FRANK DANNEMANN

UNIFIED MONITORING OF SPACECRAFTS

UNIFIED MONITORING OF SPACECRAFTS

FRANK DANNEMANN

Dissertation

zur Erlangung des naturwissenschaftlichen Doktorgrades
der Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik
Lehrstuhl Informationstechnik für Luft- und Raumfahrt

Würzburg 2015

Frank Dannemann: *Unified Monitoring of Spacecrafts*, Dissertation
© 2015

SUPERVISOR:
Prof. Dr.-Ing. Sergio Montenegro

CO-SUPERVISOR:
Prof. Dr.-Ing. Görschwin Fey

LOCATION:
Würzburg

TIME FRAME:
December 2010 – November 2014

Dedicated to my family.

ABSTRACT

Data Management Systems (DMS) are an integral part of modern spacecrafts (S/Cs). No matter if they are integrated in a satellite, an interplanetary probe or within a launcher rocket – in all these space vehicles avionic systems take over the central tasks of data processing, steering and communication while at the same time ensuring a high degree of autonomy and dependability. An essential requirement for the S/C DMS in order to fulfill these challenging tasks is to know at any time during the operational phase what is happening inside the S/C. Only with a detailed knowledge about the internal state of the system reasons of current and past failures can be revealed, errors can be deduced and avoided in future and – finally – the mission goal can be fulfilled successfully.

Nowadays a huge variety of methods of gathering the internal information of the S/C exists. They strongly depend on the development phase, the source the information is coming from, and the intended user group which can range from embedded Software (S/W) developers over S/C engineers to mission control centers. The challenge and yet unsolved question is if it is possible to find a common method for handling as much of the S/C status information as possible in one common system which shall be used by all involved actors throughout all development phases of the S/C.

One of the main research contributions of this thesis is the development of a new infrastructure and methodology in order to embed a unified access to S/C status information within its development process – which we group together under the term *Unified Monitoring of Spacecrafts*. In addition, an universal monitoring system was implemented – called *Monitoring Framework*. This portable Framework (F/W) consists of embedded and non-embedded parts and is written in C++. It is intended to provide every person who is involved in designing, building, testing, and operating the S/C with all information which is needed at every phase in the life cycle of the S/C. Being highly configurable, the F/W is not only able to present the information needed in a way the user is convenient with, but also offers a huge amount of additional views and analyzing methods of the monitoring data. In contrast to the majority of embedded S/W in the space sector, the F/W can be configured and – if necessary – turned off at runtime.

The functionality of the F/W and the maturity of its design is proven not only on development boards based on typical space-processors like LEON2 and LEON3, but also in a satellite experiment in the laboratory as well as in a concrete satellite mission of the German Aerospace Center (DLR) – the *Eu:CROPIS* satellite. Particular impor-

tance is placed on the Unified Monitoring topic by a recently started European Union (EU)-funded project, wherein the Monitoring F/W will be massively expanded by self-configuring mechanisms and integrated in the avionics system of future launchers – thus providing them with currently not possible monitoring capabilities.

PARTIAL PUBLICATIONS OF THESIS RESULTS

Some ideas, text passages and figures presented in this thesis have appeared and were previously released in the following publications:

- Frank Dannemann. Towards Unified Monitoring of Spacecrafts. In *65th International Astronautical Congress, Toronto, Canada*. International Astronautical Federation, September 2014. URL `http://elib.dlr.de/90976/`. Paper IAC-14.D1.1.11
- Göksu, Murat. Entwurf und Implementierung einer zur Laufzeit konfigurierbaren Logging-Komponente für Satelliten. Master thesis, Universität Bremen, September 2014. URL `http://elib.dlr.de/91005/`
- Felix Schwieger. DLR_School_Lab Bremen: Lageregelungsexperiment "FloatSat". Handbuch für die Bodenstation und den Satelliten, 2014. URL `http://elib.dlr.de/91054/`
- DLR, EADS Astrium Space Transportation GmbH, AAC Microtec, and Politecnico di Torino. Massively extended Modular Monitoring for Upper Stages (MaMMoTH-Up). Horizon 2020 Proposal (Number: SEP-210132615), March 2014
- Frank Dannemann and Fabian Greif. Software Platform of the DLR Compact Satellite Series. In *Proceedings of 4S Symposium*, 4S Symposium, 2014. URL `http://elib.dlr.de/89344/`
- Frank Dannemann and Sergio Montenegro. Embedded Logging Framework for Spacecrafts. In *DASIA 2013*, ESA Special Publication, 2013. URL `http://elib.dlr.de/83042/`
- Müller, Sven. Entwicklung eines Rahmenwerks zur Nachrichtenprotokollierung für das eingebettete Echzeitbetriebssystem RODOS. Diploma thesis, Universität Oldenburg, 2012. URL `http://elib.dlr.de/88788/`

