiley & Sons, 2011. ISBN: 978-0-470-76749-8.

[169] Plasson, A., Cuomo, C., Gabriel, G., Gauthier, N., Gueguen, L., and Malac-Allain, L. "GERICOS: A GENERIC FRAMEWORK FOR THE DEVELOPMENT OF ON-BOARD SOFTWARE". In: *Proceedings of DAta Systems in Aerospace (DASIA) 2016*. Talinn, Estonia, May 2016. URL: https://ui.adsabs.harvard.edu/abs/2016ESASP.736E..39P.

[170] Pöschl, T., Losekamm, M. J., Greenwald, D., and Paul, S. "A Novel CubeSat-Sized Antiproton Detector for Space Applications". In: *Proceedings of The 34th International Cosmic Ray Conference — PoS(ICRC2015)*. Sissa Medialab, Aug. 28, 2016. DOI: 10.22323/1.236.0590.

[171] Pumpkin Space Systems. *Pluggable Processor Module D1 (PPM-D1). Hardware Revision: A*. Datasheet. Oct. 21, 2009. URL: http://www.pumpkininc.com/space/datasheet/710-00527-A_DS_PPM_D1.pdf (visited on 10/26/2022).

[172] Pumpkin Space Systems. *Pluggable Processor Module A1 (PPM-A1). Hardware Revision: B*. Datasheet. Jan. 13, 2010. URL: http://www.pumpkininc.com/space/datasheet/710-00485-B_DS_PPM_A1.pdf (visited on 10/26/2022).

[173] Pumpkin Space Systems. *Pluggable Processor Module A2 (PPM-A2). Hardware Revision: B*. Datasheet. Jan. 13, 2010. URL: http://www.pumpkininc.com/space/datasheet/710-00486-B_DS_PPM_A2.pdf (visited on 10/26/2022).

[174] Pumpkin Space Systems. *Pluggable Processor Module A3 (PPM-A3). Hardware Revision: B*. Datasheet. Sept. 13, 2010. URL: http://www.pumpkininc.com/space/datasheet/710-00516-B_DS_PPM_A3.pdf (visited on 10/26/2022).

[175] Pumpkin Space Systems. *Pluggable Processor Module D2 (PPM-D2). Hardware Revision: A*. Datasheet. Oct. 21, 2010. URL: http://www.pumpkininc.com/space/datasheet/710-00528-A_DS_PPM_D2.pdf (visited on 10/26/2022).

[176] Pumpkin Space Systems. *Pluggable Processor Module B1 (PPM-B1). Hardware Revision: A*. Datasheet. Sept. 5, 2013. URL: http://www.pumpkininc.com/space/datasheet/710-00487-A_DS_PPM_B1.pdf (visited on 10/26/2022).

[177] Pumpkin Space Systems. *Pluggable Processor Module E1 (PPM E1)*. URL: https://www.pumpkinspace.com/store/p129/Pluggable_Processor_Module_E1_%28PPM_E1%29.html (visited on 11/18/2021).

[178] Reames, D. V. *Solar Energetic Particles*. Springer International Publishing, 2021. DOI: 10.1007/978-3-030-66402-2.

[179] Reitz, G., Berger, T., and Matthiae, D. "Radiation exposure in the moon environment". In: *Planetary and Space Science* 74.1 (Dec. 2012), pp. 78–83. DOI: 10.1016/J.PSS.2012.07.014.

[180] Ritter, T. "Phyiscal Unit Type System for Simplified Development of the MOVE On-board Software". Interdisciplinary Project. Technical University of Munich, July 5, 2022.

[181] Rodríguez, A.-I., Ferrero, F., Alaña, E., Jung, A., Panunzio, M., Vardanega, T., and Grenham, A. "The Component Layer of COrDeT On-Board Software Architecture". In: *Proceedings of Data Systems in Aerospace (DASIA) 2021* (May 2012). Dubrovnic, Croatia, Aug. 2012.

[182] Rückerl, S., Appel, N., Klein, R.-D., and Langer, M. "Software-Defined Communication on the Nanosatellite MOVE-II". In: *Proceedings of the 69th International Astronautical Congress*. 69. Bremen, Germany, Oct. 2018.

[183] Rückerl, S., Meßmann, D., Appel, N., Kiesbye, J., Schummer, F., Markus, F., Krempel, L., Kale, T., Lill, A., Reina, G., Schnierle, P., Sebastian, W., Langer, M., and Martin, L. "First Flight Results of the MOVE-II Satellite". In: *33rd Annual AIAA/USU Conference on Small Satellites*. Logan, UT, USA, Aug. 2019. URL: https://digitalcommons.usu.edu/smallsat/2019/all2019/49/.

[184] Rückerl, S., Ukkola, M., Würl, S., and Faehling, M. "Distributed Computing for Modular & Reliable Nanosatellites". In: *2021 IEEE Aerospace Conference*. IEEE, Mar. 2021. DOI: 10.1109/AERO50100.2021.9438474.

[185] Rushby, J. and Miner, P. S. *A Comparison and of Bus and Architectures for and Safety-Critical Embedded and Systems*. Contractor Report NASA/CR-2003-212161. National Aeronautics and Space Administration, Mar. 1, 2003. URL: https://ntrs.nasa.gov/citations/20030032956 (visited on 09/13/2021).

[186] Rutzinger, M., Krempel, L., Salzberger, M., Buchner, M., Hohn, A., Kellner, M., Janzer, K., Zimmermann, C. G., and Langer, M. "On-orbit verification of space solar cells on the CubeSat MOVE-II". In: *2016 IEEE 43rd Photovoltaic Specialists Conference (PVSC)*. Portland, USA: IEEE, June 2016, pp. 2605–2609. ISBN: 978-1-5090-2724-8. DOI: 10.1109/PVSC.2016.7750120.

[187] Saint-Gobain Ceramics & Plastics, Inc. *BC-400,BC-404,BC-408,BC-412,BC-416 Premium Plastic Scintillators*. 2018. URL: https://www.crystals.saint-gobain.com/sites/imdf.crystals.com/files/documents/bc400-404-408-412-416-data-sheet.pdf (visited on 08/30/2021).

[188] Sasaki, S., Igenbergs, E., Ohashi, H., Münzenmayer, R., Naumann, W., Hofschuster, G., Born, M., Färber, G., Fischer, F., Fujiwara, A., Glasmachers, A., Grün, E., Hamabe, Y., Iglseder, H., Kawamura, T., Miyamoto, H., Morishige, K., Mukai, T., Naoi, T., Nogami, K., Schwehm, G., and Svedhem, H. "Observation of interplanetary and interstellar dust particles by Mars Dust Counter (MDC) on board NOZOMI". In: *Advances in Space Research* 29.8 (Apr. 2002), pp. 1145–1153. DOI: 10.1016/S0273-1177(02)00130-8.

[189] Schenato, L. and Gamba, G. "A distributed consensus protocol for clock synchronization in wireless sensor network". In: *2007 46th IEEE Conference on Decision and Control*. IEEE, 2007. DOI: 10.1109/CDC.2007.4434671.

[190] Schmelz, J. T., Reames, D. V., Steiger, R. von, and Basu, S. "Composition of the Solar Corona, Solar Wind, and Soler Energetic Particles". In: *The Astrophysical Journal* 755.1 (July 2012), p. 33. DOI: 10.1088/0004-637X/755/1/33.

[191] Sebastian Eckl David Werner, A. W. u. U. B. “Towards Real-Time Checkpoint/Restore for Migration in L4 Microkernel based Operating Systems”. In: *Proceedings of the 15th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) 2019*. Stuttgart, Germany, 2019. URL: https://ospert19.tudos.org/ospert19-proceedings.pdf.

[192] Sebastian Eckl Daniel Krefft, U. B. “COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions”. In: *Conference on Future Automotive Technology*. 2015. URL: https://mediatum.ub.tum.de/1278622.

[193] Sebastian Eckl Daniel Krefft, U. B. “Migration of Components and Processes as means for dynamic Reconfiguration in Distributed Embedded Real-Time Operating Systems”. In: *Proceedings of the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) 2017*. Duprovnik, Kings Landing, Croatia, 2017. URL: https://ospert2017.snt.uni.lu/content/download/3341/16327/version/2/file/OSPERT2017_proceedings.pdf.

[194] Seltzer, S. and Bergstrom, P. *Stopping-Powers and Range Tables for Electrons, Protons, and Helium Ions, NIST Standard Reference Database 124*. en. 1993. DOI: 10.18434/T4NC7P.

[195] Sennes, A. “Design and Implementation of a Performance Evaluation Concept of the MOVE-Beyond System regarding the On-board Computing System”. Bachelor’s Thesis. Technical University of Munich, May 15, 2020.

[196] Simpson, J. A. “Elemental and Isotopic Composition of the Galactic Cosmic Rays”. In: *Annual Review of Nuclear and Particle Science* 33.1 (Dec. 1983), pp. 323–382. DOI: 10.1146/ANNUREV.NS.33.120183.001543.

[197] Sinclair, D. and Dyer, J. “Radiation Effects on COTSParts in SmallSats”. In: *27th Annual AIAA/USU Conference on Small Satellites*. 2013. URL: https://digitalcommons.usu.edu/smallsat/2013/all2013/69/.

[198] Sklaroff, J. R. “Redundancy Management Technique for Space Shuttle Computers”. In: *IBM Journal of Research and Development* 20.1 (Jan. 1976), pp. 20–28. ISSN: 0018-8646. DOI: 10.1147/RD.201.0020. URL: http://ieeexplore.ieee.org/document/5391157/.

[199] Sommer, P. and Wattenhofer, R. “Gradient clock synchronization in wireless sensor networks”. In: *2009 International Conference on Information Processing in Sensor Network*. 2009, pp. 37–48. URL: https://ieeexplore.ieee.org/abstract/document/5211944.

[200] Sosinsky, B. *Networking Bible*. Vol. 567. John Wiley & Sons, Oct. 2009. URL: https://learning.oreilly.com/library/view/networking-bible/9780470431313/ (visited on 09/13/2021).

[201] Soundarabai, P. B., Sahai, R., J, T., Venugopal, K. R., and Patnaik, L. M. “Improved Bully Election Algorithm for Distributed Systems”. In: *International Journal of Information Processing, 7(4), 43-54, 2013* (Feb. 28, 2014). arXiv: 1403.3255v1 [cs.DC]. URL: http://arxiv.org/abs/1403.3255.

[202] Space Micro. *Proton 200k® Lite Processor Board*. Datasheet. Apr. 9, 2015.

[203] Space Micro. *Cubesat Space Processor (CSP)*. Datasheet. Sept. 5, 2019.

[204] SpaceInventor. *OBC-P3*. 2021. URL: https://space-inventor.com/OBC-P3/ (visited on 11/17/2021).

[205] Spacemanic. *Eddie The Computer. OBC-MSP430*. Datasheet. 2021. URL: https://www.spacemanic.com/files/datasheet/datasheet_eddie_rev3.pdf (visited on 11/19/2021).

[206] STMicroelectronics. *STM32L496xx. Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100 DMPIS, up to 1 MB Flash, 320 KB SRAM, USB OTG FS, audio, external SMPS*. Datasheet. Version Rev 15. Aug. 2021. URL: https://www.st.com/resource/en/datasheet/stm32l496zg.pdf.

[207] STMicroelectronics. *STM32L4 - ARM Cortex M4 ultra low power MCUs*. 2022. URL: https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-ultra-low-power-mcus/stm32l4-series.html (visited on 06/14/2022).

[208] Suhadis, N. M. “Statistical Overview of CubeSat Mission”. In: *Proceedings of International Conference of Aerospace and Mechanical Engineering 2019*. Ed. by Rajendran, P., Mazlan, N. M., Rahman, A. A. A., Suhadis, N. M., Razak, N. A., and Abidin, M. S. Z. Singapore: Springer Singapore, 2020, pp. 563–573. DOI: 10.1007/978-981-15-4756-0_50.

[209] Swain, A. R. and Hansdah, R. “A model for the classification and survey of clock synchronization protocols in WSNs”. In: *Ad Hoc Networks* 27 (Apr. 2015), pp. 219–241. DOI: 10.1016/J.ADHOC.2014.11.021.

[210] Technolam. *FR-4-11PYTL, FR-4-11PYR, FR-4-11PYB*. Datasheet. Mar. 2019.

[211] The Apache Software Foundation. *Apache License, Version 2.0*. Jan. 2004. URL: https://www.apache.org/licenses/LICENSE-2.0.html (visited on 09/08/2022).

[212] The Cubesat Program. *CubeSat Design Specification. (CDS)*. Tech. rep. Version Rev. 13. California Polytechnical State University, 2014.

[213] The Cubesat Program. *CubeSat Design Specification. (CDS)*. Tech. rep. Version Rev. 14.1. California Polytechnical State University, Feb. 2022.

[214] The RTEMS Project. *RTEMS User Manual*. Release 5.1. User Manual. Aug. 26, 2020.

[215] The RTEMS Project. *RTEMS Real Time Operating System (RTOS) | Real-Time and Real Free RTOS*. 2021. URL: https://ww.rtems.org (visited on 01/13/2022).

[216] TIA/EIA-485-A. *Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*. Industry Standard. Telecommunications Industry Association, Mar. 3, 1998.

[217] Tirado-Andrés, F. and Araujo, A. “Performance of clock sources and their influence on time synchronization in wireless sensor networks”. In: *International Journal of Distributed Sensor Networks* 15.9 (Sept. 2019). DOI: 10.1177/1550147719879372.

[218] Treudler, C., Benninghof, H., Borchers, K., Brunner, B., Cremer, J., Dumke, M., Gärtner, T., Höflinger, K., Langwald, J., Lüdtke, D., Peng, T., Risse, E.-A., Schwenk, K., Stelzer, M., Ulmer, M., Vellas, S., and Westerdorff, K. “ScOSA - Scalable On-Board Computing for Space Avionics”. In: *69th International Astronautical Congress (IAC)*. Bremen, Germany, Oct. 2018.

[219] Twiggs, R. “Origin of cubesat”. In: *Small Satellites: Past, Present, and Future, Eds: Helvajian H., Janson SW, The Aerospace Press, El Segundo, California* (2008).

[220] UM10204. *I²C-bus specification and user manual*. User Manual Rev. 7.0. NXP Semiconductors, Oct. 1, 2021.

[221] Van Allen, J. A., Ludwig, G. H., Ray, E. C., and McIlwain, C. E. “Observation of High Intensity Radiation by Satellites 1958 Alpha and Gamma”. In: *Journal of Jet Propulsion* 28.9 (Sept. 1958), pp. 588–592. DOI: 10.2514/8.7396.

[222] Van Allen, J. A. and Frank, L. A. “Radiation Around the Earth to a Radial Distance of 107,400 km.” In: *Nature* 183.4659 (Feb. 1959), pp. 430–434. DOI: 10.1038/183430A0.

[223] Velazco, R., McMorrow, D., and Estela, J. *Radiation Effects on Integrated Circuits and Systems for Space Applications*. 2019. ISBN: 978-3-030-04660-6. DOI: 10.1007/978-3-030-04660-6.

[224] Vermont Technical College. *CubedOS Operating System*. Documentation generated from online sources. 2021. URL: https://github.com/cubesatlab/cubedos/tree/master/doc (visited on 12/15/2021).

[225] Vick, R. and Lyke, J. “Development of a Low Power Space Plug-and-Play Avionics Protocol for Simple Devices”. In: *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, Apr. 2010. DOI: 10.2514/6.2010-3477.

[226] Villela, T., Costa, C. A., Brandão, A. M., Bueno, F. T., and Leonardi, R. “Towards the Thousandth CubeSat: A Statistical Overview”. In: *International Journal of Aerospace Engineering* 2019 (Jan. 2019), pp. 1–13. ISSN: 1687-5966. DOI: 10.1155/2019/5063145.

[227] VORAGO Technologies. *Radiation Hardened VA416X0. 32-Bit Arm® Cortex®-M4 (with FPU) microcontroller manufactured with HARDSIL® technology offering best in class radiation performance and latch-up immunity.* Product Datasheet. Version Rev 1.4. Austin, TX, 2021.

[228] Vorago Technologies. *Radiation Hardened VA416X0. 32-Bit Arm® Cortex®-M4 (with FPU) microcontroller manufactured with HARDSIL® technology offering best in class radiation performance and latch-up immunity.* Preliminary Datasheet. Version 0.99. Austin, TX, Sept. 15, 2020.

[229] Walter, T., Hilgarth, A., Mikschl, T., and Montenegro, S. "VIDANA : A fault tolerant approach for a distributed data management system in nano-satellites". In: *10th Symposium on Small Satellites for Earth Observation*. 2013.

[230] Wilcox, T. and Seidleck, C. *Vorago RC-OBC-1 Single Event Effect Characterization Test Report*. Tech. rep. National Aeronautics and Space Administration1, Aug. 6, 2020. URL: https://ntrs.nasa.gov/api/citations/20205006200/downloads/Wilcox-TR-18-035-RH-OBC-1-2019June01-MGH%20-%20RHOBC1_SEE_Report_20200806_v4.pdf (visited on 06/14/2022).

[231] Wilmot, J. *Projects - CAST Flight Software as a CCSDS Onboard Reference Architecture*. Ed. by (CNSA), C. N. S. A. June 3, 2021. URL: https://cwe.ccsds.org/fm/Lists/Projects/DispForm.aspx?ID=595 (visited on 11/24/2021).

[232] Windriver. *VXWORKS. Redefining the role of the RTOS*. Brochure. Apr. 2021. URL: https://lp.windriver.com/rs/113-TSG-922/images/VxWorks_Redefining-the-role-of-RTOS-v10.pdf (visited on 01/14/2022).

[233] Windriver. *VxWorks. The World's Leading Real-Time Operating System for the Intelligent Edge*. Brochure. Jan. 2021. URL: https://resources.windriver.com/vxworks/vxworks-product-overview (visited on 01/14/2022).

[234] Yaghmour, K., Masters, J., Ben-Yossef, G., and Gerum, P. *Building Embedded Linux Systems*. O'Reilly Media, Inc., Aug. 2008. URL: https://learning.oreilly.com/library/view/building-embedded-linux/9780596529680/ (visited on 10/26/2022).

[235] Yashiro, H., Takahashi, Y., and Fujiwara, T. "Verification of assurance of space on-board distributed computer system". In: *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. IEEE Comput. Soc, 2001. DOI: 10.1109/HASE.2001.966810.

[236] Yashiro, H., Takahashi, Y., Fujiwara, T., and Mori, K. "A high assurance space on-board distributed computer system". In: *2003 IEEE Aerospace Conference*. Vol. 5. IEEE, 2003, pp. 2501–2509. ISBN: 0-7803-7651-X. DOI: 10.1109/AERO.2003.1235175.

[237] Zangemeister, C. *Nutzwertanalyse in der Systemtechnik. eine Methodik zur multidimensionalen Bewertung aund Auswahl von Projektalternativen*. 1970.

List of Publications

[1] Appel, N., Kimpe, A., Kraus, K., Langer, M., Losekamm, M. J., Milde, M., Pöschl, T., Rückerl, S., Schäfer, F., Stromsky, A., and Würl, K. "TDP-3 Vanguard: Verification of a New Communication System for CubeSats on BEXUS 22". In: *23rd ESA Symposium on European Rocket & Balloon programmes and related research*. Visby, Sweden, June 2017.

[2] Appel, N., Rückerl, S., and Langer, M. "Nanolink : a Robust and Efficient Protocol for Small Satellite Radio Links". In: *4S Symposium*. Valetta, Malta, May 2016.

[3] Kale, T., Yang, D., Rückerl, S., and Schulz, M. "Low Power Optimizations for Sensor Data Processing on Cubesats". In: *10th European Cubesat Symposium 2018*. Toulouse, France, Dec. 2018.

[4] Langer, M., Schummer, F., Appel, N., Gruebler, T., Janzer, K., Kiesbye, J., Krempel, L., Lill, A., Messmann, D., Rückerl, S., and Weisgerber, M. “MOVE-II the Munich Orbital Verification Experiment II”. In: *Proceedings of the 4th IAA Conference on University Satellite Missions & CubeSat Workshop*. Vol. 163. IAA-AAS-CU-17-06-05. Dec. 2017, pp. 441–459. ISBN: 9780877036470.

[5] Langer, M., Schummer, F., Rückerl, S., Vogel, D., Lill, A., Amann, R., Kale, T., Krempel, L., Kiesbye, J., Lux, P., and Meßmann, D. *MOVE-II System Documentation*. Tech. rep. LRT/WARR, 2019.

[6] Losekamm, M. J., Fierlinger, P., Golkar, A., Laurent, P., Manfletti, C., Mertens, S., Paul, S., Reiss, P., Rückerl, S., and Walter, U. “The Space Missions Laboratory at the Technical University of Munich: Rapid Satellite Missions and Instrument Development for Space Research”. In: *Proceedings of the 73rd International Astronautical Confgress*. Paris, France, Sept. 2022.

[7] Losekamm, M. J., Hinderberger, P., and Rückerl, S. “Reliable Data Handling and Processing Systems for Small-Satellite Missions”. In: *2023 IEEE Aerospace Conference*. IEEE, Mar. 2023. DOI: 10.1109/aero55745.2023.10115789.

[8] Rückerl, S., Appel, N., Klein, R.-D., and Langer, M. “Software-Defined Communication on the Nanosatellite MOVE-II”. In: *Proceedings of the 69th International Astronautical Congress*. 69. Bremen, Germany, Oct. 2018.

[9] Rückerl, S. and Losekamm, M. J. “Distributed On-Board Computing on Scientific CubeSat Missions”. In: *Proceedings of the 73rd International Astronautical Confgress*. Paris, France, Sept. 2022.

[10] Rückerl, S. and Losekamm, M. J. “Increased Flexibility and Reliability for CubeSats through Distributed Telemetry and Control. Invited Paper”. In: *20th ACM International Conference on Computing Frontiers (CF ’23)* (May 10, 2023). Bologna, Italy: ACM Press, May 2023. DOI: 10.1145/3587135.3592767.

[11] Rückerl, S., Meßmann, D., Appel, N., Kiesbye, J., Schummer, F., Markus, F., Krempel, L., Kale, T., Lill, A., Reina, G., Schnierle, P., Sebastian, W., Langer, M., and Martin, L. “First Flight Results of the MOVE-II Satellite”. In: *33rd Annual AIAA/USU Conference on Small Satellites*. Logan, UT, USA, Aug. 2019. URL: https://digitalcommons.usu.edu/smallsat/2019/all2019/49/.

[12] Rückerl, S., Ukkola, M., Würl, S., and Faehling, M. “Distributed Computing for Modular & Reliable Nanosatellites”. In: *2021 IEEE Aerospace Conference*. IEEE, Mar. 2021. DOI: 10.1109/AERO50100.2021.9438474.

List of Supervised Theses

[1] Banerjee, A. “Automated Analysis of the MOVE-II Slack for Bugs and Missing Documentation”. Interdisciplinary Project. Technical University of Munich, 2017.

[2] Bernhardt, N. “Implementation of the Central Monitoring and Control Component for the MOVE Groundstation”. Interdisciplinary Project. Technical University of Munich, 2019.

[3] Blohm-Sievers, J. “Development of a Method for Failure Identification and Recovery for MOVE-II Operations”. Semesterthesis. Technical University of Munich, 2018.

[4] Bode, F. “Safe and Secure Software Updates for Cubesats”. Bachelor’s Thesis. Technical University of Munich, Oct. 15, 2021.

[5] Burdurlu, Y. “Radiation Test Software for Embedded Micro-controllers within the MOVE Project”. Interdisciplinary Project. Technical University of Munich, Nov. 25, 2021.

[6] Costescu, C. E. “Implementiton of Backend microservices and parsers for data provided by Satellite Mission MOVE-II”. Interdisciplinary Project. Technical University of Munich, 2017.

[7] Dötterl, M. “Hardening the Linux Kernel against Soft Errors”. Master’s Thesis. Technical University of Munich, Dec. 15, 2017.

[8] Erzar, J. “Porting Electrical Power Subsystem Software onto a new Microcontroller”. Interdisciplinary Project. Technical University of Munich, Aug. 19, 2021.

[9] Faehling, M. “Development and Evaluation of the MOVE on-board Data Handling System on a Space Qualified Microcontroller”. Bachelor’s Thesis. Technical University of Munich, Mar. 1, 2021.

[10] Föger, M. “Run-Time Migration of Communication enabled Processes in the RODOS Operating System”. Bachelor’s Thesis. Technical University of Munich, Feb. 15, 2021.

[11] Föger, M. “Design and Implementation of a Transport Layer Protocol for a Half-Duplex Satellite Link”. Interdisciplinary Project. Technical University of Munich, Sept. 5, 2022.

[12] Hefele, A. “Implementation of the Frontend Graph and Visualizations used for the Satellite Mission MOVE-II”. Interdisciplinary Project. Technical University of Munich, 2017.

[13] Holl, L. “Design eines Mikrocontroller basierten On-Board Flugcomputers für einen CubeSat und Verifikation im Rahmen von MOVE-ON”. Bachelor’s Thesis. Technical University of Munich, 2018.

[14] Johne, N. “Automated Performancetracking of Software for Embedded Systems”. Bachelor’s Thesis. Technical University of Munich, June 15, 2021.

[15] Kale, T. “Event Driven Programming for Embedded Systems”. Guided Research. Technical University of Munich, 2018.

[16] Kale, T. “Dependable and Modular Command and Data Handling Platform for Small Spacecraft using MicroPython on RODOS”. Master’s Thesis. Technical University of Munich, Sept. 15, 2019. URL: https://mediatum.ub.tum.de/1519584.

[17] Konlechner, R. “Design of a Run-Time Migration System for Processes within the RODOS Operating System”. Bachelor’s Thesis. Technical University of Munich, Sept. 15, 2020.

[18] Mauracher, F. “Scalable and Modular Architecture for Self-organizing Power Systems on Small Spacecrafts”. Master’s Thesis. Technical University of Munich, Feb. 15, 2019. URL: https://mediatum.ub.tum.de/1475016.

[19] Melzer, Y. “Benchmarking Utilization of Microcontrollers in Distributed Satellite On-Board Computers”. Bachelor’s Thesis. Technical University of Munich, Sept. 15, 2020.

[20] Molz, P. “Reliable On-Orbit Software Updates for CubeSats”. Master’s Thesis. Technical University of Munich, Oct. 14, 2022.

[21] Molz, P. “Software Components for the ORIGINS ISS Mission’s On-board Computer”. Interdisciplinary Project. Technical University of Munich, June 8, 2022.

[22] Ritter, T. “Phyiscal Unit Type System for Simplified Development of the MOVE On-board Software”. Interdisciplinary Project. Technical University of Munich, July 5, 2022.

[23] Schöttl, F. “Towards energy optimized earth observation missions through modelling data processing architectures for small satellites”. Master’s Thesis. Technical University of Munich, Oct. 15, 2019.

[24] Sennes, A. “Design and Implementation of a Performance Evaluation Concept of the MOVE-Beyond System regarding the On-board Computing System”. Bachelor’s Thesis. Technical University of Munich, May 15, 2020.

[25] Soare, C.-V. “Implementation of Backend microservices and communication tools for the Satellite Mission MOVE-II”. Interdisciplinary Project. Technical University of Munich, 2017.

[26] Winter, L. “Design and Verification of Different Filesystem Abstractions for Space-Travelling Purposes”. Bachelor’s Thesis. Technical University of Munich, May 15, 2021.

[27] Würl, S. “Methodical Comparison of Computer Architectures for Data Processing in Small Satellites”. Master’s Thesis. Technical University of Munich, May 15, 2020.

[28] Zwickl, T. “Implementation of the Frontend and Backend Connectors used for the Satellite Mission MOVE-II”. Interdisiciplinary Project. Technical University of Munich, 2018.

Appendix A

COTS CubeSat OBCs

Table A.1: Available COTS CubeSat on-board computers.

Vendor	Name	Core	Main Memory	Persistent Memory	Remarks	Source
AAC Clyde Space	KRYTEN-M3	ARM Cortex M3 on Smart Fusion 2 SoC @ 50 MHz	unknown	256 kB + 8 MB boot memory 8 MB MRAM	-	[1]
AAC Clyde Space	SIRIUS OBC/TCM	32 bit LEON3-FT core on FPGA @ 50 MHz	64 MB	16 kB non-volatile RAM 2 GB system memory 32 GB mass memory (TCM only)	-	[2, 3]
EnduroSat	OBC	32 bit ARM Cortex M7	512 kB	2 MB program memory 2 SD-card slots	-	[61]
GomSpace	NanoMind A3200	32 bit AT32UC3C @ 64 MHz	32 MB	128 MB flash storage 32 kB FRAM	-	[91]
Gumush	n-ART OBC-COMMS	ARM Cortex M4 (STM32F429) @ 180 MHz	256 kB	2 MB integrated flash 128 MB external flash[i] 256 kB FRAM dual SD-card supported	includes a transceiver	[16, 72]
Gumush	n-ART SMC	ARM Cortex M4 @ 168 MHz	192 kB	unknown	-	[17]
Innoflight	CFC-400	LEON3-FT on MicroSemi FPGA ARM Cortex A53 and R5F on Xilinx MPSoC	2 GB (CPU) + 256 MB (FPGA)	8 MB MRAM 16 GB flash	-	[79]
ISISpace	iOBC	32 bit ARM9[ii] @ 400 MHz	64 MB	1 MB code memory 512 kbit FRAM 4 GB to 32 GB SD-card	-	[80]
KP Labs	Antelope OBC	Dual ARM Cortex R5F @ 300 MHz on RM57 Herkules	unknown	12 MiB MRAM 256 KiB FRAM 1 GiB to 4 GiB flash memory ECC protected code flash	includes FPGA for custom functions	[98]
NanoAvionics	3C2	32 bit ARM Cortex M7 (STM32 H7) @ 400 MHz	1 MB	2 MB integrated flash 2 MB FRAM 256 MB external flash micro SD-card up to 32 Gbit	-	[156]

Vendor	Name	Core	Main Memory	Persistent Memory	Remarks	Source
OrbAstro	TELOS OBC	64 bit ARM A53 & R5 on Xilinx Utrascale+ SoC	up to 64 GB	up to 1 TB flash	dual redundancy optional includes FPGA for custom functions	[162]
Pumpkin Space Systems	PPM-A1 – PPM-A3	16 bit MSP430F @ 7.4 MHz	5 kB to 8 kB	55 kB to 116 kB code memory SD-card support available	-	[172–174]
Pumpkin Space Systems	PPM-B1	8 bit C8051 @ 100 MHz	8448 B	128 kB code memory 128 kB static RAM SD-card support available	-	[176]
Pumpkin Space Systems	PPM-D1	16 bit PIC24 @ 32 MHz	16 kB	256 kB code memory 64 Mbit flash memory SD-card support available	-	[171]
Pumpkin Space Systems	PPM-D2	16 bit dsPIC33 @ 80 MHz	30 kB	256 kB code memory 64 Mbit flash memory SD-card support available	-	[175]
Pumpkin Space Systems	PPM-E1	16 bit PIC24 @ 23 MHz	96 kB	256 kB code memory 64 Mbit flash memory SD-card support available	-	[177]
Space Inventor	OBC-P3	ARM Cortex M7	384 kB	2 MB on-chip flash 32 kB FRAM 64 GB eMMC	all components fully redundant	[204]
Spacemanic	Eddie The Computer	16 bit MSP430[iii]	unknown	256 kbit FRAM code memory 16 Mbit FRAM data storage	-	[205]
Space Micro	CSP	Dual ARM Core on Xilinx Zynq7020	8 Gbit	32 Gbit flash	-	[203]
Space Micro	Proton 200k Lite	TI DSP	512 MB	8 MB EEPROM 32 Gbit flash	-	[202]

[i] Cypress S70FL01GS is used according to [16]. It features 128 MB of memory according to [51]. [ii] AT91SAM9G20 according to [110] [iii] Not mentioned in [205] directly, but assumed due to product identifier *OBC-MSP430*.

Appendix B

Framework and OS Selection

B.1 Criteria

Table B.1: Evaluation guidelines for selection criteria. These guidelines provide a reference for fair evaluation and scoring of frameworks and OSs. A score of 1 indicates a complete satisfaction of a certain criterion, a score of 0 indicates that all factors required for a criterion are not present within the candidate framework or OS.

Criterion	**Evaluation Guidelines**
Open-source (1)	Scoring of criterion 1 is based on the flexibility of the license. A score of 0 is used if no license is provided at all; a score of 1 indicates a license that does not limit the use and enables open-source as well as closed source applications.

Value	*Reason*
0	No license provided.
0.25	A license is provided, but it does not allow use of the software.
0.5	The license allows use in private and/or institutional applications.
0.75	The license allows use in all applications, but restrictions are available; e.g., the source code of a project using the license must be made available with the same license.
1	The license allows use in all applications, including modifications, sharing, using in closed source projects, or any other use of the software without any additional restrictions.

Table B.1 Continued.

Criterion	Evaluation Guidelines
Documentation (2)	This criterion includes only publicly available documentation. The availability of certain pieces of information within the documentation and/or the source code of a framework or OS is the baseline of the evaluation if this criterion. • Installation instructions are available. • Core feature API documentation is available (or self explaining). • API documentation of other features is available (or self explaining). • Examples are available. • The documentation explains the idea behind the framework's or OS's features. The available items as listed above divided by the total number of possible items (5) give the score for this criterion.
Support (3)	The available support is important if changes to the framework or OS should be done. A good community helps to avoid the effort of maintaining a fork of the software due to such changes and can also help to resolve any issues found throughout the development process. Similar to criterion 2, criterion 3 is evaluated based on the availability of: • An active maintainer of the software (last update less than 1 y ago). • A group of people supporting the prime maintainer. • A way to provide feedback and discuss issues exists. • Support by the developers or a community to set up things and solve issues. • Community suggestions are accepted by the maintainer(s). Merge requests (or similar) can be submitted and are processed. The available items as listed above divided by the total number of possible items (5) give the score for this criterion.

Table B.1 Continued.

<table>
<tr><th>Criterion</th><th>Evaluation Guidelines</th></tr>
<tr><td>Ports (4)</td><td>The number of available ports is an indicator for the platforms that can be supported by the framework or OS. Especially the support for smaller target platforms is verified by a port for such a platform. The availability of a port for COTS platforms simplifies the development. A diverse set of available ports increases the chance of a future use of the software. Ports for high level operating systems, such as running the software on top of Linux, reduce the development effort and complexity of tests. A perfect score is given if the following ports are available:
• A port for a larger MCU with MMU.
• A port for a small MCU without MMU.
• A port for a COTS MCU, which simplifies development as inexpensive hardware is available.
• A port for a high level OS (e.g. Linux) or another way to execute applications based on the framework or OS within the development environment.
If only a subset of these ports are available the score is calculated as the number of available items from the list above divided by the total number of items in this list (4).</td></tr>
<tr><td>Porting (5)</td><td>As the effort of porting a framework or OS to a new platform is difficult to estimate, we have to rely on the information available by third parties that performed this before. If no such information is available an estimate of the effort to port a system to a new platform is given based on some criteria within the source code of a framework or OS. If porting of a framework is not directly possible as an underlying OS is used for this purpose, the lowest score is given. Although this does not strictly represent the porting effort of the overall system, it requires the developer to be familiar with an additional piece of software. This additional required knowledge increases the difficulty as a lot of effort is required by the developer to gain this knowledge. The scoring is based on the table below:
<table>
<tr><th>Value</th><th>Reason</th></tr>
<tr><td>0</td><td>Porting is not possible with reasonable effort or would be part of another framework OS used in addition to the currently evaluated framework or OS.</td></tr>
<tr><td>0.25</td><td>Porting is possible, but it has been reported difficult.</td></tr>
<tr><td>0.5</td><td>Porting is possible, but the port is not independent of the framework. Maintaining a port will still be difficult in this case.</td></tr>
<tr><td>0.75</td><td>Porting requires a lot of functions to work, but the code required for a port is independent of the framework itself. Therefore, maintaining the port is simplified.</td></tr>
<tr><td>1</td><td>Porting is simple with only a minimal set of functions required and these are clearly marked or documented.</td></tr>
</table></td></tr>
</table>

Table B.1 Continued.

<table>
<tr><th>Criterion</th><th>Evaluation Guidelines</th></tr>
<tr><td>Multi-threading (6)</td><td>Support of multi-threading is required for a simplified distribution of the applications over a set of nodes. In some cases a node has to execute various applications at the same time or a single application has to be split into several threads. The score calculates as the amount of available multi-threading features in a framework or OS. The list of features used for this evaluation is:
Multiple threads can be executed simultaneously.
Thread creation and termination is possible via a simple interface.
Thread synchronization is available.
Timed execution of threads or code snippets is supported.
Waiting for data availability is available. An example would be an input queue that suspends a thread until data is available.
Support for external interrupts is available (to resume threads or trigger other actions).
If only a part of the features listed are available, the score is reduced proportionally.</td></tr>
<tr><td>Real-time (7)</td><td>Besides the multi-threading features in criterion 6, the scheduler itself is important for real-time systems. Although a soft real-time or best-effort system can be suitable for most applications, it is beneficial for future developments to include hard real-time support. The table below provides the scoring for this criterion:
<table>
<tr><th>Value</th><th>Reason</th></tr>
<tr><td>0</td><td>The framework or OS does not include a scheduler. Even if the used OS below a framework provides a scheduler no points are given. The increased effort of fully understanding the framework and a OS is not desired.</td></tr>
<tr><td>0.25</td><td>The framework or OS includes a simplistic scheduler, e.g., a round-robin based scheduler without any additional features.</td></tr>
<tr><td>0.5</td><td>The framework or OS includes a complex scheduler, including at least priorities based scheduling.</td></tr>
<tr><td>0.75</td><td>The included scheduler is a real-time scheduler.</td></tr>
<tr><td>1</td><td>The included scheduler is a mixed criticality scheduler combining real-time and best-effort tasks in a single system.</td></tr>
</table></td></tr>
</table>

Table B.1 Continued.

Criterion	Evaluation Guidelines
Message-passing (8)	Inter process communication is essential for a distributed system. Although this could be implemented as part of the suggested framework, basic inter process communication artifacts reduce the development process. The list below provides a guideline for the evaluation of this criterion: • Shared memory message-passing is available (at least within a single node). • Point-to-point message-passing within a single node is available. • Point-to-point message-passing within several nodes is available. • Publisher-subscriber message-passing within a single node is available. • Publisher-subscriber message-passing within a network of nodes is available. The score is proportionally reduced if only a subset of the items above are fulfilled.

B.2 Preference Analysis

Table B.3: Pairwise criteria comparison for preference analysis. Each cell depicts the winner of the comparison of two criteria. A 1 indicates the winner is the criterion of this row, a 0 indicates the winner is the criterion of the column. The diagonal row (pairwise comparison with same criterion) is filled with 1. This offset ensures that no criterion has zero wins and thus would get a weight of 0. The last two columns show for the criterion i, the sum of wins of the criterion N_i in this row and its normalized weight w_i. The normalized weight is calculated as $w_i = \frac{N_i}{\sum_{j=0}^{8} N_i}$.

	i	Open-source	Documentation	Support	Ports	Porting	Multi-threading	Real-time	Message-passing	N_i	w_i
Open-source	1	1	0	0	0	0	0	0	0	1	0.027
Documentation	2	1	1	1	1	0	0	1	1	6	0.16
Support	3	1	0	1	1	0	0	0	1	4	0.11
Ports	4	1	0	0	1	0	0	0	0	2	0.054
Porting	5	1	1	1	1	1	0	0	0	5	0.14
Multi-threading	6	1	1	1	1	1	1	0	1	7	0.19
Real-time	7	1	1	1	1	1	1	1	1	8	0.22
Message-passing	8	1	0	0	1	1	0	0	1	4	0.11

B.3 Framework Scoring

Table B.4: Scoring of cFS based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	Apache License 2.0
Documentation (2)	1	Evaluation based on public documentation available or referenced at https://github.com/nasa/cFS. ✓ install instructions ✓ core features documented ✓ other API documented ✓ examples ✓ background explained
Support (3)	1	Evaluation based on public community at https://github.com/nasa/cFS. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform ✓ direct support ✓ community contributions accepted

Table B.4: Scoring of cFS continued.

Criterion	Score	Reason
Ports (4)	1	Evaluation based on public documentation or references given at https://github.com/nasa/cFS. ✓ larger MCU: via RTEMS or VxWorks ✓ small MCU: via RTEMS or VxWorks ✓ COTS MCU: via RTEMS or VxWorks ✓ high level OS: Linux/Posix supported
Porting (5)	0.25	Porting capability provided via VxWorks and RTEMS porting. Cudmore [50] report that porting to CubeSat hardware is difficult.
Multi-threading (6)	0.33	Features evaluated based on documentation and source code available at https://github.com/nasa/cFS. ✓ basic multi-threading ✓ simple thread handling ✓ thread synchronization available ✓ timed execution supported (system-wide time can be used to support this feature) ✓ waiting for data supported ✗ external interrupts supported (deprecated, thus no point given) The partial support for timed execution (system-wide time is available, but not directly supported) and the partial support for waiting for data (only queues, no other way of waiting for data found) are combined into a full point. Thus, 4 out of 6 possible items are fulfilled.
Real-time (7)	0	No scheduler included.
Message-passing (8)	0.6	Features evaluated based on documentation and source code available at https://github.com/nasa/cFS. The supported message-passing schemas are: ✓ shared memory ✓ local point-to-point ? point-to-point over several nodes ✓ local publisher-subscriber ? publisher-subscriber over several nodes As no information on usage of the messaging in a network of nodes is available, these items are not counted as fulfilled.

Table B.6: Scoring of COrDeT based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	Mozilla Public License 2.0
Documentation (2)	0.8	Score based on information provided in [163, 164] and the public repository available at https://github.com/pnp-software/cordetfw. ✓ install instructions ✓ core features documented ✓ other API documented, documentation in source code files ✗ examples do not exist in repository, although examples are mentioned in [163] ✓ background explained
Support (3)	0.6	Evaluation based on public repository at https://github.com/pnp-software/cordetfw. ✓ active maintainer ✓ group of people supporting maintainer ✗ feedback/discussion platform, no external participation found and not promoted ✓ direct support (supported by company) ✗ community contributions accepted, only internal merge and pull requests accepted
Ports (4)	0	Based on information available in public repository at https://github.com/pnp-software/cordetfw and [163, 164]. No hardware specific code is available, only requirements on used OS are explained. ✗ larger MCU ✗ small MCU ✗ COTS MCU ✗ high level OS
Porting (5)	0	Porting effort unknown and no demo port available. Thus porting assumed to be not feasible.

Table B.6: Scoring of COrDeT continued.

Criterion	Score	Reason
Multi-threading (6)	0	Not supported as part of the framework, but implicitly assumed to be available. ✗ basic multi-threading ✗ simple thread handling ✗ thread synchronization available ✗ timed execution supported ✗ waiting for data supported ✗ external interrupts supported
Real-time (7)	0	No own scheduler included.
Message-passing (8)	0	Message-passing required in environment, not provided by COrDeT. ✗ shared memory ✗ local point-to-point ✗ point-to-point over several nodes ✗ local publisher-subscriber ✗ publisher-subscriber over several nodes

Table B.8: Scoring of CubedOS based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	0	no license provided
Documentation (2)	0.6	Evaluation based on documentation in [224] and https://github.com/cubesatlab/cubedos ✓ install instructions ✗ core features documented ✗ other API documented ✓ examples ✓ background explained

Table B.8: Scoring of CubedOS continued.

Criterion	Score	Reason
Support (3)	0.4	Based on community in public repository at https://github.com/cubesatlab/cubedos ✓ active maintainer ✓ group of people supporting maintainer ✗ feedback/discussion platform ? direct support ✗ community contributions accepted No point is given for the direct support item, as it is not known nor is it suggested within the repository.
Ports (4)	1	Ports are generally available for all platforms with Ada runtime support. ✓ larger MCU ✓ small MCU ✓ COTS MCU ✓ high level OS
Porting (5)	0	Porting unreasonable if not supported by Ada runtime.
Multi-threading (6)	0	Not supported as part of the framework could be found in any documentation or the source code. It is implicitly assumed to be available. Most likely multi-threading is part of the Ada runtime environment. ✗ basic multi-threading ✗ simple thread handling ✗ thread synchronization available ✗ timed execution supported ✗ waiting for data supported ✗ external interrupts supported
Real-time (7)	0	No own scheduler included.
Message-passing (8)	0.4	Evaluation based on [224] and https://github.com/cubesatlab/cubedos. ✗ shared memory ✓ local point-to-point ✗ point-to-point over several nodes ✓ local publisher-subscriber ✗ publisher-subscriber over several nodes

Table B.10: Scoring of F' based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	Apache License 2.0
Documentation (2)	1	Evaluation based on the information available at https://nasa.github.io/fprime/. ✓ install instructions ✓ core features documented ✓ other API documented ✓ examples ✓ background explained
Support (3)	1	Evaluation based on the community at https://github.com/nasa/fprime/ and https://github.com/fprime-community. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform ✓ direct support ✓ community contributions accepted
Ports (4)	1	Based on ports available at https://github.com/nasa/fprime/ and https://github.com/fprime-community. ✓ larger MCU, e.g., using Linux ✓ small MCU, e.g., using VxWorks or FreeRTOS ✓ COTS MCU same as above ✓ high level OS, Linux and Posix supported
Porting (5)	1	Minimal ports are possible (e.g., community supplied Arduino port: https://github.com/fprime-community/fprime-arduino). The porting process and potential issues are documented at https://nasa.github.io/fprime/UsersGuide/dev/porting-guide.html.

Table B.10: Scoring of F' continued.

Criterion	Score	Reason
Multi-threading (6)	0.92	Available features (partly via OS abstraction) as presented in https://nasa.github.io/fprime/ and the git repository at https://github.com/nasa/fprime. ✓ basic multi-threading ✓ simple thread handling ✓ thread synchronization available ✓ timed execution supported ✓ waiting for data supported ✓ external interrupts supported External interrupts are only partially supported via events. This is used as half an item to calculate the score for this criterion.
Real-time (7)	0	No own scheduler included.
Message-passing (8)	0.6	According to the documentation at https://nasa.github.io/fprime/ publisher-subscriber communication is not supported. The framework is based on a well known communication represented in a directed graph representing point-to-point communication. ✓ shared memory ✓ local point-to-point ✓ point-to-point over several nodes ✗ local publisher-subscriber ✗ publisher-subscriber over several nodes

Table B.12: Scoring of fsfw based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	Apache License 2.0
Documentation (2)	0.6	Documentation evaluated based on [18] and the public repository at https://egit.irs.uni-stuttgart.de/fsfw/fsfw/. ✓ install instructions ✓ core features documented ✗ other API documented, not all features documented and doxygen comments partially incomplete. ✗ examples, link exists but is broken ✓ background explained

Table B.12: Scoring of fsfw continued.

Criterion	Score	Reason
Support (3)	0.8	Community evaluated based on public repository at https://egit.irs.uni-stuttgart.de/fsfw/fsfw/
		✓ active maintainer
		✓ group of people supporting maintainer
		✓ feedback/discussion platform
		✗ direct support (unknown and not advertised)
		✓ community contributions accepted
Ports (4)	1	
		✓ larger MCU using FreeRTOS or RTEMS
		✓ small MCU using FreeRTOS or RTEMS
		✓ COTS MCU using FreeRTOS or RTEMS
		✓ high level OS; Linux port is available
Porting (5)	0.75	Ports to different underlying OS possible and separated from other parts of the framework. Effort depends on used OS.
Multi-threading (6)	1	Evaluated based on repository at https://egit.irs.uni-stuttgart.de/fsfw/fsfw/ and [18]. Features are partily provided via OS abstractions.
		✓ basic multi-threading
		✓ simple thread handling
		✓ thread synchronization available
		✓ timed execution supported
		✓ waiting for data supported
		✓ external interrupts supported
Real-time (7)	0	No own scheduler included.
Message-passing (8)	0.6	According to Bätz [18], message-passing is available but not intended for distributed applications.
		✓ shared memory
		✓ local point-to-point
		✗ point-to-point over several nodes
		✓ local publisher-subscriber
		✗ publisher-subscriber over several nodes

B.4 Operating System Scoring

Table B.14: Scoring of FreeRTOS based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	MIT License
Documentation (2)	1	Evaluation based on public documentation available at https://www.freertos.org and the *The FreeRTOS™ Reference Manual* [7]. ✓ install instructions ✓ core features documented ✓ other API documented ✓ examples ✓ background explained
Support (3)	0.9	Evaluation based on public community at https://github.com/FreeRTOS/FreeRTOS. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform ✓ direct support, paid premium support by Amazon Web Services. ✓ community contributions accepted
Ports (4)	1	Evaluation based on supported devices listed at https://www.freertos.org/RTOS_ports.html and https://www.freertos.org/emulation-simulation/. ✓ larger MCU ✓ small MCU ✓ COTS MCU ✓ high level OS, via Windows or Linux simulator.
Porting (5)	1	Evaluation based on the FreeRTOS kernel repository at https://github.com/FreeRTOS/FreeRTOS-Kernel. Ports for various tool chain and controller combinations exist and demonstrate the minimal effort required to create a new port.

Table B.14: Scoring of FreeRTOS continued.

Criterion	Score	Reason
Multi-threading (6)	0.92	Evaluation based on public documentation available at https://www.freertos.org.
		✓ basic multi-threading
		✓ simple thread handling
		✓ thread synchronization available
		✓ timed execution supported, only delay available, no repeated activation
		✓ waiting for data supported using queues and/or notifications
		✓ external interrupts supported
Real-time (7)	0.75	A real-time scheduler is included in FreeRTOS.
Message-passing (8)	0.2	
		✗ shared memory
		✓ local point-to-point using queues
		✗ point-to-point over several nodes
		✗ local publisher-subscriber
		✗ publisher-subscriber over several nodes, only via over TCP/IP with external message broker. This is not usable in a stand-alone system of smaller MCUs.

Table B.16: Scoring of RODOS based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	1	Apache 2.0
Documentation (2)	1	Evaluation based on documentation available at https://gitlab.com/rodos/rodos/.
		✓ install instructions
		✓ core features documented
		✓ other API documented
		✓ examples
		✓ background explained

Table B.16: Scoring of RODOS continued.

Criterion	Score	Reason
Support (3)	1	Evaluation of community based on GitLab repository at https://gitlab.com/rodos/rodos. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform ✓ direct support, contact provided in repository ✓ community contributions accepted
Ports (4)	1	Evaluation based on ports directly available in GitLab at https://gitlab.com/rodos/rodos. ✓ larger MCU, e.g., RespberryPi 3 ✓ small MCU, e.g., STM32F4/L4 ✓ COTS MCU, e.g., STM32F4 discovery ✓ high level OS, Linux and general Posix ports available
Porting (5)	1	A minimal port consists only of a few required files and implementation of basic functions. A documentation with a list of these functions is available.
Multi-threading (6)	1	Evaluation based on API and examples in GitLab at https://gitlab.com/rodos/rodos. ✓ basic multi-threading ✓ simple thread handling via statically created objects ✓ thread synchronization available e.g., semaphores ✓ timed execution supported via timed loops or relative/absolute sleep intervals ✓ waiting for data supported ✓ external interrupts supported
Real-time (7)	0.75	RODOS includes a preemptive, priority based real-time scheduler.
Message-passing (8)	1	Message-Passing is evaluated based on the primitives provided within the API and documentation at https://gitlab.com/rodos/rodos. ✓ shared memory ✓ local point-to-point ✓ point-to-point over several nodes ✓ local publisher-subscriber ✓ publisher-subscriber over several nodes

Table B.18: Scoring of RTEMS based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	0.75	GNU General Public License v2
Documentation (2)	1	Evaluation of RTEMS documentation based on documentation available at https://docs.rtems.org. ✓ install instructions ✓ core features documented ✓ other API documented ✓ examples provided at https://git.rtems.org/rtems-examples/ ✓ background explained
Support (3)	1	Community evaluation based on information provided at https://www.rtems.org and the git activity at https://git.rtems.org/rtems. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform ✓ direct support via RTEMS discord ✓ community contributions accepted
Ports (4)	0.88	Evaluated based on the available board support packages listed at https://docs.rtems.org/branches/master/user/bsps/index.html. These are cross-checked with the git repository ✓ larger MCU, e.g., Raspberry Pi ✓ small MCU, e.g., STM32F4 ✓ COTS MCU, e.g., the ones named before. ✓ high level OS, only mentioned with QEMU virtualization
Porting (5)	1	Porting of RTEMS is simple and only a few files must be provided. Other board support packages give a great starting point to adapt for own custom ports.
Multi-threading (6)	1	Evaluated based on documentation at https://docs.rtems.org/. ✓ basic multi-threading ✓ simple thread handling ✓ thread synchronization available ✓ timed execution supported ✓ waiting for data supported, e.g., via messages ✓ external interrupts supported

Table B.18: Scoring of RTEMS continued.