Success

To laugh often and much;
To win the respect of intelligent people
and the affection of children;
To earn the appreciation of honest critics
and endure the betrayal of false friends;
To appreciate beauty; to find the best in others;
To leave the world a bit better, whether by a healthy child,
a garden patch or a redeemed social condition;
To know even one life has breathed easier because you have lived.
This is to have succeeded.

— R.W. Emerson

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr.-Ing. Sergio Montenegro and my co-supervisor Prof. Dr.-Ing. Görschwin Fey for their great support and the ideas they contributed to this thesis.

I would also like to acknowledge the efforts of Sven Müller and Murat Göksu who implemented my ideas and brought them into practice during their theses – supervising them was always a pleasure for me.

In addition, I would like to thank Fabian Greif for his indispensable support during development and implementation of the flight version of the Monitoring Framework, and Felix Schwieger for integrating the framework's prototype into the School_Lab experiment – demonstrating the potential of the new idea of Unified Monitoring. A special thanks goes to Manuela Günther and Dr. Marco Scharringhausen for proof-reading this thesis and giving me valuable hints and suggestions.

The fruitful discussions with all of these people during the last four years helped me clarifying the essential topics I wanted to work on, and their valuable ideas finally led me to successful finishing this thesis.

Last but not least I would like to thank my children for missing their father during the time writing this thesis. Most of all I have to be grateful to my wife Tanja for her patience, constant support and great ideas during this tough time – I know it was not always easy.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ACS	Attitude Control System
AIT	Assembly, Integration and Test
AIV	Assembly, Integration and Verification
AOCS	Attitude Determination and Orbit Control System

API	Application Programming Interface
APID	Application Identifier
ASIC	Application-Specific Integrated Circuit
BDM	Background Debug Monitor
BSP	Board Support Package
CAN	Controller Area Network
CCSDS	Consultive Committee for Space Data Systems
C&DH	Command and Data Handling
CPU	Central Processing Unit
EM	Engineering Model
EPS	Electrical Power System
FM	Flight Model
DLR	German Aerospace Center
DMS	Data Management Systems
DSU	Debug Support Unit
ECSS	European Cooperation For Space Standardization
EGSE	Electrical Ground Support Equipment
ESA	European Space Agency
ESOC	European Space Operations Center
EU	European Union
FAR	Flight Acceptance Review
FDIR	Failure Detection, Isolation & Recovery
FPGA	Field Programmable Gate Array
F/W	Framework
GSOC	German Space Operations Center
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Dependent Layer
H/W	Hardware

ID	identifier
IMU	Inertial Measurement Unit
I/O	Input/Output
JSF AV	Joint Strike Fighter Air Vehicle
JTAG	Joint Test Action Group
LoC	Lines of Code
MOS	Mission Operations Services
OBC	Onboard Computer
OBDH	Onboard Data Handling
OBSW	Onboard Software
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
ONS	Onboard Navigation System
OS	Operating System
PC	Personal Computer
PDH	Payload Data Handling
PUS	Packet Utilization Standard
QM	Qualification Model
RAM	Random Access Memory
ROM	Read-Only Memory
RTM	Requirements Traceability Matrix
RTOS	Real-Time Operating System
SBC	Satellite Bus Controller
S/C	spacecraft
SPWRTC	SpaceWire Remote Terminal Controller
STM	Structure-Thermal Model
S/W	Software
SysML	Systems Modeling Language
TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol

TC	Telecommand
TM	Telemetry
TMR	Triple Modular Redundancy
TRL	Technology Readiness Level
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modeling Language
UTC	Coordinated Universal Time
USB	Universal Serial Bus
VM	Virtual Machine

Part I

INTRODUCTION

1 MOTIVATION

Data management systems are an integral part of modern S/Cs. Be it within a satellite, an interplanetary probe or within a rocket – in all these space vehicles avionic systems take over the central task like data processing, steering and communication while at the same time ensuring high dependability. Challenging tasks for todays avionic system are the increasing complexity of the S/C which are demanding for high high-performance onboard computers which at the same time should save resources like power- and memory consumption.