Criterion	Score	Reason
Real-time (7)	0.75	Priority based and preemptive scheduling is available. Various settings for the scheduler exist and are documented at https://docs.rtems.org/branches/master/c-user/scheduling-concepts
Message-passing (8)	0.5	Evaluated based on documentation at https://docs.rtems.org/. ✓ shared memory ✓ local point-to-point ✗ point-to-point over several nodes ✓ local publisher-subscriber, local messaging with broadcast is possible, but not exactly a publisher-subscriber based messaging scheme ✗ publisher-subscriber over several nodes

Table B.20: Scoring of ThreadX based on the criteria defined in appendix B.1.

Criterion	Score	Reason
Open-source (1)	0.5	Microsoft Software License Use only allowed for a very limited set of MCUs. In other cases special restrictions apply, limiting the usability of ThreadX for commercial projects.
Documentation (2)	1	Evaluation based on information available at https://github.com/azure-rtos/threadx and the documentation at https://docs.microsoft.com/de-de/azure/rtos, specifically the part concerning ThreadX itself at https://docs.microsoft.com/de-de/azure/rtos/threadx. ✓ install instructions ✓ core features documented ✓ other API documented ✓ examples ✓ background explained
Support (3)	0.8	Evaluated based on the public GitLab project at https://github.com/azure-rtos/threadx. ✓ active maintainer ✓ group of people supporting maintainer ✓ feedback/discussion platform (Stack Overflow) ✗ direct support, only as paid service ✓ community contributions accepted

Table B.20: Scoring of ThreadX continued.

Criterion	Score	Reason
Ports (4)	1	Evaluation based on ports available at https://github.com/azure-rtos/threadx/tree/master/ports. ✓ larger MCU, e.g., ARM A35 ✓ small MCU, e.g., ARM Cortex M3/M4 ✓ COTS MCU, the MCUs mentioned above are available as COTS modules ✓ high level OS supported via Win32 and Linux ports
Porting (5)	0.75	Evaluation based on ports available at https://github.com/azure-rtos/threadx/tree/master/ports. No instructions on how to start a new port and what is needed could be found within the official documentation at https://docs.microsoft.com/de-de/azure/rtos/threadx. Although porting is possible and encapsulated into a single directory, a lot of macros are defined. It is not clear, which ones are necessary for a minimal port.
Multi-threading (6)	1	Evaluation based on the documentation at https://docs.microsoft.com/de-de/azure/rtos/threadx. ✓ basic multi-threading ✓ simple thread handling ✓ thread synchronization available ✓ timed execution supported ✓ waiting for data supported, using queues ✓ external interrupts supported
Real-time (7)	0.75	Evaluation based on the documentation at https://docs.microsoft.com/de-de/azure/rtos/threadx. ThreadX includes a priority based scheduler. The scheduler can operate as preemptive scheduler using time-slices for threads of equal priority. Due to these properties the scheduler is a real-time scheduler.
Message-passing (8)	0.2	Evaluation based on the documentation at https://docs.microsoft.com/de-de/azure/rtos/threadx. ✗ shared memory could be implemented using locking mechanisms, but is not directly supported ✓ local point-to-point possible using queues ✗ point-to-point over several nodes; no messaging over several nodes is mentioned ✗ local publisher-subscriber ✗ publisher-subscriber over several nodes

Appendix C

Time Synchronization

C.1 Proof of Offset Compensation with P-Controlled Clock Update

Assuming a clock $C(t)$ as presented in equation (3.11)

$$C(t) = \begin{cases} o + s \cdot t + d(t) & t \leq t_1 \\ C(t_k) + (s + u(k)) \cdot (t - t_k) + d(t) & t > t_1 \end{cases} \tag{C.1}$$

with

$k \in \mathbb{N}$: identifier of synchronization interval,
t_k : start time of synchronization interval k, and
$u(k)$: skew correction term for synchronization interval k.

Furthermore, we assume the clock's only error is a static offset, i.e.,

$$\begin{aligned} s &= 1 \text{ and} \\ d(t) &= 0\,. \end{aligned} \tag{C.2}$$

Additionally, we assume a regular update interval $T > 0$ defined by two adjacent synchronization time points:

$$T = t_{k+1} - t_k \tag{C.3}$$

Equation (C.1) can now be rewritten by inserting equation (C.2) and extracting a special case for $t = t_k$ to

$$C(t) = \begin{cases} o + t & t \leq t_1 \\ C(t_{k-1}) + T + T \cdot u(k-1) & t = t_k\,, k \geq 2 \\ C(t_k) + (1 + u(k)) \cdot (t - t_k) & \text{otherwise}\,. \end{cases} \tag{C.4}$$

Using the error and clock update functions from equation (3.12)

$$e(k) = t_{\mathrm{rx},k} - C(t_k)\,, \text{ and} \tag{C.5}$$
$$u(k) = K_p \cdot e(k)\,, \tag{C.6}$$

with

K_p : proportional gain, and
$t_{\mathrm{rx},k}$: received reference time after message delay compensation in time update message k,

and assuming a perfect time transfer, thus $t_{\mathrm{rx},k} = t_k$, we can calculate the remaining error after $n \in \mathbb{N}$ synchronization intervals. At the first synchronization, the error calculates as

$$\begin{aligned} e(k=1) &= t_1 - C(t_1) \\ &= t_1 - o - t_1 \\ &= -o\,. \end{aligned} \tag{C.7}$$

The error for $k = n+1$ can be calculated based on the error of the previous step as

$$\begin{aligned} e(k=n+1) &= t_{n+1} - C(t_{n+1}) && \text{definition C.5} \\ &= t_n + T - C(t_{n+1}) && \text{using C.3} \\ &= t_n + T - C(t_n) - T - T \cdot u(n) && \text{using C.4} \\ &= t_n - C(t_n) - T \cdot u(n) \\ &= t_n - C(t_n) - T \cdot K_p \cdot e(n) && \text{using C.6} \\ &= e(n) - T \cdot K_p \cdot e(n) && \text{using C.5} \\ &= e(n) \cdot (1 - T \cdot K_p)\,. \end{aligned} \tag{C.8}$$

Due to the recursive character, $e(k)$ can be rewritten as

$$\begin{aligned} e(k) &= e(1) \cdot (1 - T \cdot K_p)^{k-1} \\ &= -o \cdot (1 - T \cdot K_p)^{k-1} \quad \text{using C.7.} \end{aligned} \tag{C.9}$$

Thus, the error of a clock with static offset only and a P-controlled clock update converges to

$$\begin{aligned} \lim_{k\to\infty} e(k) &= \lim_{k\to\infty} -o \cdot (1 - T \cdot K_p)^{k-1} \\ &= 0 \qquad\qquad \text{if } |1 - T \cdot K_p| < 1\,, \end{aligned} \tag{C.10}$$

otherwise it does not converge. As $T > 0$,

$$|1 - T \cdot K_p| < 1 \quad \forall K_p \in]0, \frac{2}{T}[\,. \tag{C.11}$$

Thus, a P-controlled clock update compensates an offset if and only if $K_p \in]0, \frac{2}{T}[$. ■

C.2 Proof of Remaining Offset due to Skew Compensation with P-Controlled Clock Update

Similar to appendix C.1, we assume a clock $C(t)$ as presented in equation (3.11)

$$C(t) = \begin{cases} o + s \cdot t + d(t) & t \le t_1 \\ C(t_k) + (s + u(k)) \cdot (t - t_k) + d(t) & t > t_1 \end{cases} \tag{C.12}$$

with

$k \in \mathbb{N}$: identifier of synchronization interval,
t_k : start time of synchronization interval k, and
$u(k)$: skew correction term for synchronization interval k.

This time, we assume a clock that only suffers from a skew different to the reference clock. Thus,

$$\begin{aligned} o &= 0 \text{ and} \\ d(t) &= 0\,. \end{aligned} \tag{C.13}$$

Again, we assume a regular update interval $T > 0$ defined by two adjacent synchronization time points:

$$T = t_{k+1} - t_k \tag{C.14}$$

Equation (C.12) can now be rewritten by inserting equation (C.13) and extracting a special case for $t = t_k$ to

$$C(t) = \begin{cases} s \cdot t & t \le t_1 \\ C(t_{k-1}) + T \cdot s + T \cdot u(k-1) & t = t_k\,, k \ge 2 \\ C(t_k) + (s + u(k)) \cdot (t - t_k) & \text{otherwise}\,. \end{cases} \tag{C.15}$$

Using the error and clock update functions from equation (3.12)

$$e(k) = t_{\mathrm{rx},k} - C(t_k), \text{ and} \tag{C.16}$$
$$u(k) = K_p \cdot e(k), \tag{C.17}$$

with

K_p : proportional gain, and
$t_{\mathrm{rx},k}$: received reference time after message delay compensation in time update message k,

and assuming a perfect time transfer, thus $t_{\mathrm{rx},k} = t_k$, we can calculate the remaining error after $n \in \mathbb{N}$ synchronization intervals. At the first synchronization, the error calculates as

$$\begin{aligned} e(k=1) &= t_1 - C(t_1) \\ &= t_1 - s \cdot t_1 \\ &= t_1(1-s). \end{aligned} \tag{C.18}$$

The error for $k = n+1$ can be calculated based on the error of the previous step as

$$\begin{aligned} e(k=n+1) &= t_{n+1} - C(t_{n+1}) && \text{definition C.16} \\ &= t_n + T - C(t_{n+1}) && \text{using C.14} \\ &= t_n + T - C(t_n) - T \cdot s - T \cdot u(n) && \text{using C.15} \\ &= t_n - C(t_n) + T - T \cdot s - T \cdot u(n) \\ &= e(n) + T - T \cdot s - T \cdot u(n) && \text{using C.16} \\ &= e(n) + T - T \cdot s - T \cdot K_p \cdot e(n) && \text{using C.17} \\ &= e(n) \cdot (1 - T \cdot K_p) + T \cdot (1-s). \end{aligned} \tag{C.19}$$

Using this recursive definition, $e(k)$ can be rewritten as

$$\begin{aligned} e(k) &= e(k-1) \cdot (1 - T \cdot K_p) + T \cdot (1-s) \\ &= \Big(\big(e(1) \cdot (1 - T \cdot K_p) + T \cdot (1-s) \big) \\ &\qquad \cdot (1 - T \cdot K_p) + T \cdot (1-s) \Big) \cdot \ldots && \text{repeated } k-1 \text{ times} \\ &= e(1) \cdot (1 - T \cdot K_p)^k \\ &\qquad + T \cdot (1-s) \cdot (1 - T \cdot K_p)^{k-1} \\ &\qquad + T \cdot (1-s) \cdot (1 - T \cdot K_p)^{k-2} \\ &\qquad + \ldots \\ &\qquad + T \cdot (1-s) \cdot (1 - T \cdot K_p) \\ &\qquad + T \cdot (1-s) \\ &= e(1) \cdot (1 - T \cdot K_p)^{k-1} + \sum_{n=0}^{k-2} \big(T \cdot (1-s) \cdot (1 - T \cdot K_p)^n \big) && \text{geometric series} \\ &= e(1) \cdot (1 - T \cdot K_p)^{k-1} + T \cdot (1-s) \cdot \frac{1 - (1 - T \cdot K_p)^{k-3}}{1 - (1 - T \cdot K_p)} && \text{geometric series} \\ &= e(1) \cdot (1 - T \cdot K_p)^{k-1} + T \cdot (1-s) \cdot \frac{1 - (1 - T \cdot K_p)^{k-3}}{T \cdot K_p} \\ &= e(1) \cdot (1 - T \cdot K_p)^{k-1} + (1-s) \cdot \frac{1 - (1 - T \cdot K_p)^{k-3}}{K_p}. \end{aligned} \tag{C.20}$$

Thus, the error of a clock with drift as only error and a P-controlled clock update converges to

$$\begin{aligned} \lim_{k \to \infty} e(k) &= \lim_{k \to \infty} \left(e(1) \cdot (1 - T \cdot K_p)^{k-1} + (1-s) \cdot \frac{1 - (1 - T \cdot K_p)^{k-3}}{K_p} \right) \\ &= \lim_{k \to \infty} \big(e(1) \cdot (1 - T \cdot K_p)^{k-1} \big) + \lim_{k \to \infty} \left((1-s) \cdot \frac{1 - (1 - T \cdot K_p)^{k-3}}{K_p} \right). \end{aligned} \tag{C.21}$$

With

$$\lim_{k\to\infty}\left(e(1)\cdot\underbrace{(1-T\cdot K_p)^{k-1}}_{\to 0 \text{ if } |1-T\cdot K_p|<1}\right)=e(1)\cdot 0=0 \quad \text{if } |1-T\cdot K_p|<1 \tag{C.22}$$

and

$$\lim_{k\to\infty}\left((1-s)\cdot\frac{1-\overbrace{(1-T\cdot K_p)^{k-3}}^{\to 0 \text{ if } |1-T\cdot K_p|<1}}{K_p}\right)=\frac{1-s}{K_p} \quad \text{if } |1-T\cdot K_p|<1\,, \tag{C.23}$$

this can be simplified to

$$\lim_{k\to\infty} e(k)=0+\frac{1-s}{K_p}=\frac{1-s}{K_p} \quad \text{if } |1-T\cdot K_p|<1\,. \tag{C.24}$$

For other values of $|1-T\cdot K_p|$ it does not converge. As $T>0$,

$$|1-T\cdot K_p|<1 \quad \forall K_p\in\left]0,\frac{2}{T}\right[\,. \tag{C.25}$$

Thus, a P-controlled clock update does not fully compensate a clock skew and instead converges to a static offset of $\frac{1-s}{K_p}$ for $K_p\in\left]0,\frac{2}{T}\right[$. ■

C.3 Proof of Separation of Error within P-Controlled Clock Update

To combine the result of appendix C.1 and appendix C.2 we have to show that the error term can be separated, i.e., that

$$e_{\text{combined}}(k)=e_{\text{offset}}(k)+e_{\text{skew}}(k) \quad \forall k\in\mathbb{N}\,. \tag{C.26}$$

Note that if a function or variable name is not unique in the two previous proofs, we will use a subscript to reference the two versions. To proof equation (C.26) we have to show that

$$e_{\text{combined}}(1)=e_{\text{offset}}(1)+e_{\text{skew}}(1) \qquad \text{and} \tag{C.27}$$
$$e_{\text{combined}}(k)=e_{\text{offset}}(k)+e_{\text{skew}}(k) \qquad \forall k\geq 2\,. \tag{C.28}$$

Similar to the previous proofs, we define a clock with offset and skew error as

$$C(t)=\begin{cases} o+s\cdot t & t\leq t_1 \\ C(t_{k-1})+T\cdot s+T\cdot u(k-1) & t=t_k\,, k\geq 2 \\ C(t_k)+(s+u(k))\cdot(t-t_k) & \text{otherwise}\,, \end{cases} \tag{C.29}$$

Note that only the initial case is different to equation (C.15). The generic definition of the error and clock update functions and the interval T are identical to those of both previous proofs:

$$e(k)=t_{\text{rx},k}-C(t_k) \qquad \text{identical to C.5 and C.16,} \tag{C.30}$$
$$u(k)=K_p\cdot e(k) \qquad \text{identical to C.6 and C.17, and} \tag{C.31}$$
$$T=t_{n+1}-t_n \qquad \text{identical to C.3 and C.14.} \tag{C.32}$$

The proof for equation (C.27) can be by applying the definitions of the respective functions:

$$\begin{aligned} e_{\text{combined}}(1) &= t_1+C(t_1) && \text{definition} \\ &= t_1-o-s\cdot t_1 && \text{using C.29} \\ &= -o+t_1-s\cdot t_1 && \\ &= e_{\text{offset}}(1)+e_{\text{skew}}(1) && \text{using C.5 and C.16.} \end{aligned} \tag{C.33}$$

Similarly, we can show that equation (C.28) is valid:

$$\begin{aligned} e_{\text{combined}}(k=n+1) &= t_{n+1} - C(t_n) - T\cdot s - T\cdot u(n) && \text{using C.30 and C.29} \\ &= t_n + T - C(t_n) - T\cdot s - T\cdot K_p\cdot e(n) && \text{using C.32 and C.31} \\ &= e_{\text{combined}}(n)\cdot(1-T\cdot K_p) + T\cdot(1-s) && \text{using C.30.} \end{aligned} \tag{C.34}$$

Thus, again as geometric series

$$\begin{aligned} e_{\text{combined}}(k) &= e_{\text{combined}}(1)\cdot(1-T\cdot K_p)^{n-1} + T\cdot(1-s)\cdot\frac{1-(1-T\cdot K_p)^{k-3}}{1-(1-T\cdot K_p)} \\ &= e_{\text{offset}}(1)\cdot(1-T\cdot K_p)^{n-1} \\ &\quad + e_{\text{skew}}(1)\cdot(1-T\cdot K_p)^{n-1} + (1-s)\cdot\frac{1-(1-T\cdot K_p)^{k-3}}{K_p)} \\ &= e_{\text{offset}}(k) + e_{\text{skew}}(k)\,. \end{aligned} \tag{C.35}$$

As equation (C.27) and equation (C.28) can be shown, equation (C.26) is valid, and thus the error terms can be analyzed separately. ■

C.4 Time Synchronization Verification Test Results

C.4.1 Internal Oscillator

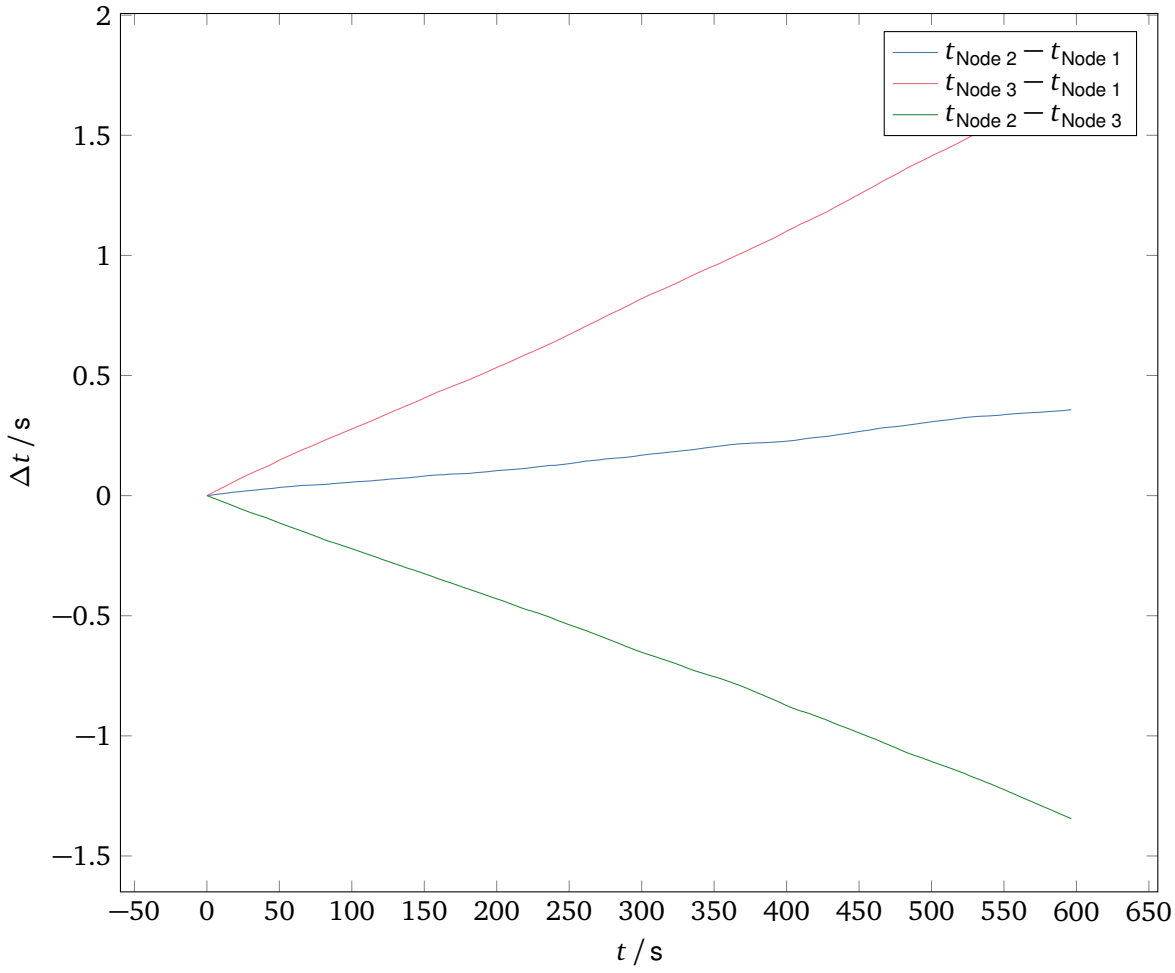


Figure C.1: Unsynchronized local time difference of nodes used for verification tests using internal oscillator. Static offset compensated and set to zero at $t = 0$.

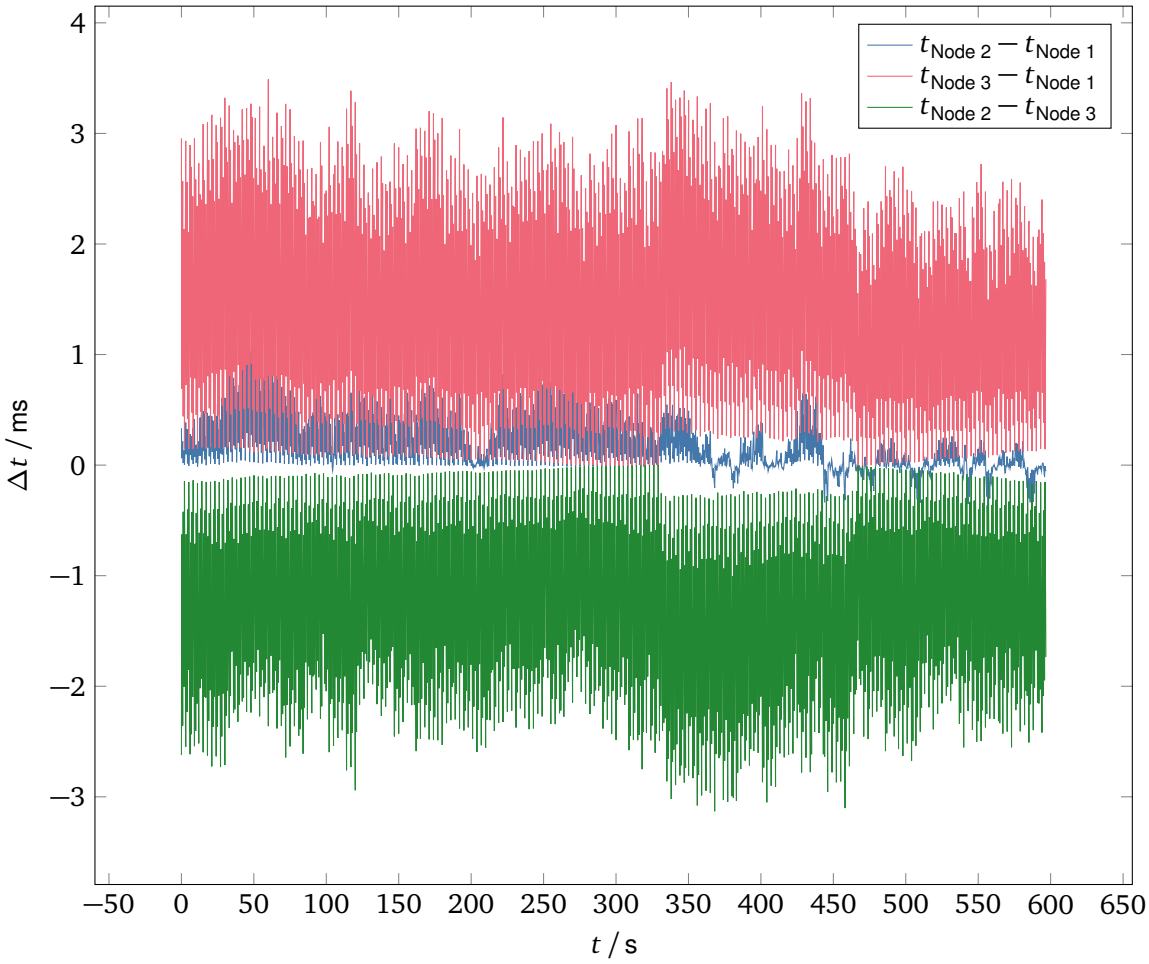


Figure C.2: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and no additional CAN load using internal oscillator.