Being implemented as an embedded system, the so called Command and Data Handling (C&DH) subsystem of a S/C is responsible for all major tasks during operation. In the early beginning of S/C development most of its functionality has been realized by using pure mechanics, but with ongoing technological advances more and more electrical systems have been introduced. Nowadays the evolution of C&DH systems is moving to a state where the onboard software takes over most of the system's functionalities.

Sometimes a software update is the only way to safe the mission.

One of the most important advantages of handing over the majority of control tasks to the onboard software is, that in case of a failure a software update is the most of the time the only solution: the onboard software is the only component which can be changed (=updated) during operation. Unlike embedded systems residing on earth, after the launch of a S/C it is not possible to access the hardware directly any more. Only on very rare occasions a dedicated repair mission like the one to the Hubble-Space-Telescope is initiated [1]. This is of course only possible, when e.g. the satellite is in a reachable orbit [2] and a mission loss would be too expensive. Because of these reasons it is most important not only to test the S/C thoroughly while being build, but also to ensure, that all necessary information needed to analyze the S/C is sent down to the groundstation.

Software is most important in the lifecycle of a S/C

The lifecycle of a S/C and its avionic system is divided into several phases: collection of requirements, design & implementation, integration & test, commissioning, nominal operation and – at the end – decommissioning (e.g. through de-orbiting). During all these phases different actors in different roles (inter-)act with the S/C system. Depending on which phase the system is actually in, the information to be provided from the system to the outside differs not only in its content, but also in the way it has to be presented to the actor dealing with the S/C.

1 See e.g. `http://hubblesite.org/the_telescope/team_hubble/servicing_missions.php` for further details.

2 unlike e.g. within a deep space mission

Information is scattered all over the place

CHALLENGE: FIND A SYSTEM WHICH HANDLES ALL INFORMATION One challenging problem during the S/C construction- and operational phases results out of the actor/role/information/presentation diversity described above: there is no common system inside the S/C which manages *all* information. If so, such a system has to be able to handle all kinds of information sources and sinks, meaning the places where information is generated and where it is provided to. It needs to collect, (pre-)process and pass-on the information according to the user's needs – covering the whole S/C information flow. What kind of information exist and details about the flow are topics of the Chapters 9 and 10.

Important information is not accessible in space

CHALLENGE: MONITOR A SYSTEM IN SPACE Not only the diversity of information, also the access to it differs from systems on earth in comparison to S/Cs: once launched, there is normally no GUI or terminal which can be accessed in order to directly determine and search for the possible errors in case of a failure or misbehavior. Typically only historical or realtime telemetry can be analyzed which often hold only a subset of the information needed and provided e.g. by the internal debugging features of the onboard computer. Moreover, even if the challenge of finding a common system for handling all kind of S/C status information is successfully solved, in the emergency case of a Failure Detection, Isolation & Recovery (FDIR) activity the limited bandwidth within most mission scenarios for transferring data to ground inhibits the downlink of *all* relevant information due to the huge amount of data. This shortcoming is further analyzed in Chapter 13, Paragraph 13.2.4.

CHALLENGE: GETTING VISIBILITY INTO HARDWARE Besides challenges arising from functional or operational needs as described above, the miniaturization in electronic hardware leads to demanding requirements on the software: it becomes more and more difficult, to *physically* attach hardware debug tools like oscilloscopes, logic analyzers or Read-Only Memory (ROM) emulators to the printed circuit boards of an embedded system. This is especially evident for the so-called *system-on-chips*, which contain everything (microprocessor, memory, UART, etc.) inside one single chip (Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC)).
Therefore a demanding requirement put on the embedded software to be developed is, to provide means for detailed debugging *from within* the software itself [see 77, sec. 10, p.323 ff].

Software-based monitoring of S/Cs has been little explored until now

Though software-based monitoring of S/Cs has always been an important topic, this research area was not very active in the past: like the S/W development part in a space mission was underestimated for quite a long time, also the monitoring capabilities resulting from systems, which are more and more software-controlled, are explored to only a small degree [3]. The reason for this lies in the evolution of the C&DH subsystem: started with purely hardware-based solutions, the development is heading towards transferring as much control to the onboard software as possible. This trend is owed to some degree to the fact, that modern design techniques result in most reliable S/W system. The status regarding the design of current C&DH systems and a closer look on the H/W-S/W interaction will follow in Section 5.