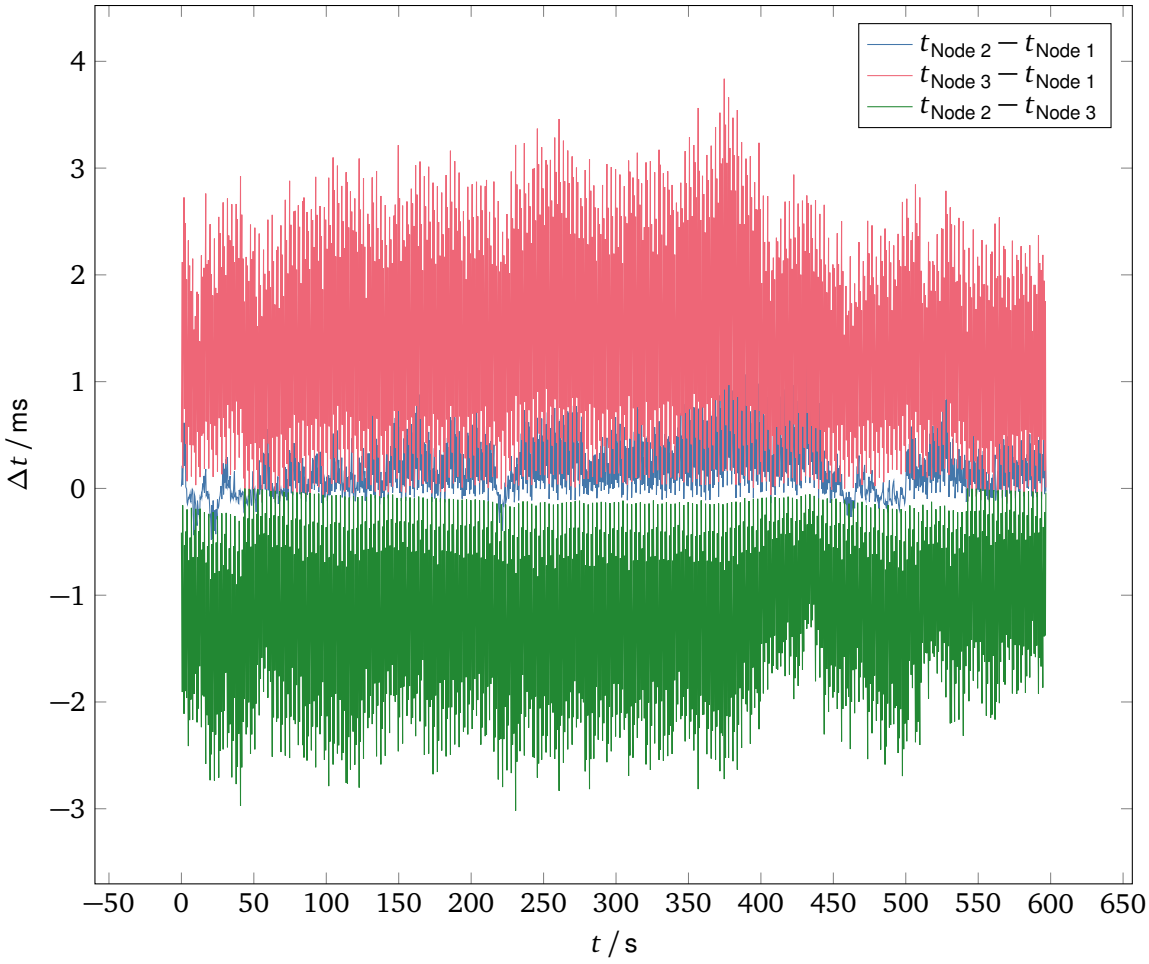


Figure C.3: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and client side generated CAN load using internal oscillator.

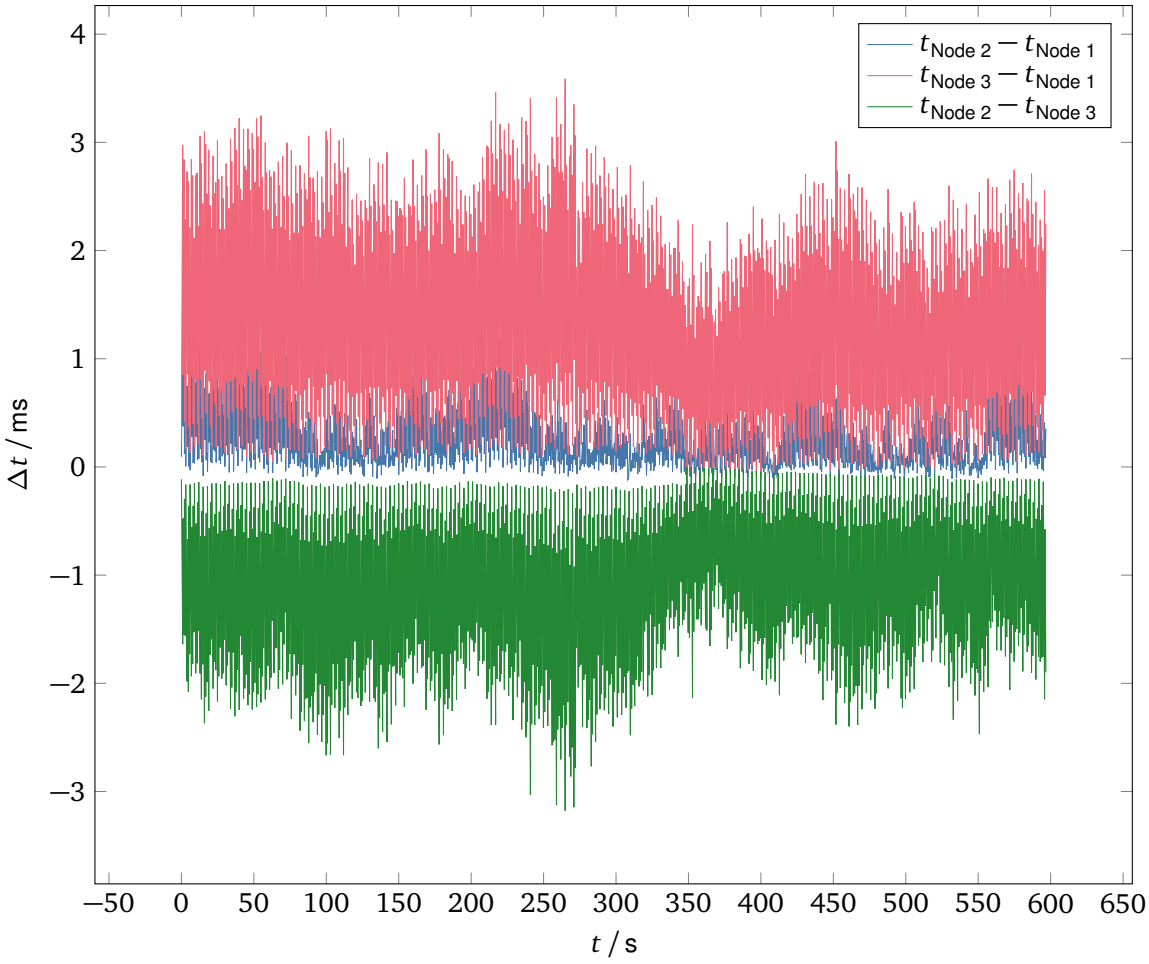


Figure C.4: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and server side generated CAN load using internal oscillator.

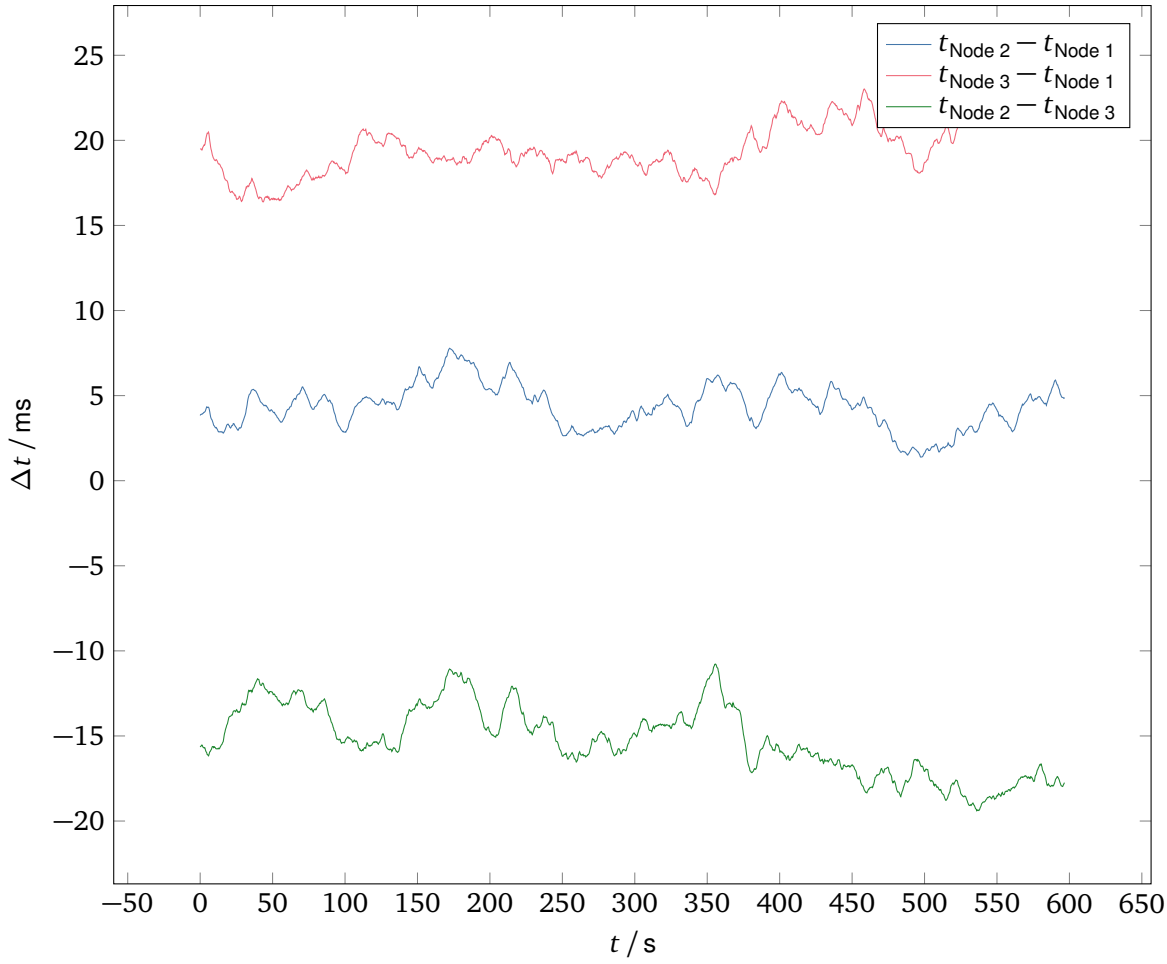


Figure C.5: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and no additional CAN load using internal oscillator.

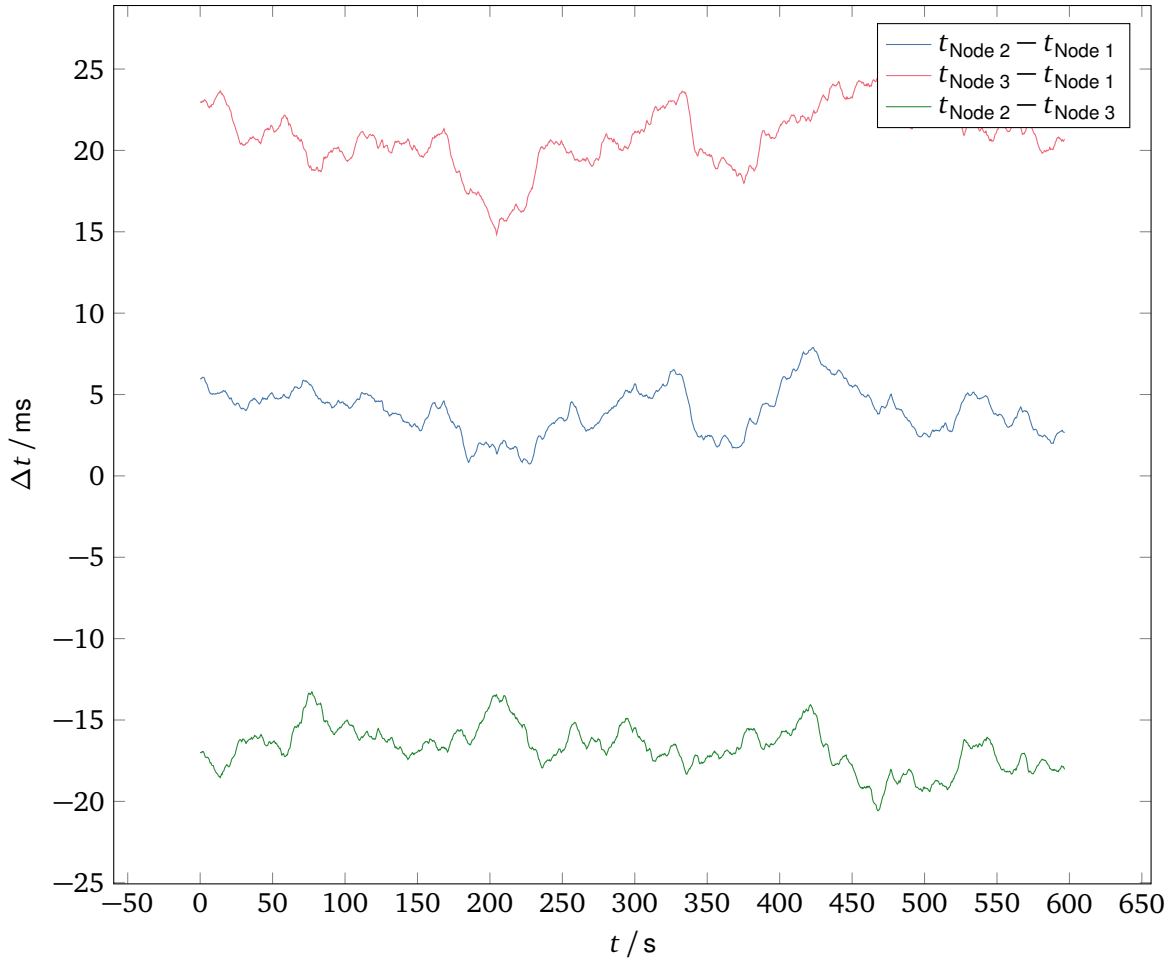


Figure C.6: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and client side generated CAN load using internal oscillator.

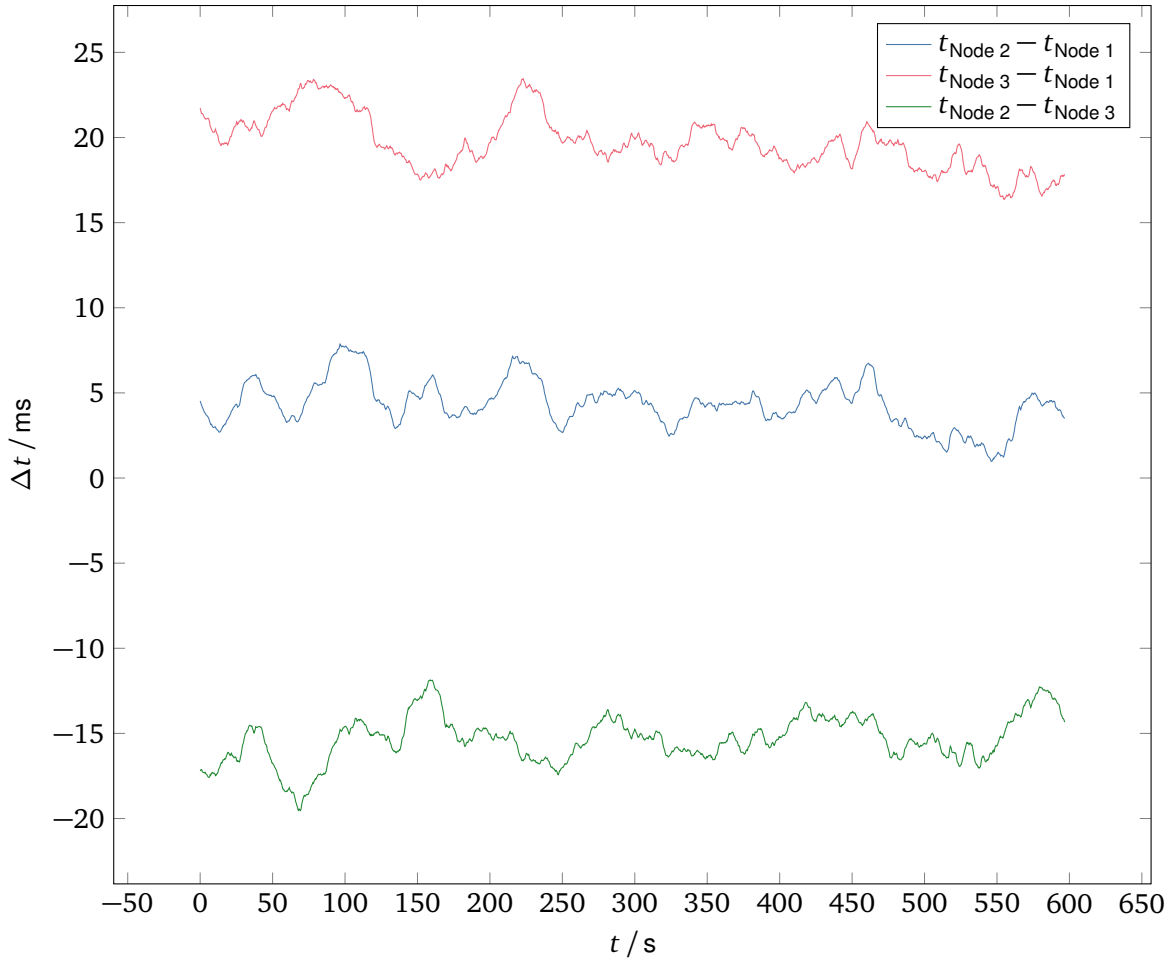


Figure C.7: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and server side generated CAN load using internal oscillator.

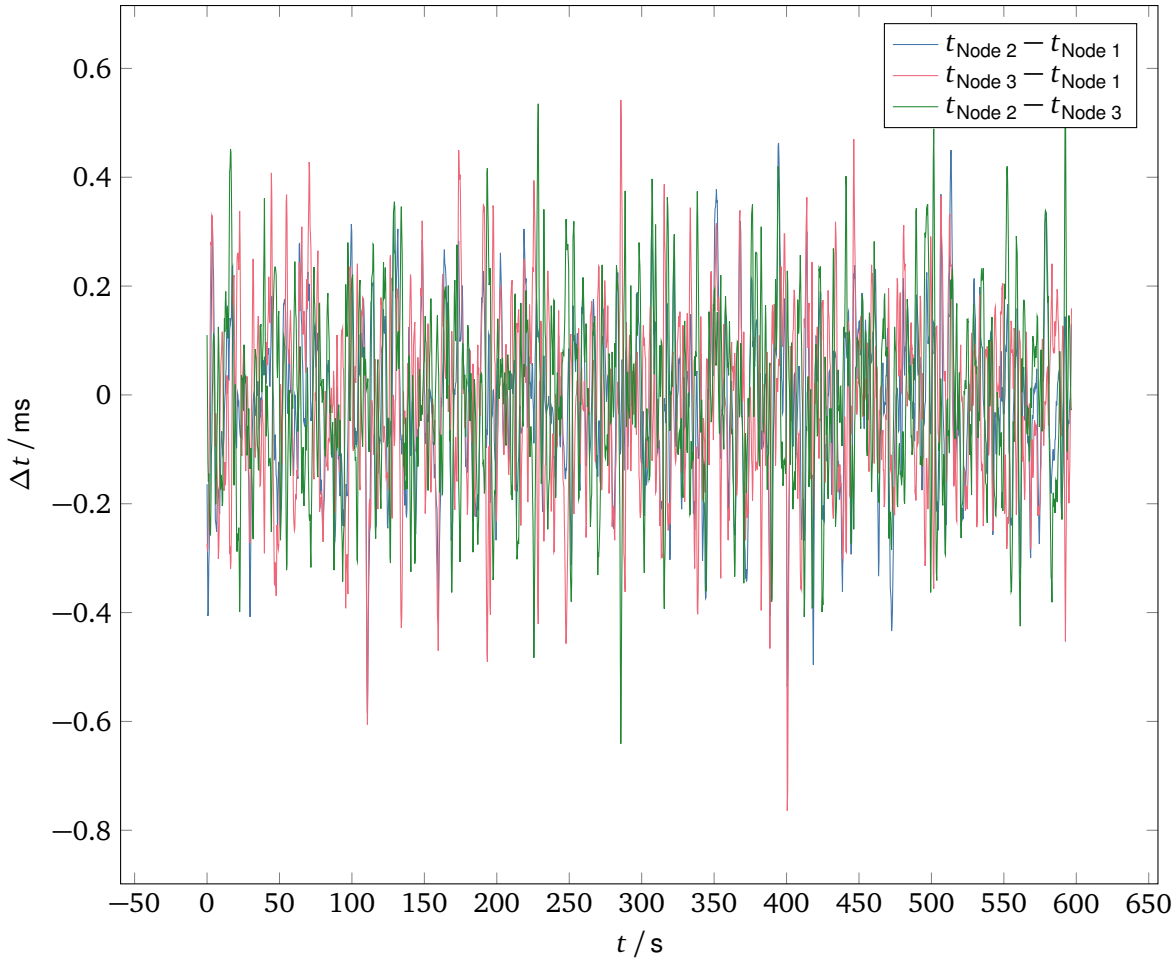


Figure C.8: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and no additional CAN load using internal oscillator.

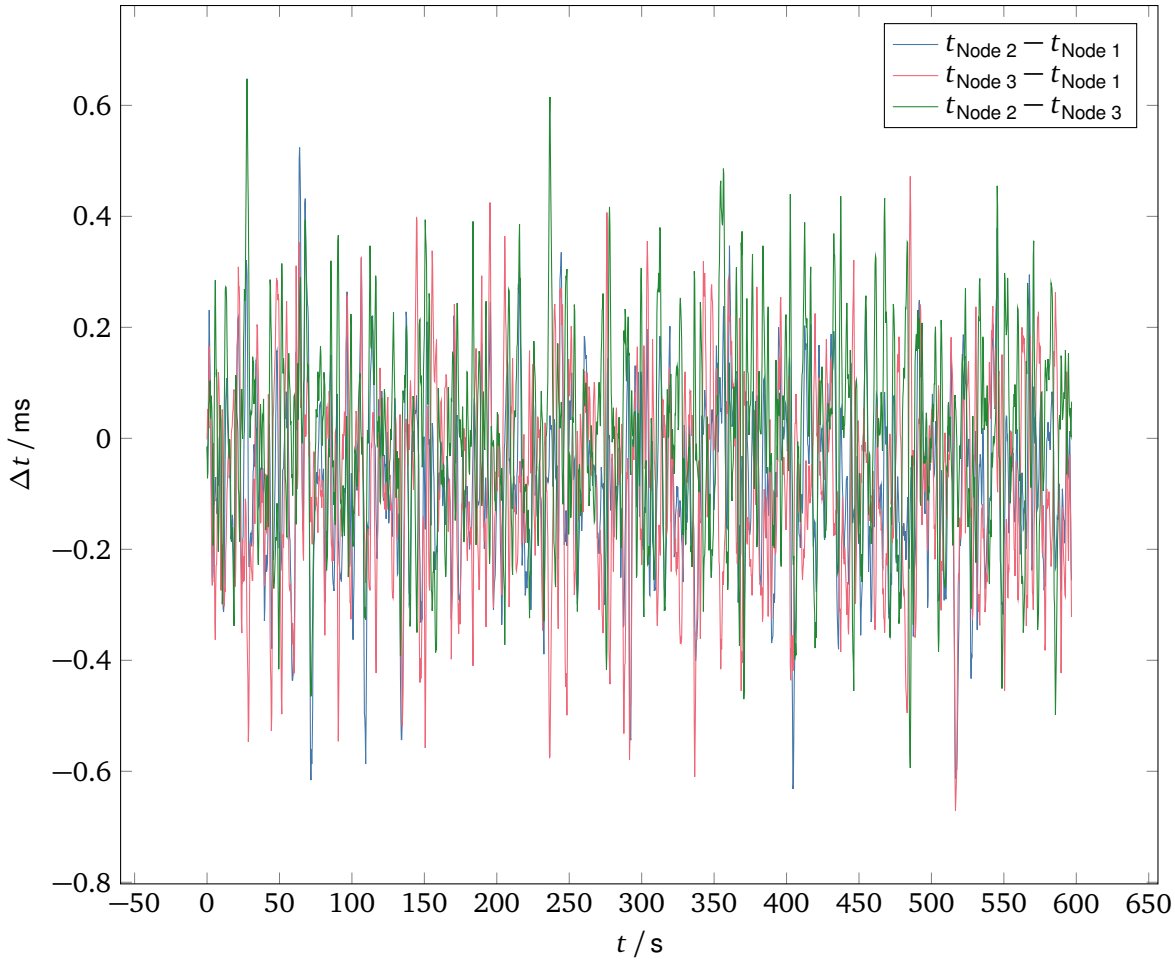


Figure C.9: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and client side generated CAN load using internal oscillator.

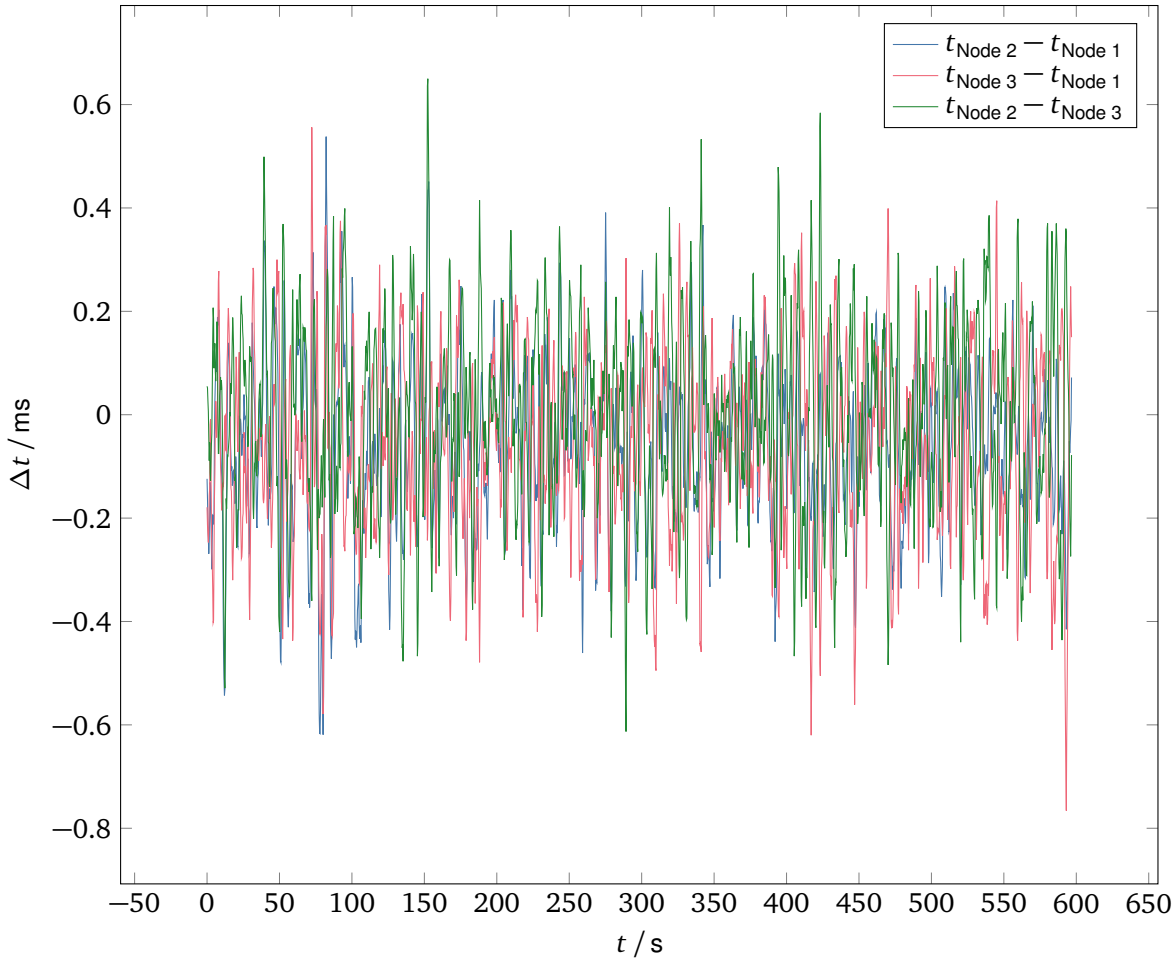


Figure C.10: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and server side generated CAN load using internal oscillator.

C.4.2 External 32.768 kHz Oscillator and Reference Node 1

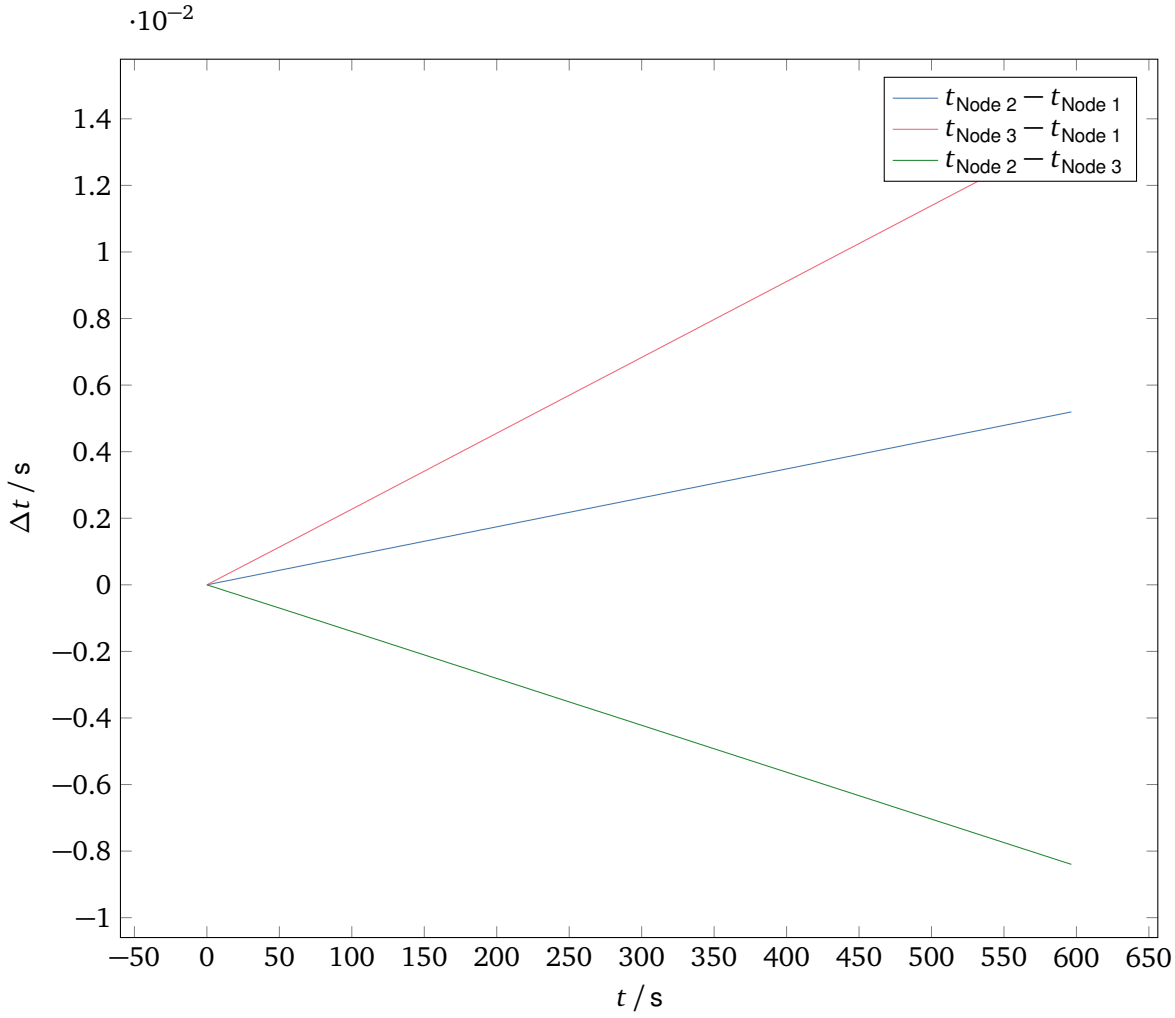


Figure C.11: Unsynchronized local time difference of nodes used for verification tests using external 32.768 kHz oscillator. Static offset compensated and set to zero at $t = 0$.