3 This becomes evident and is reflected by the fact, that only very few literature exists in this area.

2 IDEA & VISION

In order to gain a better understanding of the improvements resulting from the idea of Unified Monitoring presented within this work which are going beyond the state of the art in monitoring embedded S/C systems, an analogy to the diagnosis of a patient offers a good example:
Would it not be the dream of every doctor to have a *universal plug* to the human body, from which he can read out all the information which are relevant for his special field of work in every detail he wants – for the past as well as for the presence? Having such a powerful tool would put him in the situation to provide the patient with diagnostics of his illnesses without having to use countless different measurement devices: one device to measure the blood-pressure, one device to measure the blood sugar level, one device to measure the temperature of the body, and so on.

The new idea: a universal Monitoring System

Going back to the scenario of a space mission where the S/C takes over the role of the patient and EGSE engineers or the operational personnel in the control room serve as the doctors interested in the health status of their S/C-"patient", the idea of the universal plug for the human body can be transferred as follows: a monitoring system is needed which manages all kinds of information describing the state of the S/C, hence acting as a central collecting point for all information sources and sinks.

For bringing this new information system into practice also a completely new toolset has to be developed. Within this thesis the first steps into this direction are undertaken, replacing the standard debugging world of the onboard software with a monitoring framework which is at first connected to the housekeeping and TM/Telecommand (TC) world of a S/C, but has the potential – with proper support from groundstations in the future – to replace the traditional way of monitoring S/Cs.

The new toolset: the Monitoring Framework

Like for the human body, where the doctor's measurements should result in a less discomfort for the patient, the same applies for our space "patient", the embedded avionic subsystem: the one single framework for information processing which shall be integrated into the S/C, should not or only very little influence the normal operation, meaning as well the functional as the timing behavior.

Non-intrusive Measurement

But contrary to the patient a S/C cannot be accessed directly. Though the restrictions resulting from the remote scenario in orbit (e.g. timing delays) cannot be changed in general, we can still try to optimize the access to the S/C data by reducing the transmitted data to only

Optimized Data Transfer to Groundstation the relevant information and try to transfer them in the most efficient way. This subject is also covered by this thesis.

In Chapter 3 all of the above mentioned visionary points will be addressed during the introduction of the new idea called *Unified Monitoring*.

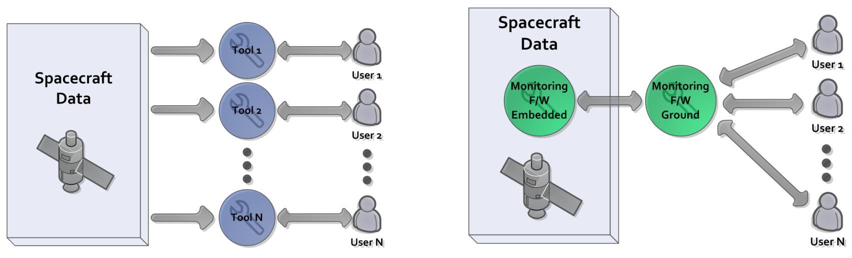
3 APPROACH & CONTRIBUTIONS

The topic of this doctoral thesis is to develop a new infrastructure and methodology in order to embed a unified access to S/C status information within its development process – which we group together under the term *Unified Monitoring of Spacecrafts*.

In addition – within the scope of several theses, internships, student studies and also in cooperation with the Department of Aerospace Information Technology of the University of Würzburg – a universal monitoring system was implemented – called *Monitoring Framework*. This F/W acts as the basis for all further efforts on combining the different information worlds (described in Chapter 9) within a S/C into one information system which serves all purposes. It is the technical implementation supporting the Unified Monitoring concept. Information units will be collected at a central place within the C&DH unit. They have exactly one source, but within the F/W they can be used and deployed in many ways.

The technical evolution between the current state of the art in handling and accessing the data of a S/C and after integrating and using the new Monitoring F/W into the system is depicted in Figure 3.1a and Figure 3.1b.

The new Approach: a universal Monitoring Framework



(a) State of the Art

(b) With new Monitoring F/W

Figure 3.1: Evolution in the Access to Spacecraft Data by adding the Monitoring F/W

The different users – later on described in Chapter 7 – do not have to deal with a heterogeneous toolset anymore. They benefit from the design of the Monitoring F/W which is not only running embedded, but is completed by additional F/W components running on the user's host computers on ground.

Within various tests we will proof, that the overhead by using the concrete implementations of the F/W (see below in Section 3.1) does not only stay on a tolerable level, but rather fades away when looking at the various benefits the F/W offers.

FEATURES OF THE MONITORING FRAMEWORK As already mentioned in the MOTIVATION Chapter, the huge amount of monitoring data handled by the F/W in contrast to the limited transmission bandwidth is longing for new ways to get the *relevant* information to ground. Up to now the diverse tools in Figure 3.1a could be configured to some degree by the users, but not all of them offer the adjustment of the amount of data to be collected or the changing of the granularity of recorded information. This is especially true for the tools which are used for accessing system debug information.

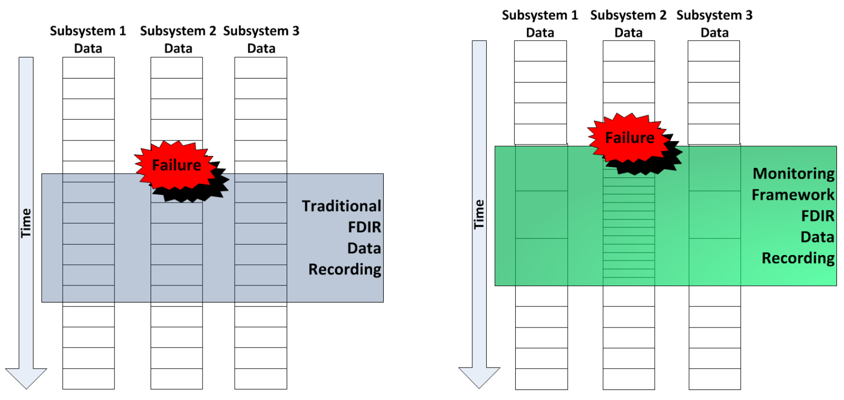
As the change in the direction of the arrows in Figure 3.1 indicates, the Monitoring F/W offers the possibility to be configured at runtime by the user in terms of granularity, depth and criticality of information which is also a novelty in the conservative area of embedded avionics software in space. The flexibility in terms of configuration of the acFW is achieved – amongst others – by the integration of new filtering and adaption techniques. One of these techniques concentrate on recording only the relevant information which is most important in case of an FDIR activity. Figure 3.2 visualizes – in a simplified view – this new technique for the monitoring of three different S/C subsystems. The traditional way of recording debugging data (shown in Figure 3.2a) from these subsystems in case of an emergency event does not take into account the user needs in terms of looking into more detail into the data of the *affected* subsystems. In contrast, the new Monitoring F/W has the capability to be adapted in terms of the granularity and criticality of the debugging data to be recorded (e.g. by adapting the F/Ws's logging level). Within this example the available bandwidth is used for transferring as much data from the faulty and/or misbehaving subsystem as possible to ground. Therefore – as depicted in Figure 3.2b – the amount of data recorded from the subsystems 1 and 3 is reduced significantly, whereas the subsystem 2 is monitored in much greater depth.

Evolution in Recording of S/C Data

In addition to the deeper insight into the S/C during its operation, the interoperation of (1) the way how the monitoring information is handled by the F/W and (2) the F/W's composition out of embedded and non–embedded parts lead to an enhanced and easier traceability of information coming from the S/C: *enhanced* in terms of the degree of details, and *easier* in terms of access to this information by the users. Figure 3.3 depicts how the users get much more insight in the origin/source and involved processing steps of every piece of S/C data by using the Monitoring F/W.