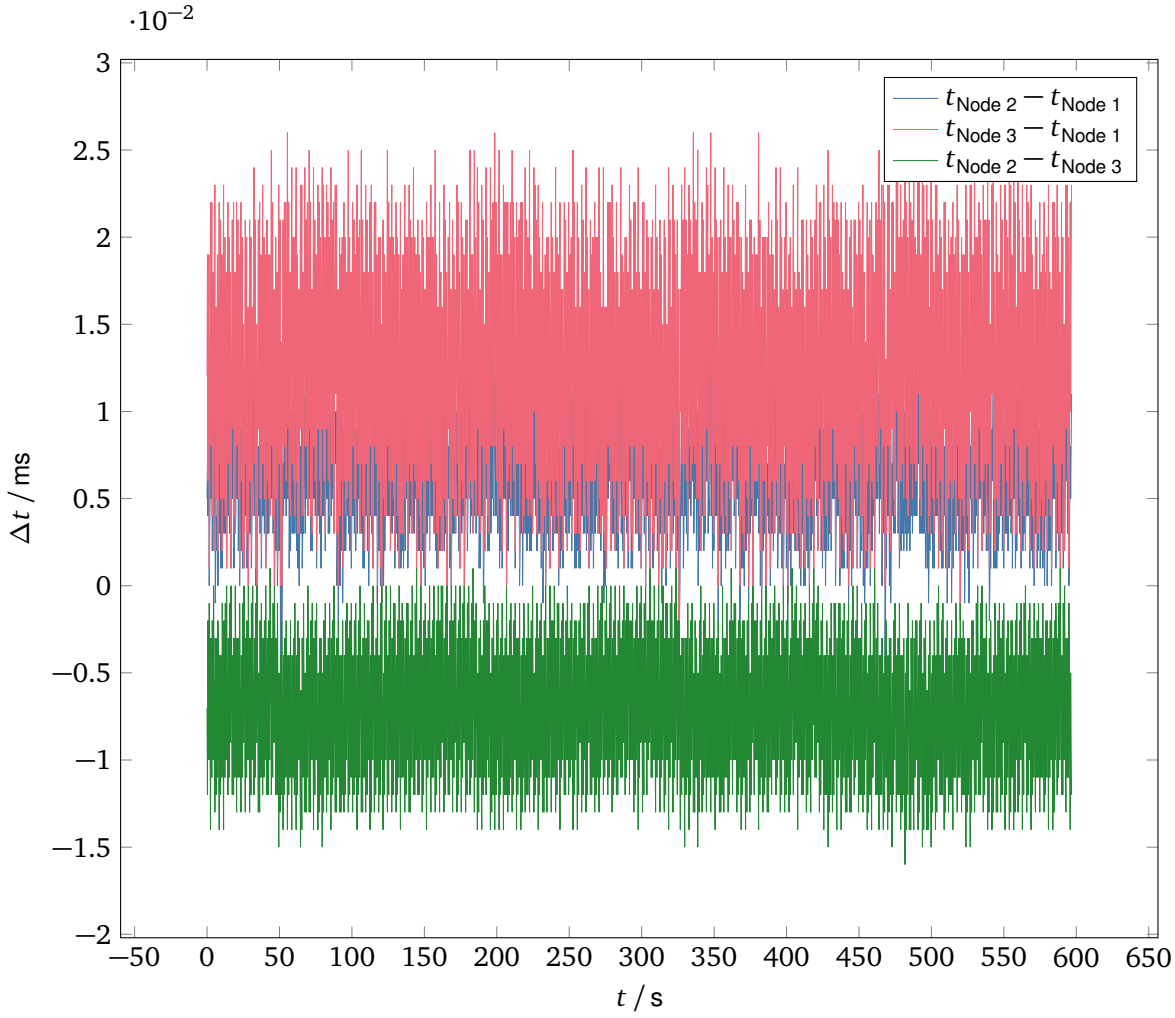


Figure C.12: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

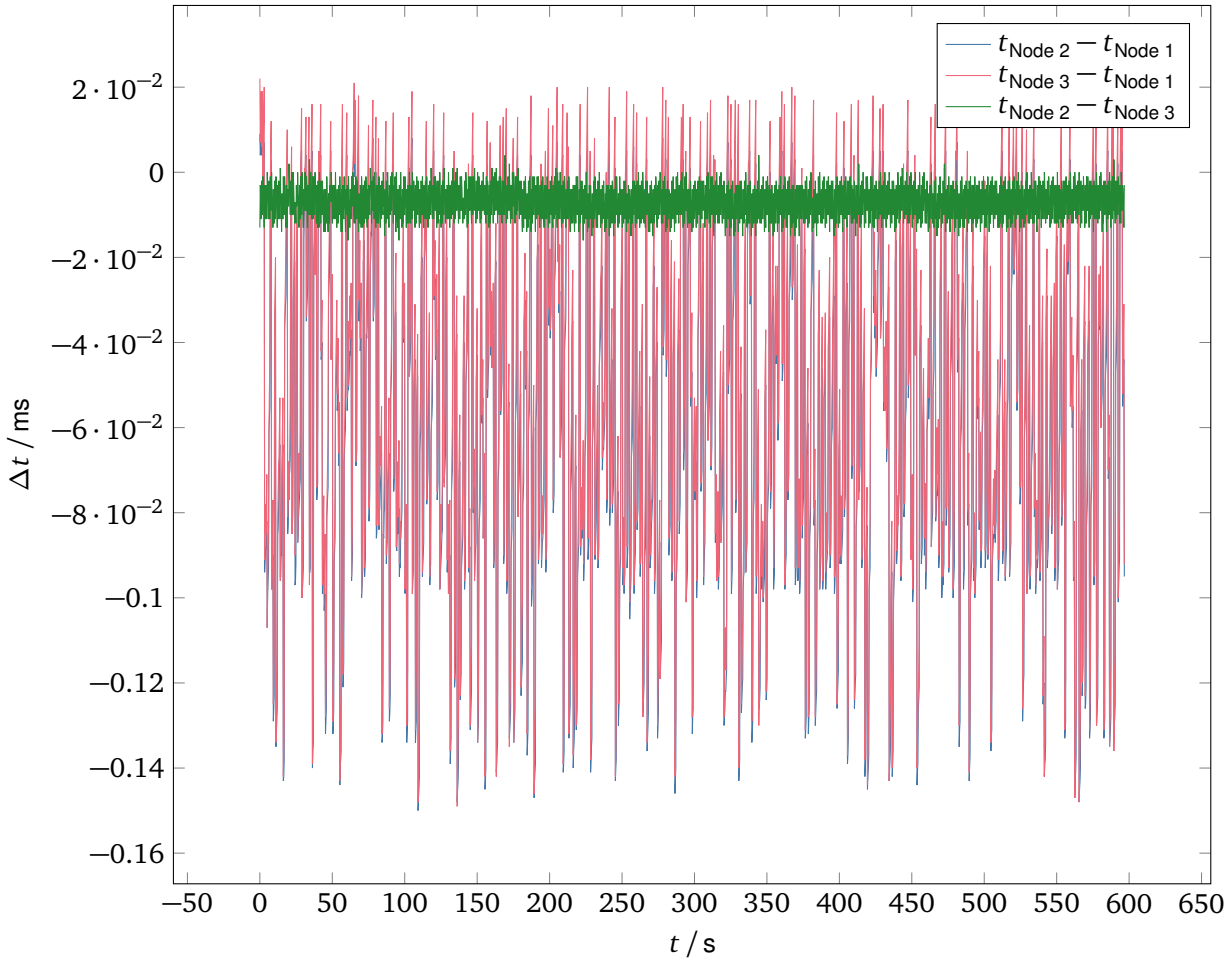


Figure C.13: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

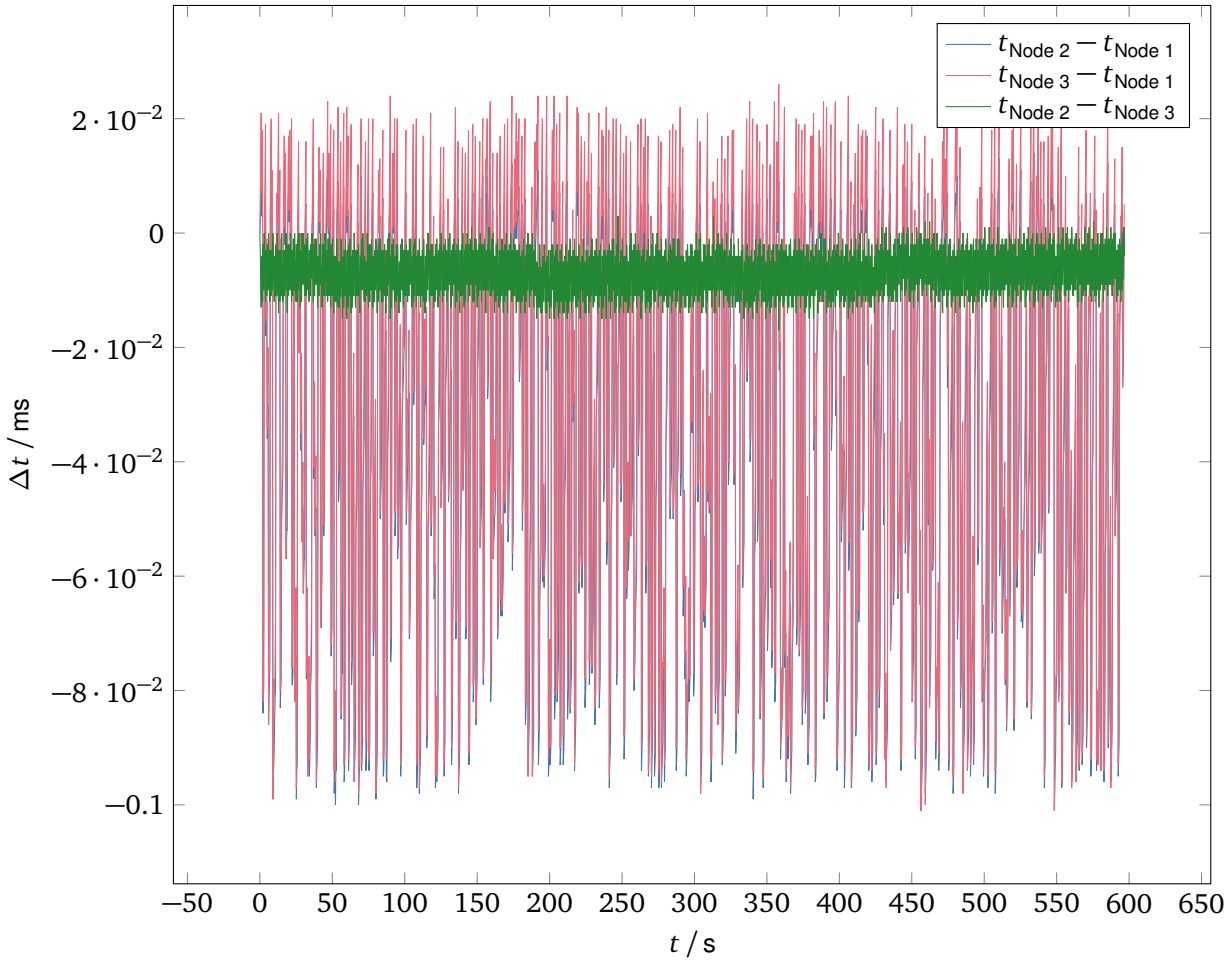


Figure C.14: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

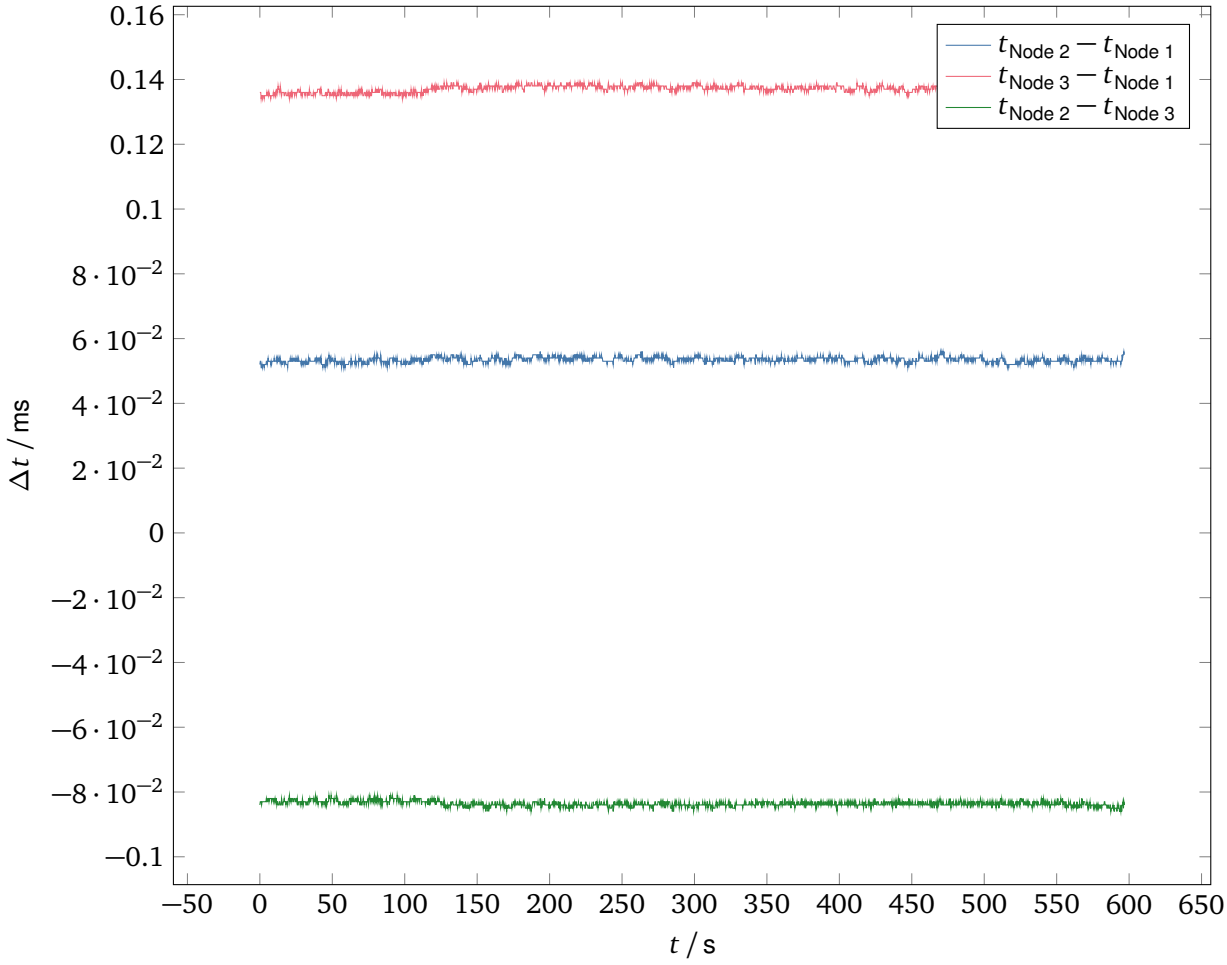


Figure C.15: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

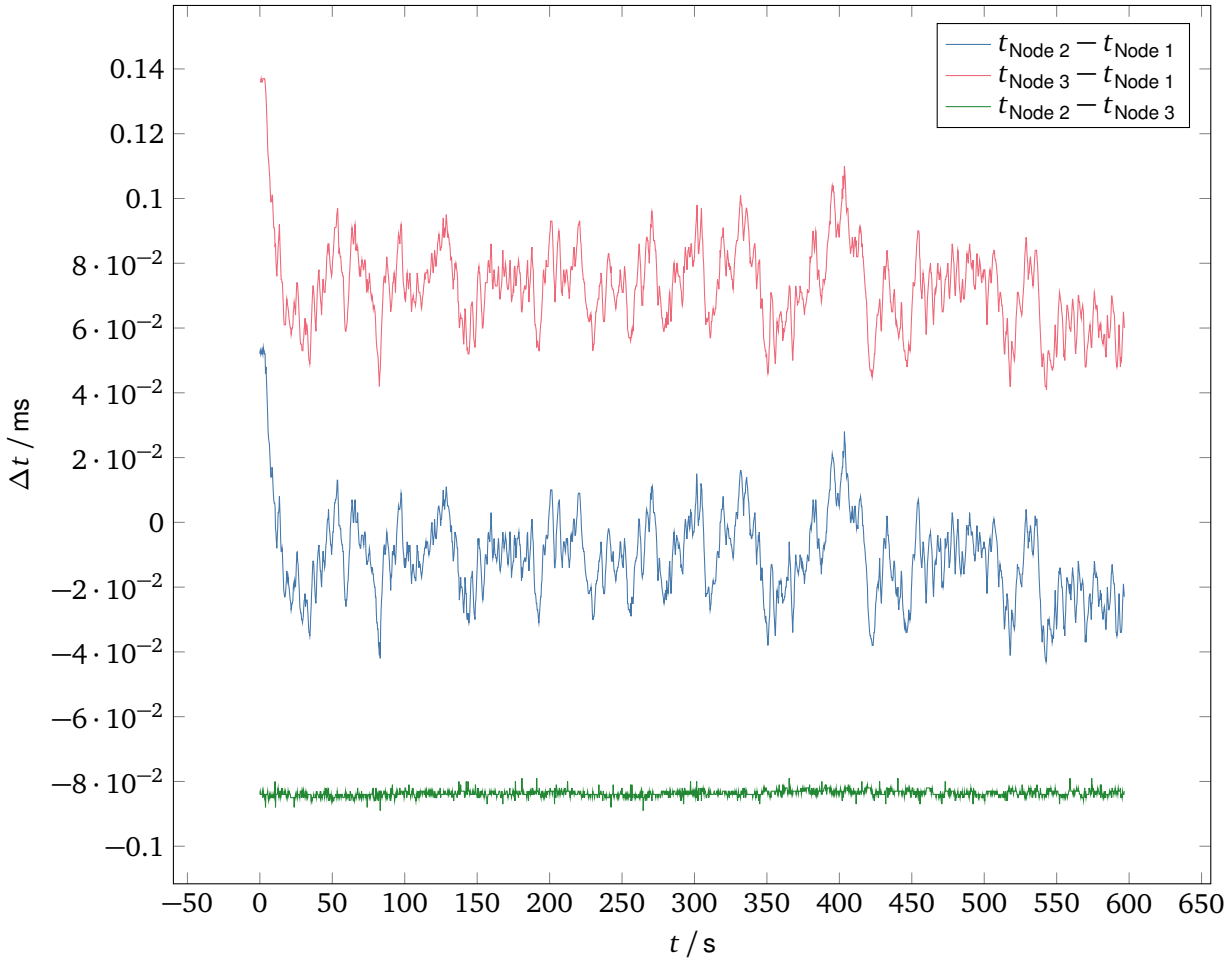


Figure C.16: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

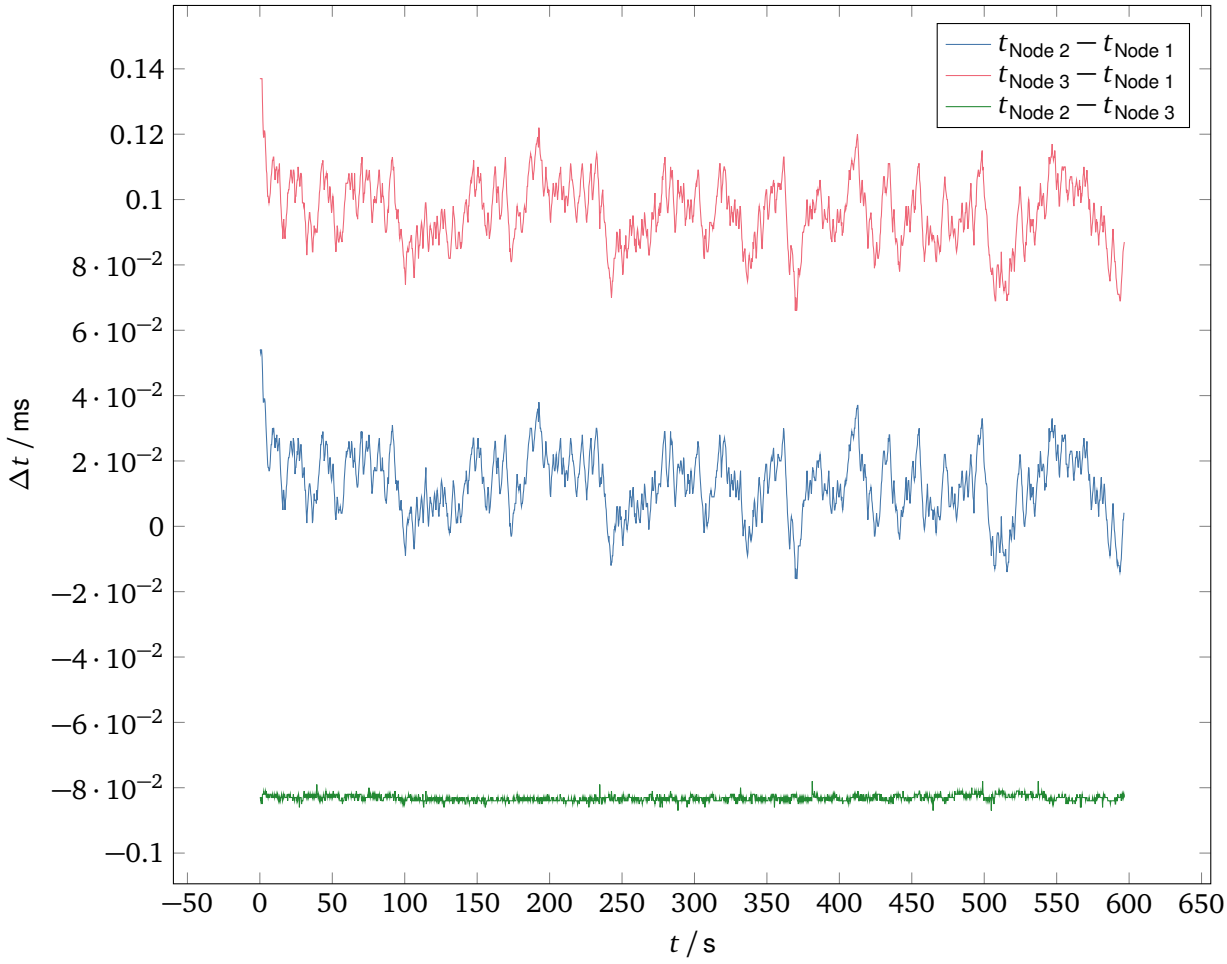


Figure C.17: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

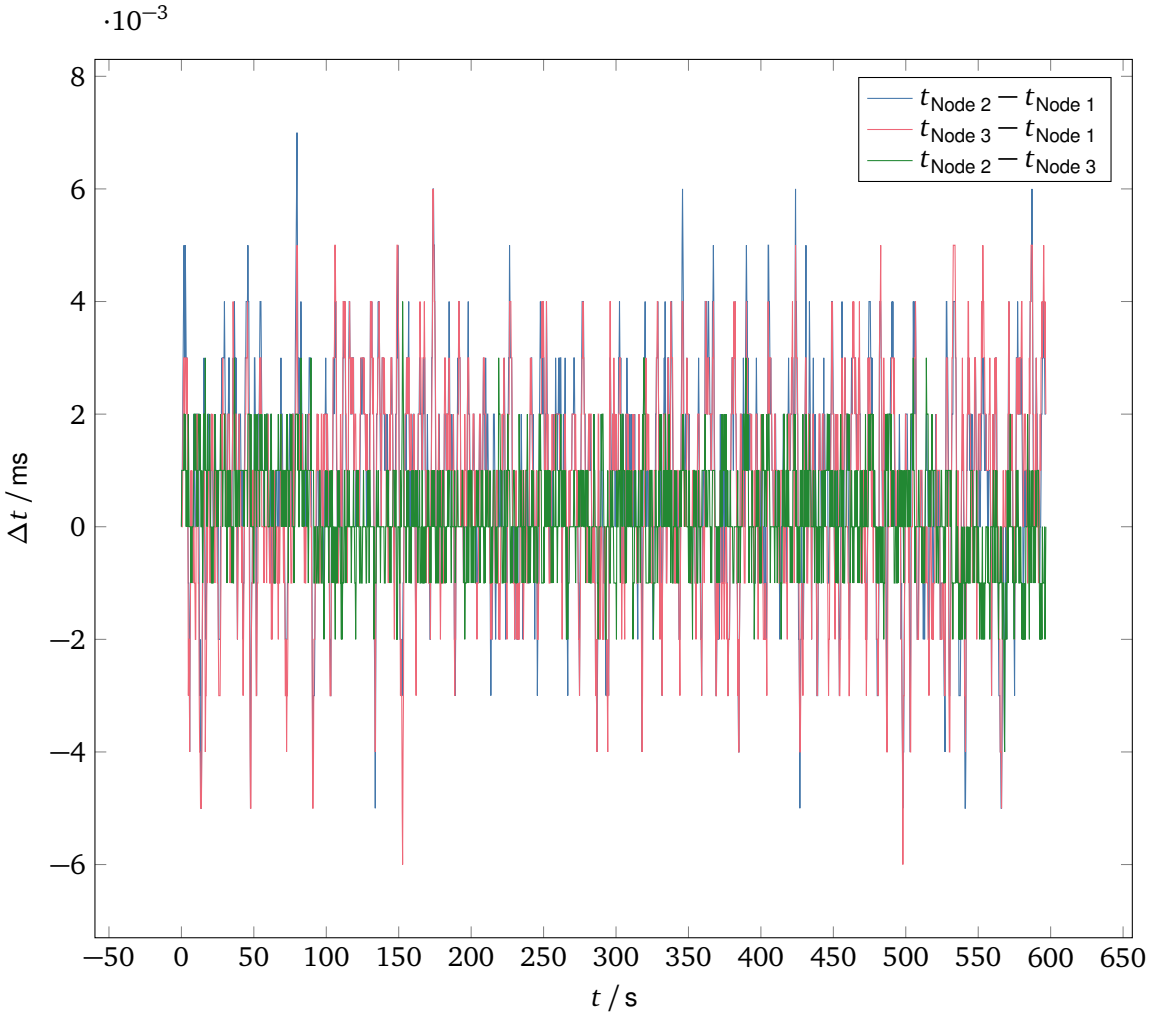


Figure C.18: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

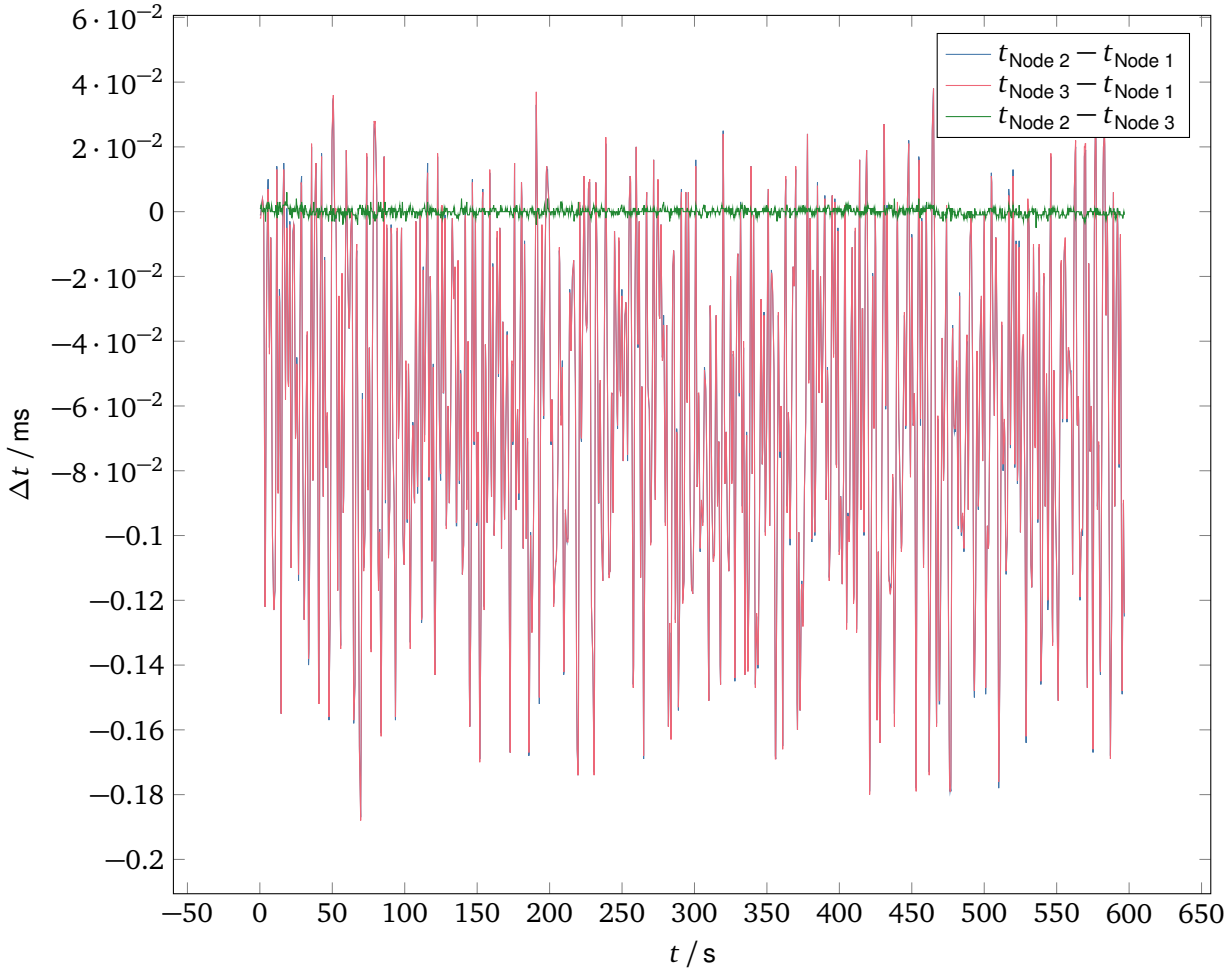


Figure C.19: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

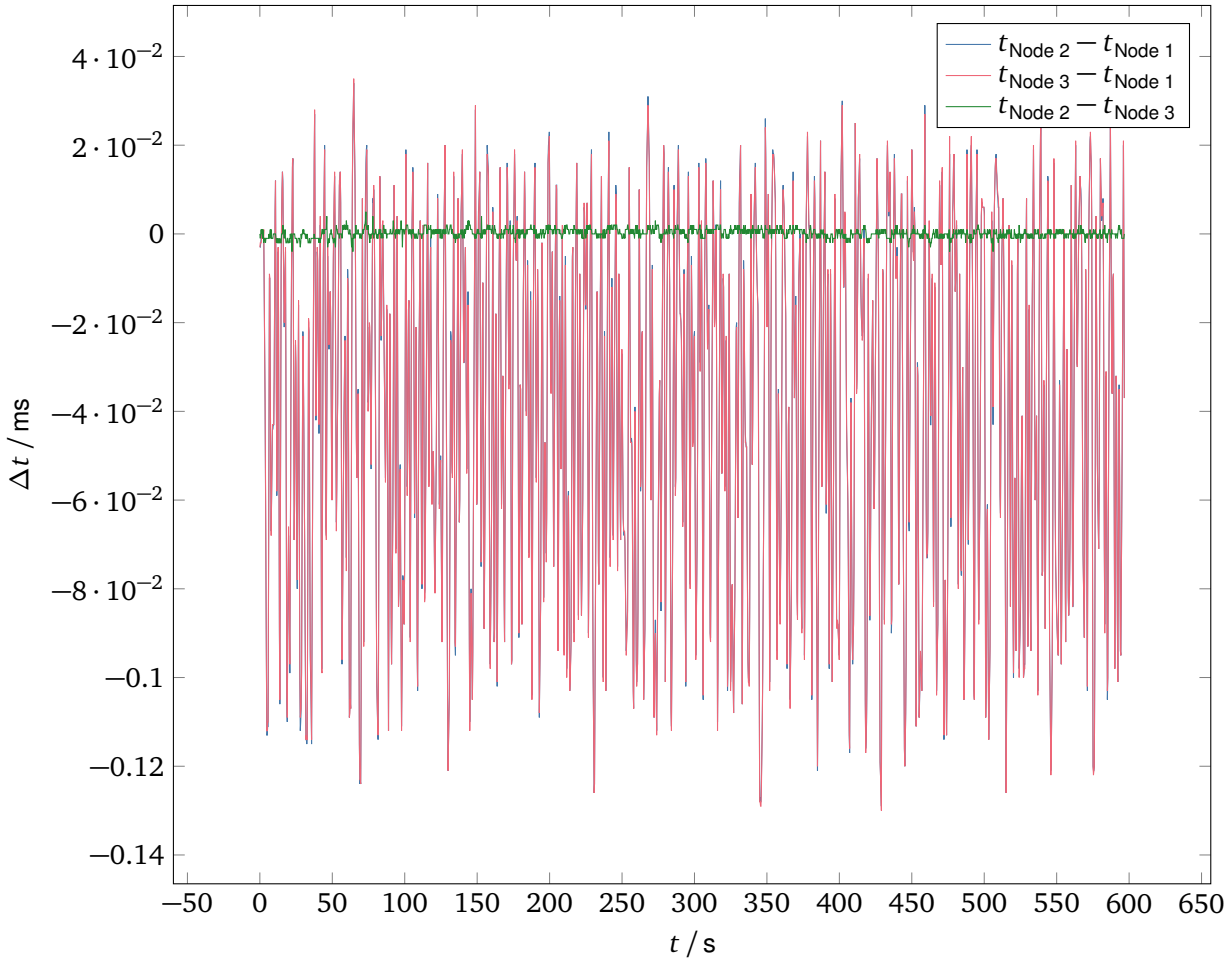


Figure C.20: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 1 is the time synchronization reference node.

C.4.3 External 32.768 kHz Oscillator and Reference Node 3

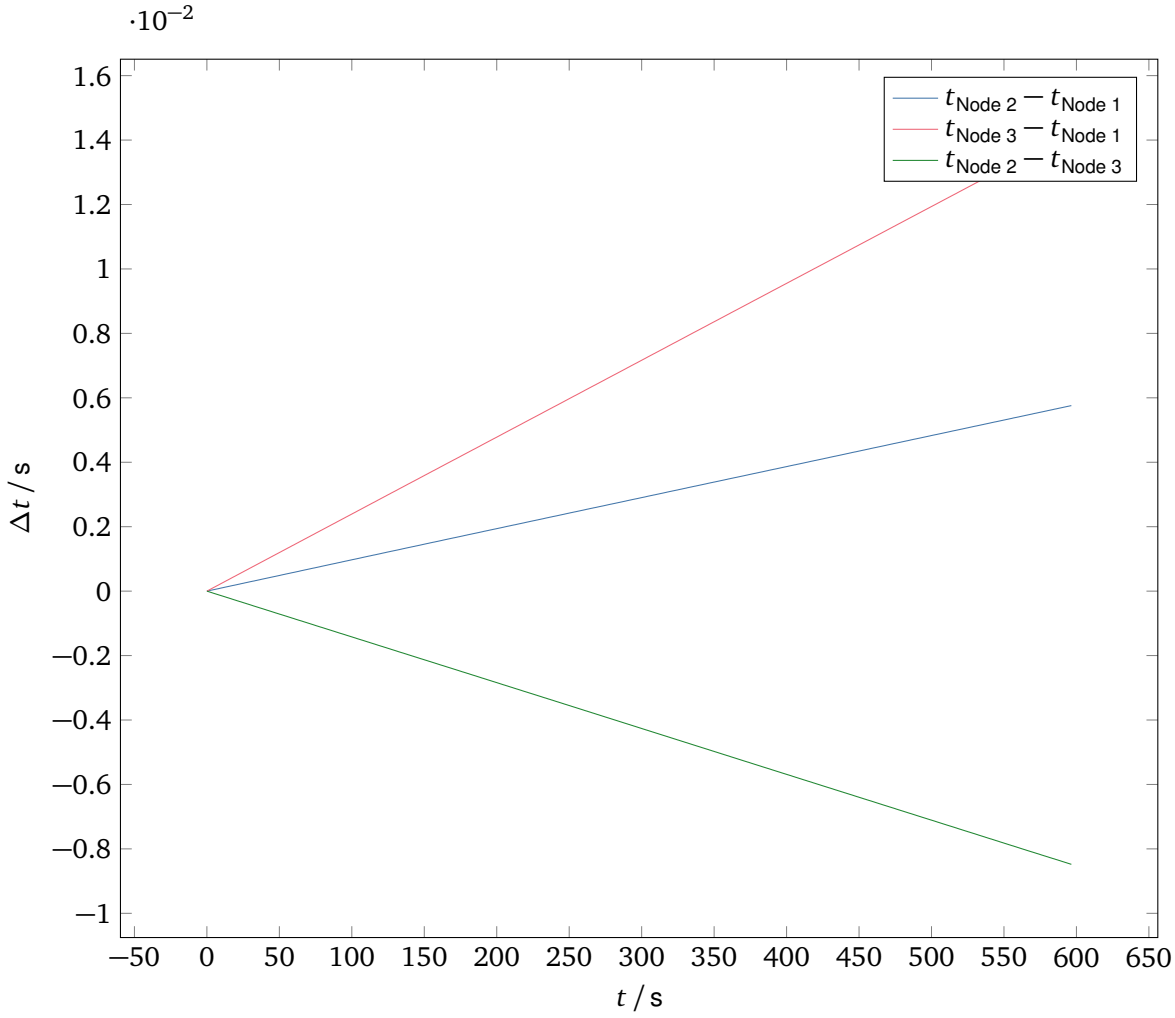


Figure C.21: Unsynchronized local time difference of nodes used for verification tests using external 32.768 kHz oscillator. Static offset compensated and set to zero at $t = 0$. Node 3 is the time synchronization reference node.