Evolution in Traceability of S/C Data

As it can be seen in Figure 3.3a, traditionally debugging data is transferred to external facilities like Electrical Ground Support Equipment (EGSE) equipment and afterwards stored into one big data chunk



(a) State of the Art (b) With new Monitoring F/W

Figure 3.2: Evolution in the Recording of relevant Spacecraft Data by adding the Monitoring F/W (simplified view; horizontal bars representing recorded monitoring data)

[4]. If the user wants to split the data up again in order to see only that part in which he is interested in, quite some effort has to be undertaken and external tools have to be used to analyze the S/C TM data (e.g. through data mining techniques [75]). Figure 3.3b shows how the new Monitoring F/W will improve this situation significantly:

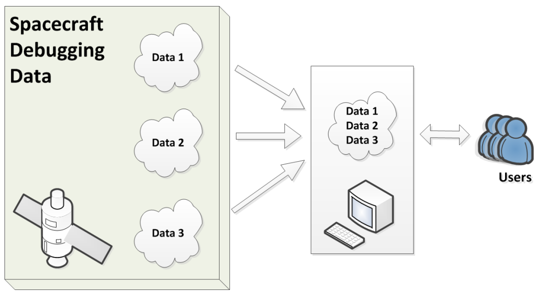
1. S/C data will be collected and stored in so called *Monitoring Items* which contain the data structures for storing information within the single information world.

2. Once the S/C data is transfered to this information world it is transported from the embedded part of the F/W to the non-embedded counterpart (like the check-out equipment in the integration room). Here it is very easy for the user to access the information by using the interfaces and tools offered by the Monitoring F/W and get them out of the single information world again.

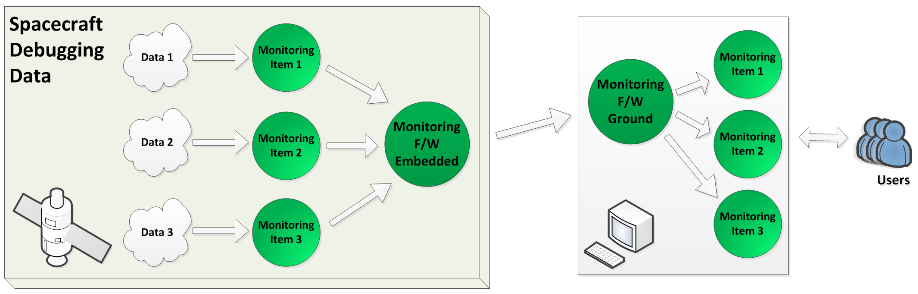
3.1 MONITORING FRAMEWORK IMPLEMENTATIONS

Including Unified Monitoring and the Monitoring F/W into the on-board S/W is the key to make it an integral part of the S/C. To achieve this, two versions of the F/W were implemented: the Monitoring F/W prototype and the successor – the Monitoring F/W flight version. The F/W prototype can be understood as a first testbed for gathering experiences with the new monitoring technique, containing e.g. also parts running non-embedded within a simulated groundstation. On

4 If the "normal" telemetry has been sent to a groundstation using the Consultive Committee for Space Data Systems (CCSDS) Packet Utilization Standard (PUS) protocol, the contained information has already been split up into separate information chunks per subsystem using the corresponding Application Identifier (APID).



(a) State of the Art



(b) With new Monitoring F/W

Figure 3.3: Evolution in the Traceability of relevant Spacecraft Data by adding the Monitoring F/W (simplified view)

the other hand, the flight version is a reduced version of the F/W: it is originated in the prototype and its lessons-learned, but tailored and tested to be integrated in the flight S/W of a real space mission.

Monitoring Framework Prototype (based on RODOS)

The prototype of the F/W is seamlessly integrated into the Real-Time Operating System (RTOS) RODOS and its middleware which is already used as the operating system of choice for many S/C avionic units. It can be deployed in a distributed scenario and is therefore suitable also for missions where multiple onboard computers communicate with each other and together fulfill one common mission goal – acting as a closed system to the outside. These onboard computers can either reside within one single S/C (like a multi-board/multi-core bus computer or like a concept with separated payload and bus computer (cf. to Section 5.1)), on different S/Cs (like in a satellite swarm or fractionated S/C [5]) , or even on completely different systems like at a groundstation or within a launcher. Like RODOS itself, the F/W will provide a simple Application Programming Interface (API), making it easy for application developers to understand and use it in order to provide monitoring information to the F/W.

Monitoring Framework Flight Version (RTOS independent)

After proving its efficiency in terms of functionality, usability and performance on the basis of various demonstrators (Linux-based and several space processors), the Monitoring F/W will be tested within a real S/C – the DLR Eu:CROPIS mission. Within this mission a S/W platform was developed which makes the integrated software compo-

5 A fractionated S/C is a rather novel architecture of space systems, where functionalities are spread over multiple heterogeneous modules [32].

nents independent from the underlying hardware, operating system and middleware by introducing various kinds of abstraction layers. Therefore, the integration of the Monitoring F/W as an additional service [6] into this library offers completely new possibilities, like expanding the usage of the Monitoring F/W throughout the whole space system or – like the prototype – within distributed scenarios.