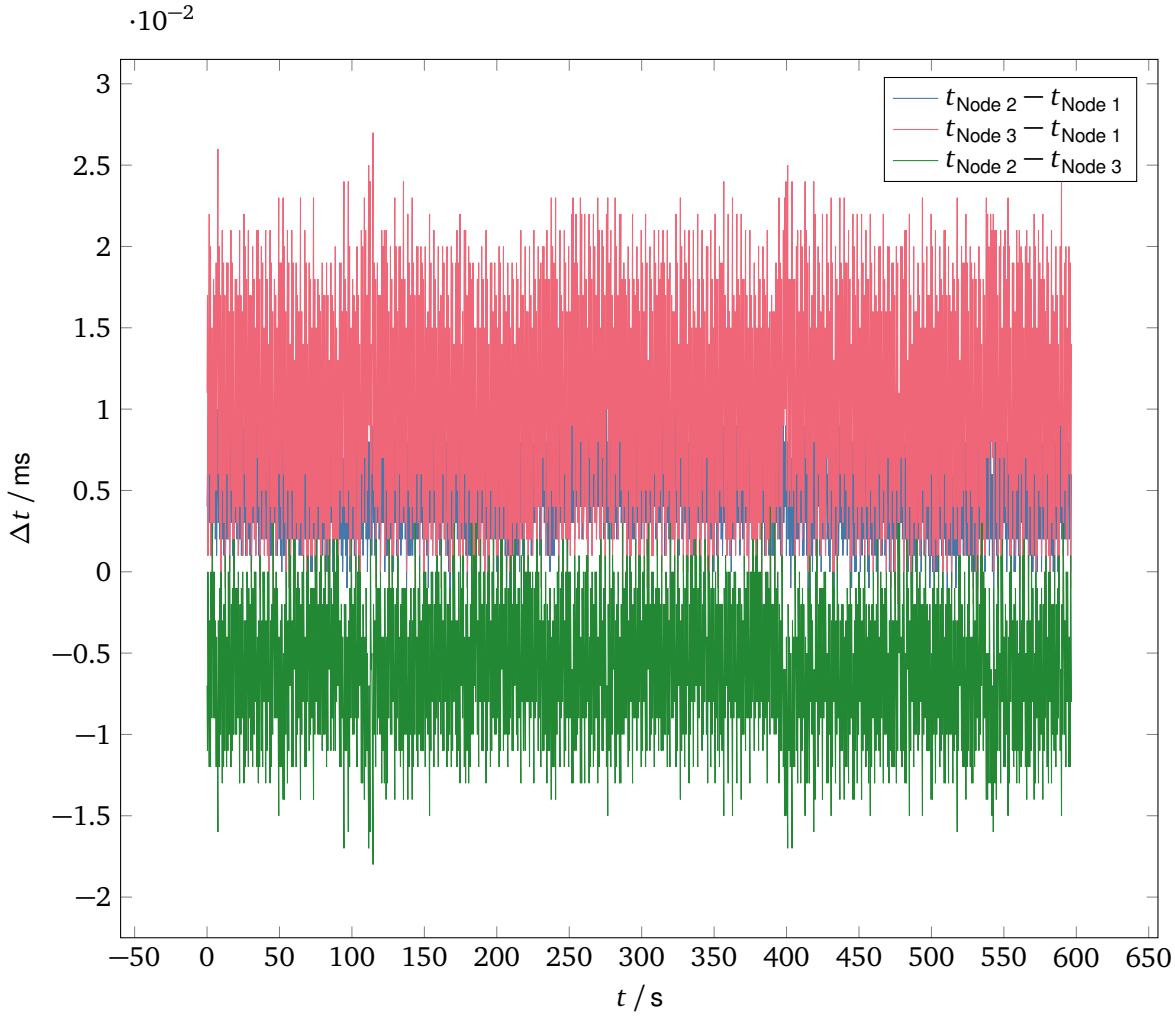


Figure C.22: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

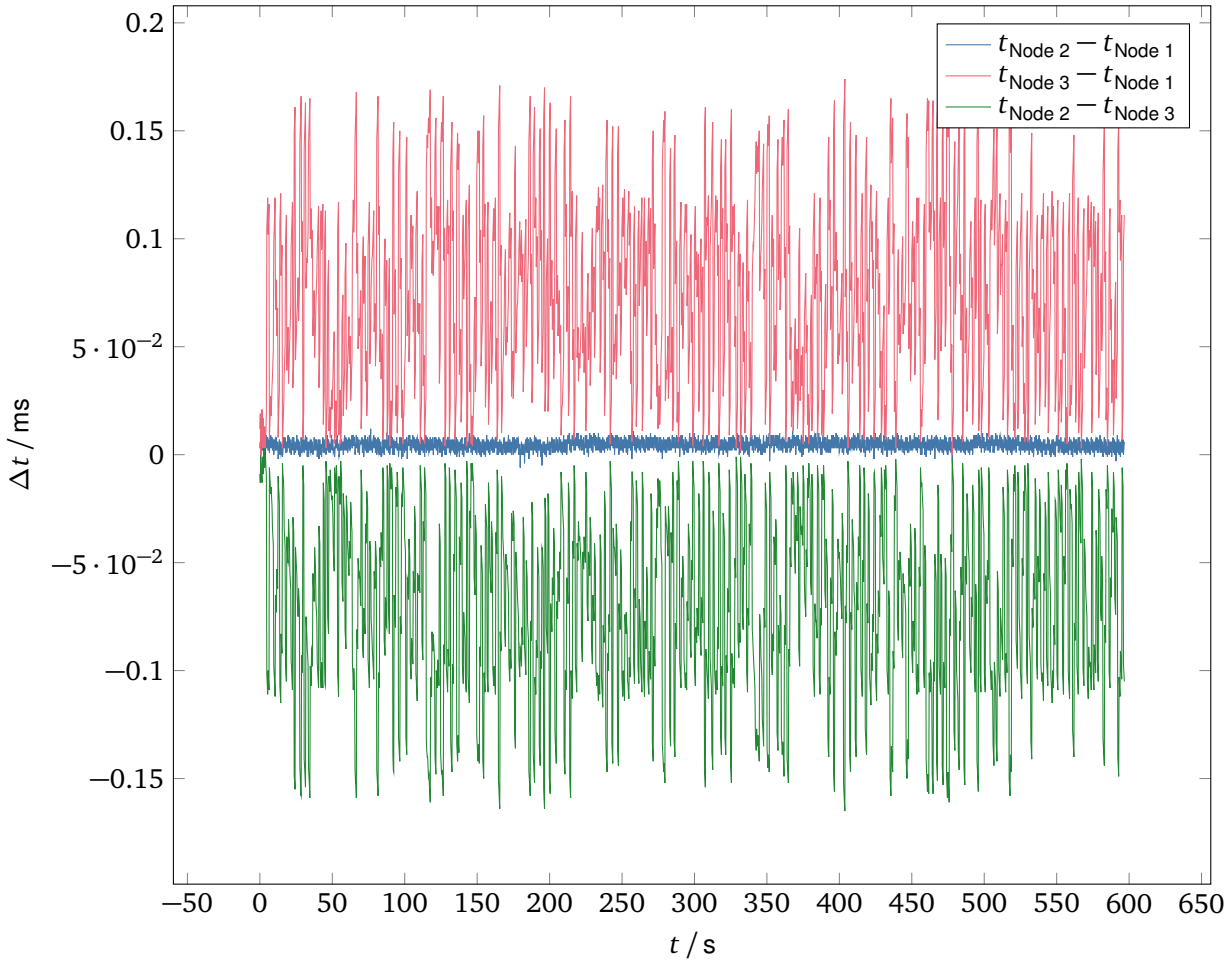


Figure C.23: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

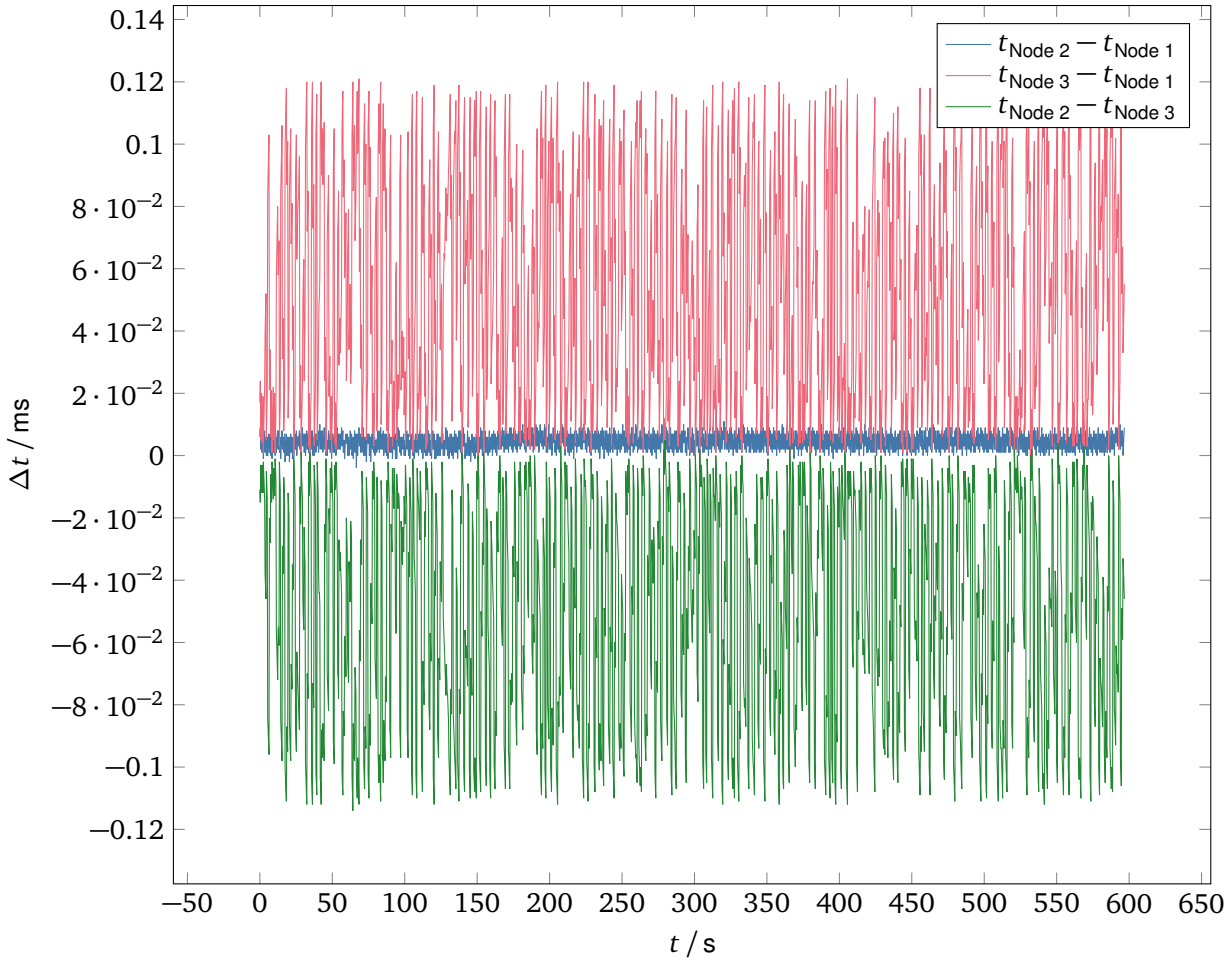


Figure C.24: Pairwise comparison of local time differences of nodes during verification test with direct-set clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

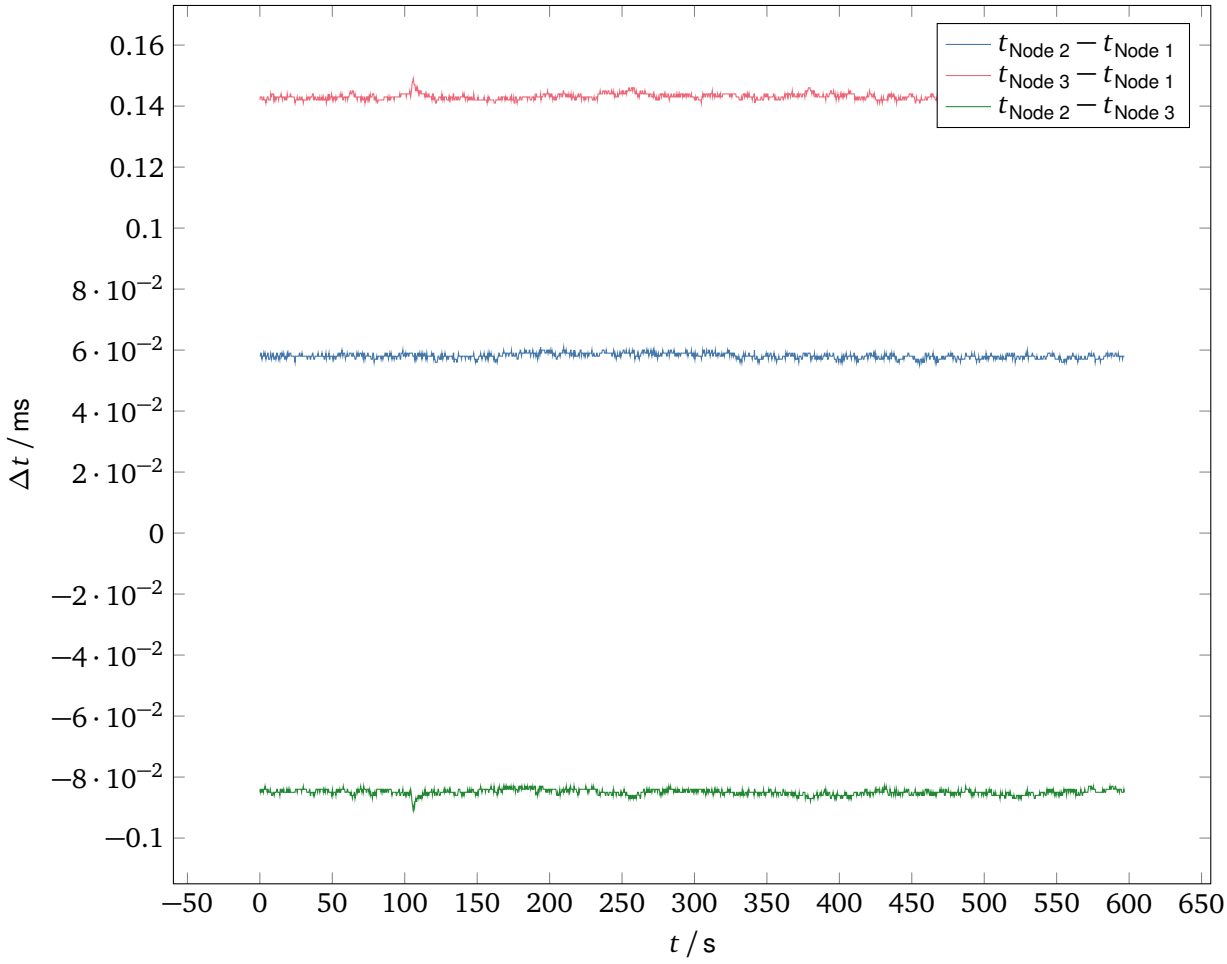


Figure C.25: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

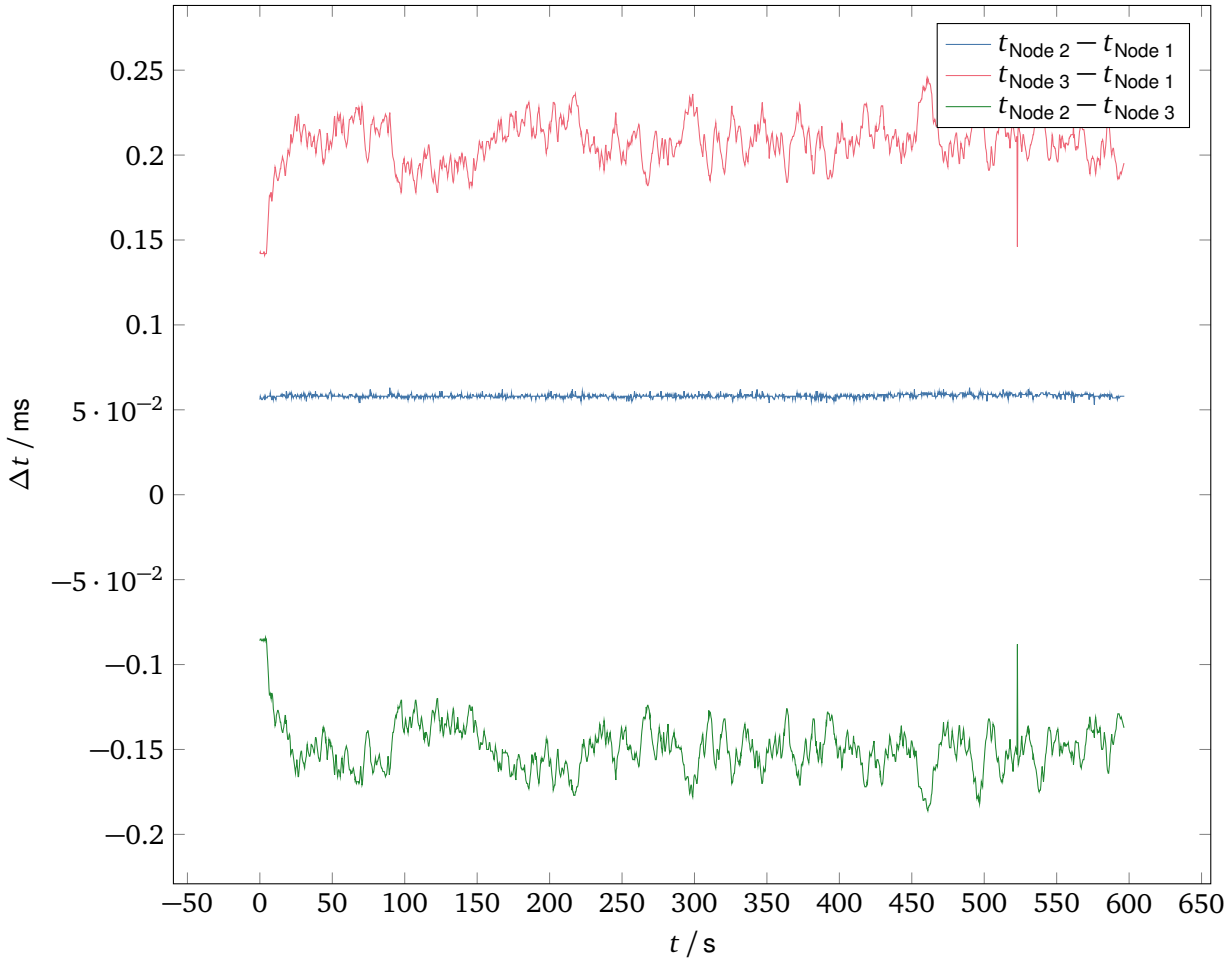


Figure C.26: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

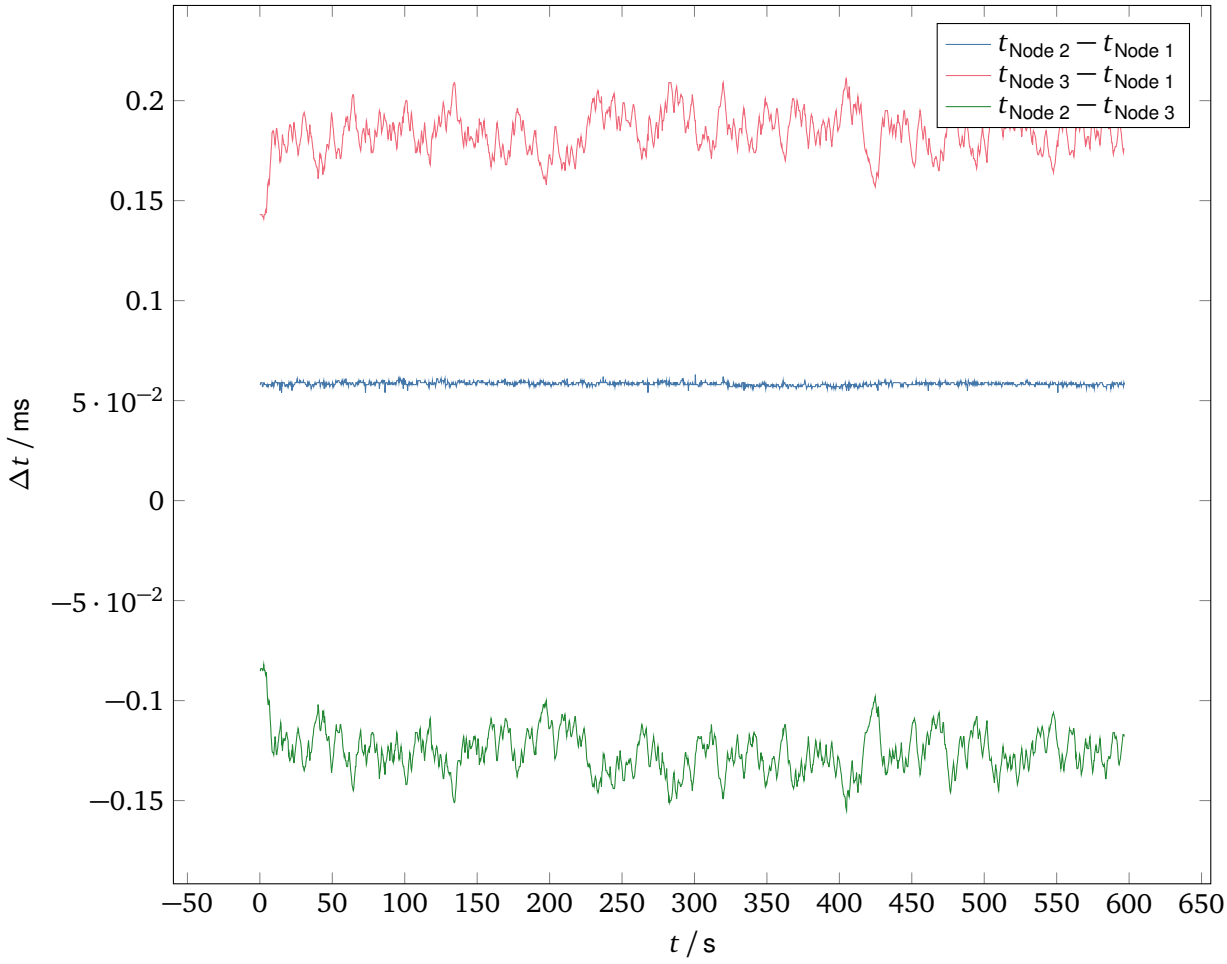


Figure C.27: Pairwise comparison of local time differences of nodes during verification test with P-controlled clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

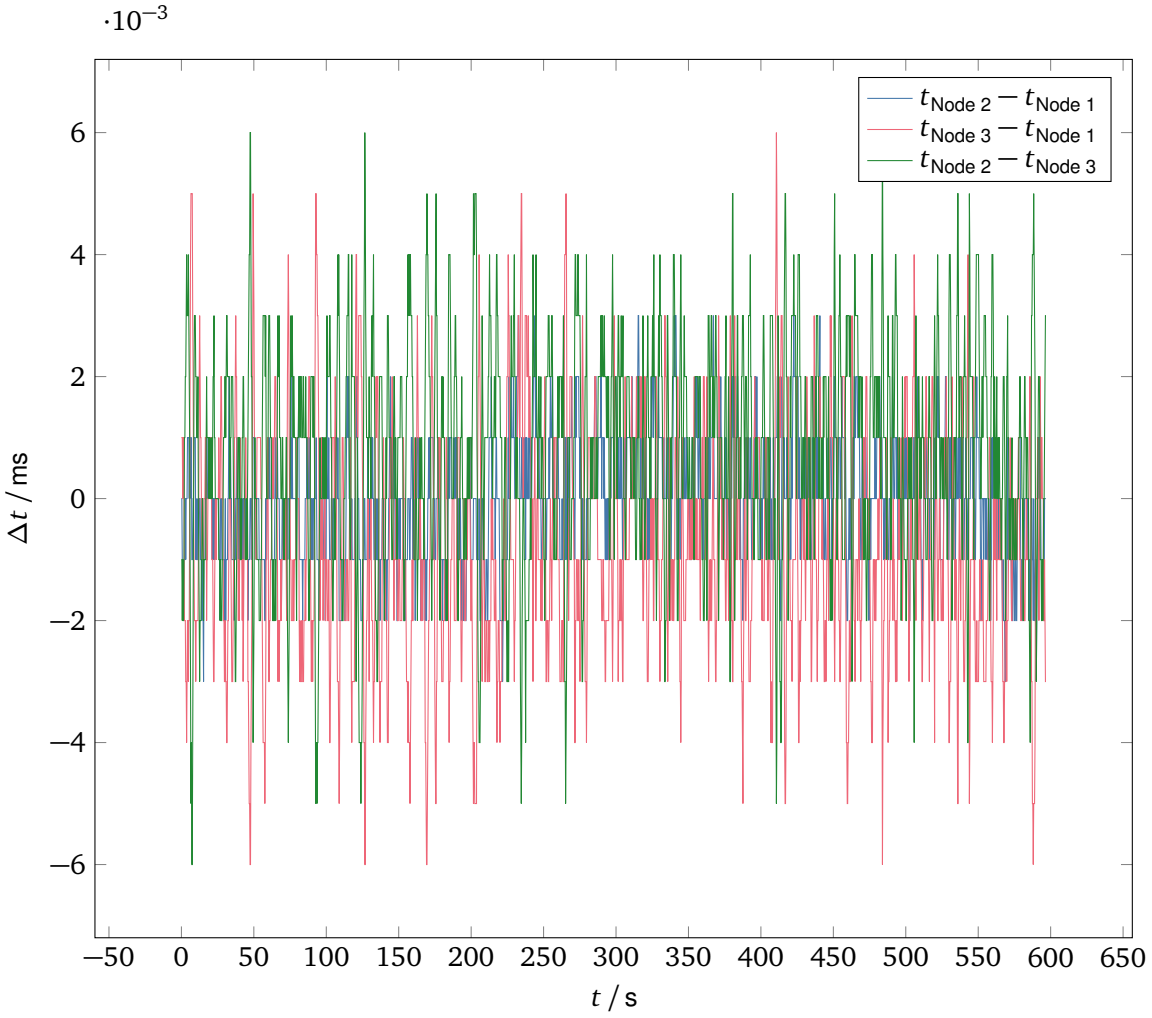


Figure C.28: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and no additional CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

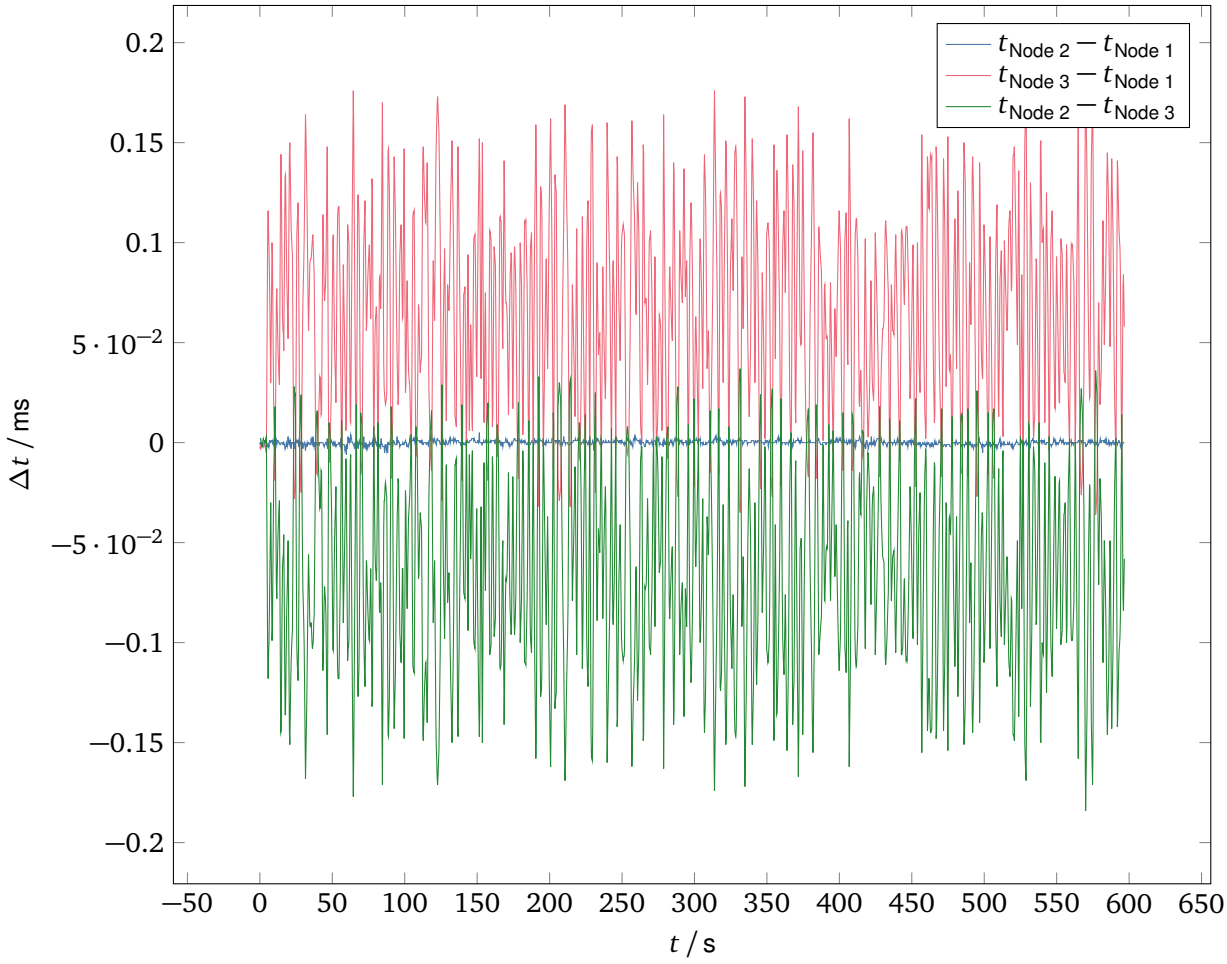


Figure C.29: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and client side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

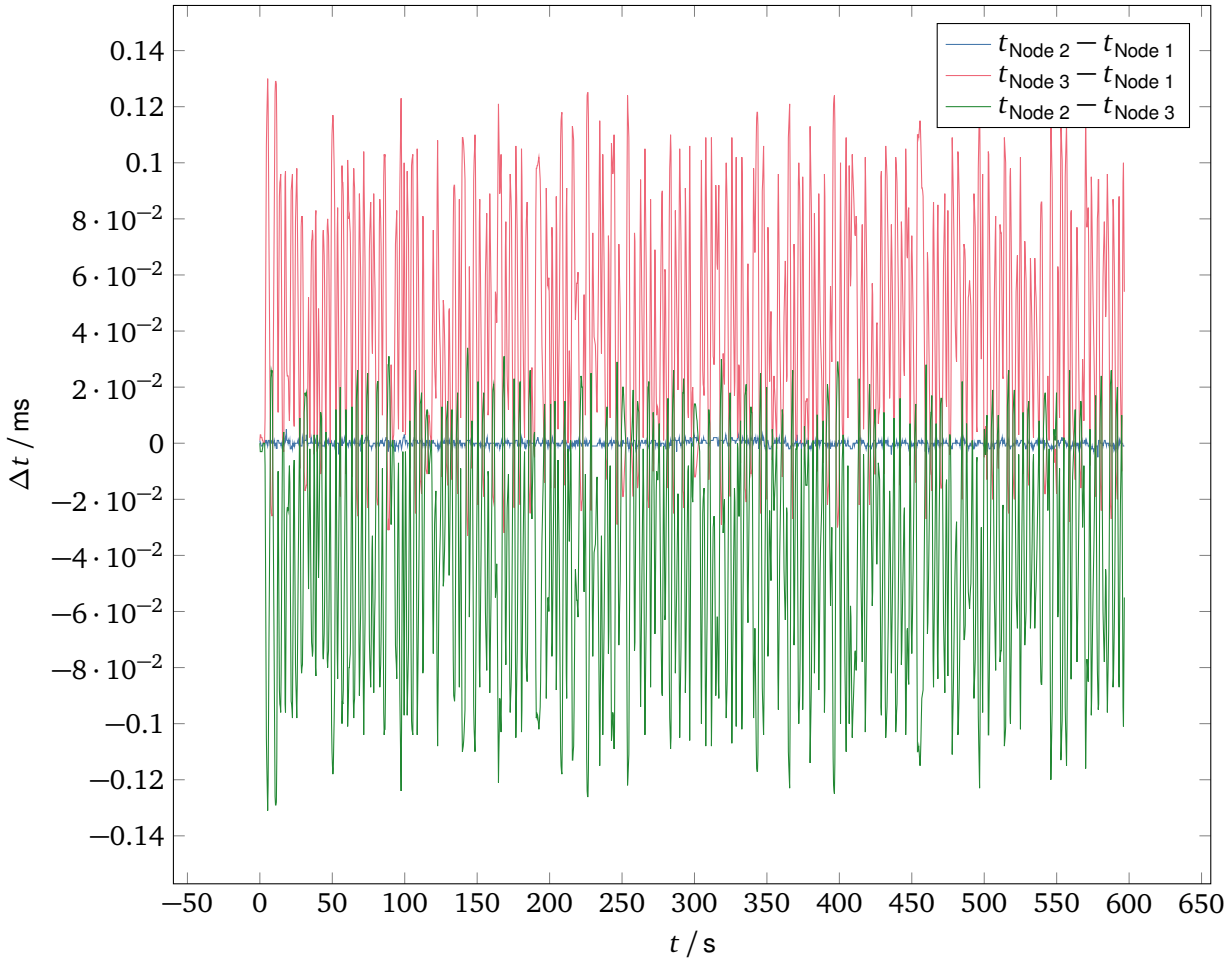


Figure C.30: Pairwise comparison of local time differences of nodes during verification test with PI-controlled clock updated and server side generated CAN load using external 32.768 kHz oscillator. Node 3 is the time synchronization reference node.

Appendix D

Implementation

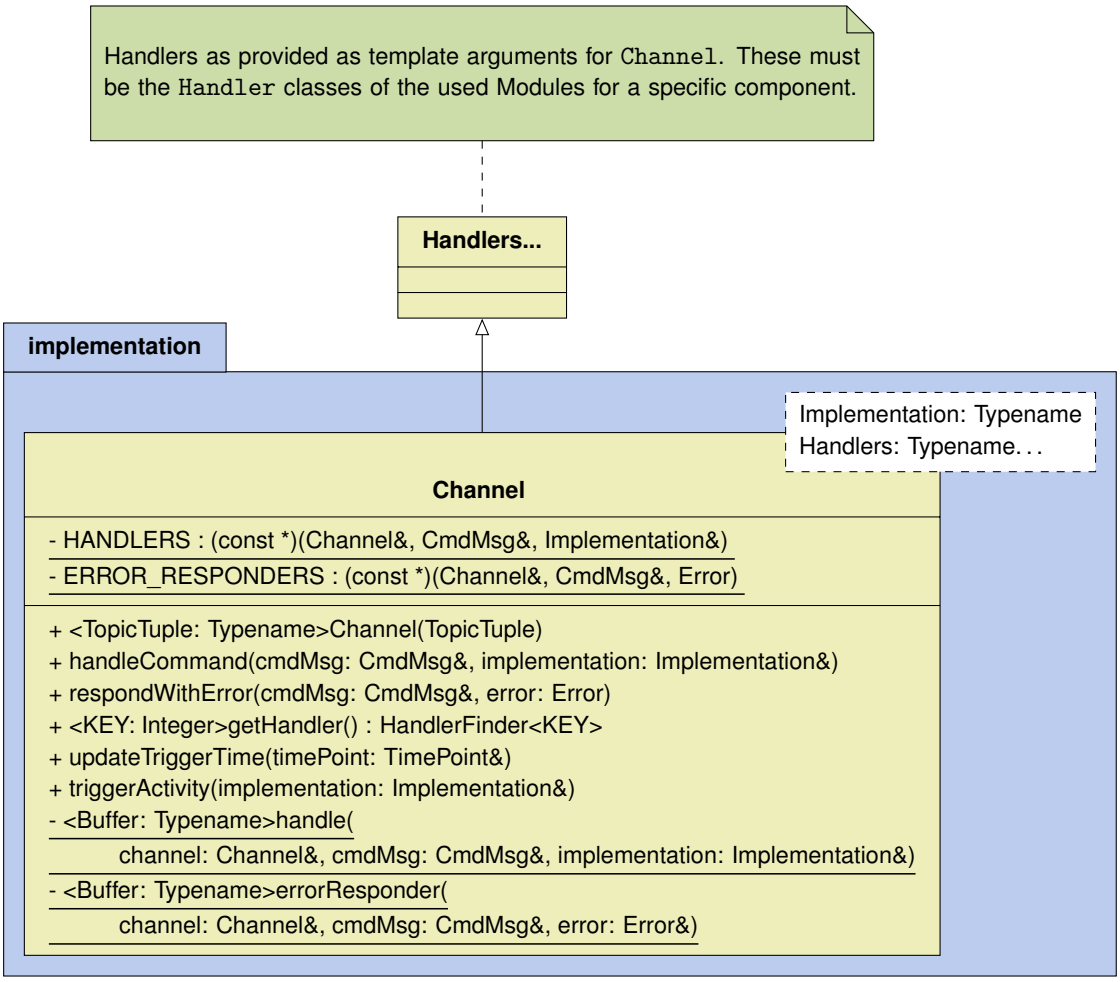


Figure D.1: Class diagram of the DOSIS `Channel`. `Channel` expects the `Implementation` template argument to be a *ComponentImplementation* class. `Channel` inherits from its variadic template parameter `Handlers`.

D.1 Usage of DOSIS

This section presents the usage of the DOSIS framework for a simple light emitting diode (LED) toggling example. Therefore, it presents the implementation of a simple DOSIS *Driver* controlling a GPIO pin and a RODOS thread that controls the toggling of the LED.

D.1.1 GPO Driver

Interface

Listing D.1 depicts the general purpose output (GPO) *Driver's DriverInterface*. The *Driver* consists of a single *Module* to access and modify the current state of the GPO pin. The enumerator `Key` (lines 3-5) provides the identifier for the *Driver's Modules*. The surrounding namespace (`dosis::io::gpo`) avoids name clashes of the `Key` enumerators of different *Components*. The rather simple GPO *Driver* uses a single Actuator *Module* representing the state of the pin as a boolean value (declared in line 7). The *DriverInterface* is declared as a combination of all its *Modules*, in this case only the `GPOState` *Module* (line 8). Finally, the `GPO` class, which is the *DriverInterface* used to access the *Driver's* modules, is defined (lines 9-13). The `state` member provides simplified access to the *Module's* `Interface` class (line 12).

```
1  namespace dosis::io::gpo {
2
3  enum class Key : uint8_t {
4      STATE,
5  };
6
7  using GPOState = dosis::Actuator<gpo::Key::STATE, bool>;
8  using GPOInterface = dosis::\gls{driver}Interface<GPOState>;
9  class GPO : public GPOInterface {
10 public:
11     using GPOInterface::GPOInterface;
12     GPOState::Generic::Interface& state = get<Key::STATE>();
13 };
14 }
```

Listing D.1: *DriverInterface* of GPO *Driver*.

Implementation

Listing D.2 presents the *Driver's* declaration, which is again part of the `dosis::io::gpo` namespace. This `GPODriver` inherits from the general `Driver` using the *DriverInterface* and the `GPODrive` itself as template arguments. The constructor (line 5) forwards the *deviceDef* to the parent constructor and sets the `GPODriver` instance's member variables accordingly. The following `getter` and `setter` methods are a copy of the template provided within the generic `Driver` (lines 15-20). This is a requirement of the C++ language to allow template specialization of those methods for the individual *Module's* identifiers. Finally, line 31 and line 34 declare these specific template specializations. Such a specialization must be explicitly declared as otherwise C++ cannot know that a specialized definition of those methods exists.

Listing D.3 depicts the definition of the *Driver's* methods. Note that this simple implementation relies on the RODOS provided `HAL_GPIO`. Therefore, it forwards the get and set operation to the specific methods of the RODOS GPIO interface.

D.1.2 Using a DriverInterface

Listing D.4 depicts the usage of the GPO *Driver* from within a simple RODOS thread. It contains an instance of `GPO`, i.e., the *DriverInterface*, and a duration used to regularly enable the LED. Line 18 and line 20 initialize the default-value and hold-time parameters of the *Actuator Module* to default-off, i.e., `false`, and the specified duration respectively. Afterward, a RODOS loop starting at line 22 synchronously activates the LED.

A *deviceDef* provides the required topic identifiers to connect a driver and its interface. Listing D.5 depicts the used *deviceDef* for a green LED. Note that this *deviceDef* does not actually reference any specific hardware, but assigns a human-readable name and a pair of RODOS topics to any instances using this *deviceDef*.

```
1  namespace dosis::io::gpo {
2  class GPODriver : public dosis::Driver<GPO, GPODriver> {
3  public:
4      template <typename DeviceDef>
5      GPODriver(DeviceDef device, const RODOS::GPIO_PIN pin, bool initVal = false)
6          : Driver<GPO, GPODriver>::Driver { device }
7          , m_gpio { pin }
8          , m_initVal { initVal }
9      {
10     }
11
12     void init() final;
13
14     template <auto KEY>
15     dosis::Result<Type<KEY>> getter()
16     {
17         return { reg<KEY>(), dosis::Error::OK };
18     }
19     template <auto KEY>
20     dosis::Error setter([[maybe_unused]] const Type<KEY>& val)
21     {
22         return dosis::Error::OK;
23     }
24
25 private:
26     gpio::GPIO m_gpio;
27     bool m_initVal;
28 };
29
30 template <>
31 dosis::Result<bool> GPODriver::getter<Key::STATE>();
32
33 template <>
34 dosis::Error GPODriver::setter<Key::STATE>(const bool& newval);
35
36 }
```

Listing D.2: GPO *Driver* declaration.

The final application instantiates the required *Driver* and the RODOS thread interfacing with said *Driver*. Listing D.6 depicts these instantiations. Line 2 instantiates the GPO *Driver* using the `GreenLED` *deviceDef* and the pin-number of the green LED. This example uses an STM32L496ZG-P Nucleo board, which includes a green LED connected to the pin C07 of the MCU. Afterward, line 7 initializes the controlling RODOS thread. It assigns a name, the *deviceDef* of the green LED and the interval of 100 ms for the regular actuation.

After compilation, the resulting binary will toggle the LED every 100 ms. Additional LEDs may be added by duplicating the respective lines for instantiation.

The *Driver* and controlling thread may be deployed to individual microcontrollers. In this case, the developer has to split the application file and generate separate binaries. An additional RODOS gateway connects the nodes and enables such a setup. No changes to the *Driver* or the *DriverInterface* are necessary as the DOSIS framework entirely abstracts this aspect from the users of the framework. More advanced examples, including distributed applications, can be found in the DOSIS GitLab repository, which is provided upon request.

```cpp
namespace dosis::io::gpo {

void GPODriver::init()
{
    m_gpio.configureOutputPin(m_initVal);
    RODOS::PRINTF("GPO␣Driver␣ready");
}

template <>
dosis::Error GPODriver::setter<Key::STATE>(const bool& newval)
{
    m_gpio.setPin(newval);
    return dosis::Error::OK;
}

template <>
dosis::Result<bool> GPODriver::getter<Key::STATE>()
{
    return { m_gpio.getPin(), dosis::Error::OK };
}
}
```

Listing D.3: GPO *Driver* definitions.

```cpp
class Blinky : public RODOS::StaticThread<> {
  dosis::io::gpo::GPO m_led;
  dosis::Duration m_interval;

public:
  template <class DeviceDef>
  Blinky(const char* name, DeviceDef led, dosis::Duration interval)
    : StaticThread<> { name }
    , m_led { led, name }
    , m_interval { interval }
  {
  }

  void run() final
  {
    /* set the default state to disabled */
    const bool defaultState = false;
    m_led.state.setDefaultSync(defaultState);
    /* set the on-time (hold-time) of the led */
    m_led.state.setHoldTimeSync(m_interval);

    TIME_LOOP(0, (m_interval * 2).asRodosLocalTime())
    {
      /* Trigger blinking cycle */
      dosis::Timed<bool> switchOn = { .time = dosis::TimePoint::now(), .value = true };
      m_led.state.setSync(switchOn);
    }
  }
};
```

Listing D.4: RODOS thread using the *Driver*.

```cpp
struct GreenLED {
  using InterfaceType = dosis::io::gpo::GPO;
  static constexpr char NAME[] = "green␣LED␣GPO";
  static constexpr int CMD_ID = 3001;
  static constexpr int DATA_ID = 3002;
};
```

Listing D.5: *DeviceDef* for a green LED.

```
1 // STM32L496ZG-P NUCLEO PC07, green LED
2 dosis::io::gpo::GPODriver greenLED { GreenLED(), RODOS::GPIO_039 };
3
4 using namespace dosis::literal;
5 constexpr dosis::Duration GREEN_LED_INTERVAL { 100_ms };
6
7 Blinky blinky { "green␣Blinky", GreenLED(), GREEN_LED_INTERVAL };
```

Listing D.6: Example application.

Appendix E

Radiation Test Additional Figures

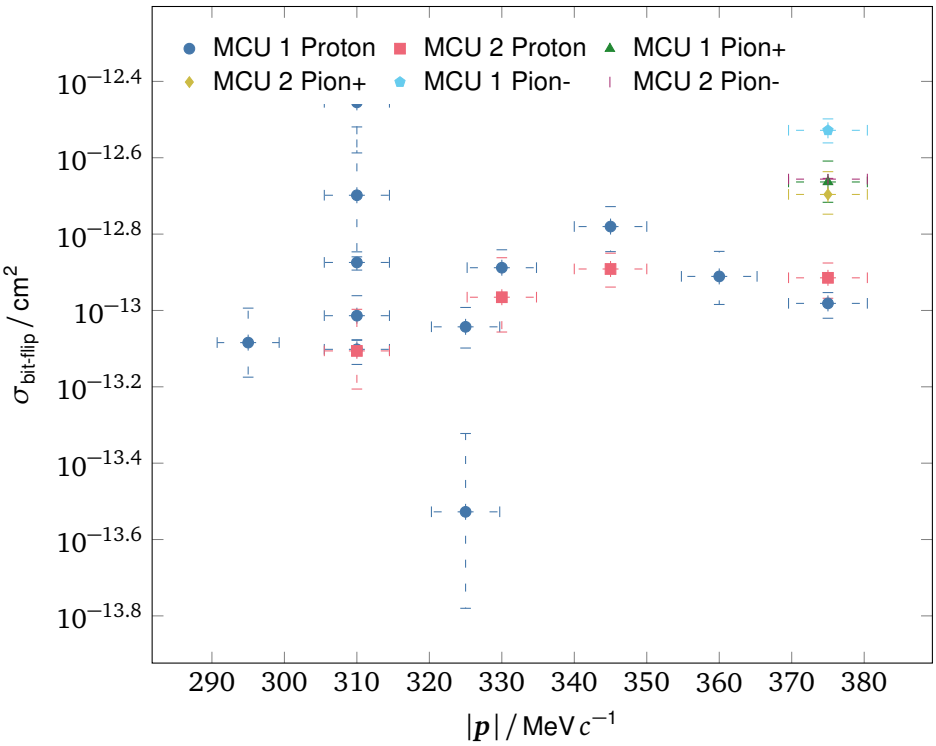


Figure E.1: Hardware corrected bit-flip cross-section of the VA41620 RAM and ROM memory regions for different particles and momentum for all test runs including 1-σ error bars. Impulse error bars show the πM1 beam line's full width at half maximum according to [166].

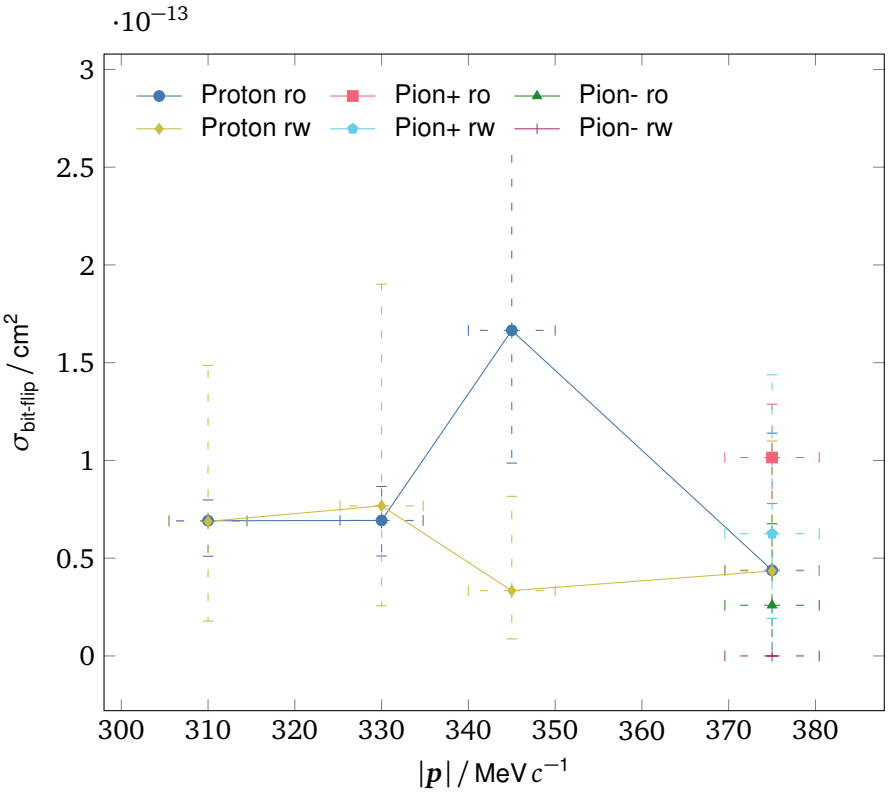


Figure E.2: Observed bit-flip cross-section in the monitored STM32L4 read-only (ro) and read-write (rw) memory region of MCU 2 for different test patterns, particles, and momenta including 1-σ error bars. Impulse error bars show the πM1 beam line's full width at half maximumaccording to [166] .

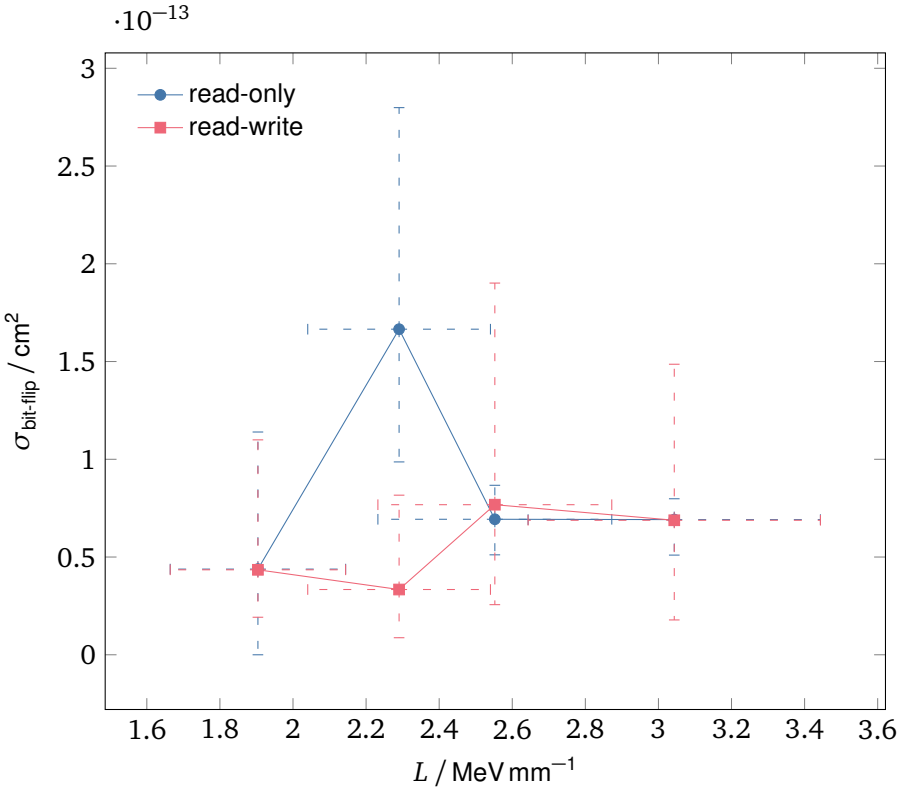


Figure E.3: Observed bit-flip cross-section in the monitored STM32L4 memory region of MCU 2 for different test patterns and momenta due to proton irradiation including 1-σ error bars.

DOSIS Terms

Actuator	An *Actuator Module* is similar to a *TimedSettable*, but the value will automatically revert to a default value after a specified timeout is reached. A detailed description is presented in section 3.5.5.
ActuatorHandler	An *ActuatorHandler* is the *ModuleHandler* of an *Actuator Module*. Within the implementation, it is represented by the `Actuator::Handler` class.
ActuatorInterface	An *ActuatorInterface* is the *ModuleInterface* of an *Actuator Module*. Within the implementation, it is represented by the `Actuator::Interface` class.
Component	*Components* are the largest basic building blocks within the DOSIS framework. A detailed description is presented in section 3.4. (*see also Driver* & *Daemon*)
ComponentImplementation	A *ComponentImplementation* is the counterpart to a *ComponentInterface* and implements the actual functionality of a *Component*. A detailed description is presented in section 3.4.2.
ComponentInterface	A *ComponentInterface* is the public interface to a *Component*. It provides a user-friendly way to remotely access the functionality of the *Component* from arbitrary nodes of the network. A detailed description is presented in section 3.4.1 and details about the implementation are presented in section 4.2.3.
Config	A *Config Module* is a special module which is used for *Daemons* only. It provides a way to modify the parameters of all other *Modules* within a *Component* with a single command. A detailed description is presented in section 3.5.7.
ConfigHandler	A *ConfigHandler* is the *ModuleHandler* of a *Config Module*. Within the implementation, it is represented by the `Config::Handler` class.
ConfigInterface	A *ConfigInterface* is the *ModuleInterface* of a *Config Module*. Within the implementation, it is represented by the `Config::Interface` class.
Daemon	A *Daemon* is a *Component* that, in contrast to a *Driver*, does not require direct hardware access and operates on data only.
DaemonImplementation	A *Daemon's ComponentImplementation* is called *DaemonImplementation*. Within the implementation, a class represents a *DaemonImplementation* if it inherits from the `Daemon` class. Details about the implementation are presented in section 4.2.5.
DaemonInterface	A *Daemon's ComponentInterface* is called *DaemonInterface*. Within the implementation, a class represents a *DaemonInterface* if it inherits from the `DaemonInterface` class. Details about the implementation are presented in section 4.2.3.
deviceDef	A device definition (*deviceDef*) binds a *Component's* instances to a specific pair of RODOS topics. More information on *deviceDefs* is presented in section 3.7.

Doable — A *Doable Module* is a special module intended for remote procedures. It does not represent a single value, but instead supports different data types for request and response. A detailed description is presented in section 3.5.6.

DoableHandler — A *DoableHandler* is the *ModuleHandler* of a *Doable Module*. Within the implementation, it is represented by the `Doable::Handler` class.

DoableInterface — A *DoableInterface* is the *ModuleInterface* of a *Doable Module*. Within the implementation, it is represented by the `Doable::Interface` class.

Driver — A *Driver* is a *Component* that, in contrast to a *Daemon*, requires direct hardware access.

DriverImplementation — A *Driver's ComponentImplementation* is called *DriverImplementation*. Within the implementation, a class represents a *DriverImplementation* if it inherits from the `Driver` class. Details about the implementation are presented in section 4.2.4.

DriverInterface — A *Driver's ComponentInterface* is called *DriverInterface*. Within the implementation, a class represents a *DriverInterface* if it inherits from the `DriverInterface` class. Details about the implementation are presented in section 4.2.3.

Interval — An *Interval Module* represents a value that is published to users of the *Component* at a specific frequency. While a user may change the frequency, the value itself cannot be modified and behaves like a *ReadOnly*. A detailed description is presented in section 3.5.3.

IntervalHandler — An *IntervalHandler* is the *ModuleHandler* of an *Interval Module*. Within the implementation, it is represented by the `Interval::Handler` class.

IntervalInterface — An *IntervalInterface* is the *ModuleInterface* of an *Interval Module*. Within the implementation, it is represented by the `Interval::Interface` class.

Module — *Modules* are the smallest building blocks within the DOSIS framework. They represent individual bits of a *Component's* interface and functionality. Details about *Modules* in general are presented in section 3.5 and details about their implementation are presented in section 4.2.2.

ModuleHandler — A *ModuleHandler* provides the actual functionality of a specific *Module* including a register to store the associated value and an implementation of the *Module's* default behavior. Within the implementation, a *ModuleHandler* is represented by the specific *Module's* inner `Handler` class. Details about the implementation are presented in section 4.2.2. (*see also ModuleInterface* & *ComponentImplementation*)

ModuleInterface — A *ModuleInterface* provides methods to remotely access a *Module's* functionality remotely. Within the implementation, a *ModuleInterface* is represented by the specific *Module's* inner `Interface` class. Details about the implementation are presented in section 4.2.2. (*see also ModuleHandler* & *ComponentInterface*)

ReadOnly — A *ReadOnly Module* represents a single value that cannot be modified via the *Component's* public interface. A detailed description is presented in section 3.5.1.

ReadOnlyHandler — A *ReadOnlyHandler* is the *ModuleHandler* of a *ReadOnly Module*. Within the implementation, it is represented by the `ReadOnly::Handler` class.

ReadOnlyInterface — A *ReadOnlyInterface* is the *ModuleInterface* of a *ReadOnly Module*. Within the implementation, it is represented by the `ReadOnly::Interface` class.

Settable	A *Settable Module* represents a single value that can be read and modified via the *Component's* public interface. A detailed description is presented in section 3.5.2.
SettableHandler	A *SettableHandler* is the *ModuleHandler* of a *Settable Module*. Within the implementation, it is represented by the `Settable::Handler` class.
SettableInterface	A *SettableInterface* is the *ModuleInterface* of a *Settable Module*. Within the implementation, it is represented by the `Settable::Interface` class.
TimedSettable	A *TimedSettable Module* is similar to a *Settable*, but setting the value is delayed until a specified time is reached. A detailed description is presented in section 3.5.4.
TimedSettableHandler	A *TimedSettableHandler* is the *ModuleHandler* of a *TimedSettable Module*. Within the implementation, it is represented by the `TimedSettable::Handler` class.
TimedSettableInterface	A *TimedSettableInterface* is the *ModuleInterface* of a *TimedSettable Module*. Within the implementation, it is represented by the `TimedSettable::Interface` class.

List of Figures

List of Tables

List of Algorithms and Programm Code