6 Another important integrated service is a CCSDS PUS software stack.

4 CHAPTER OVERVIEW

The different parts into which this thesis is structured are depicted in Figure 4.1.

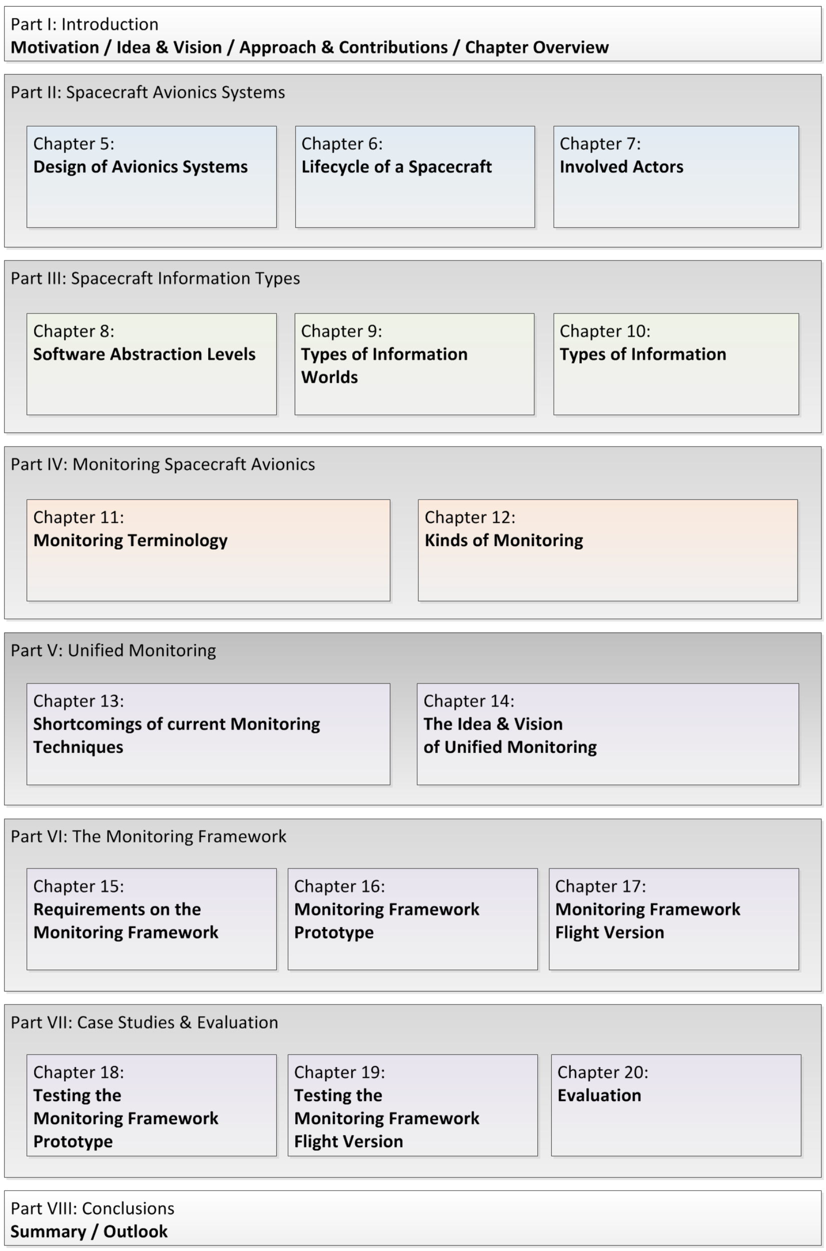

Part I: Introduction
Motivation / Idea & Vision / Approach & Contributions / Chapter Overview
Part II: Spacecraft Avionics Systems
Chapter 5: Design of Avionics Systems
Chapter 6: Lifecycle of a Spacecraft
Chapter 7: Involved Actors
Part III: Spacecraft Information Types
Chapter 8: Software Abstraction Levels
Chapter 9: Types of Information Worlds
Chapter 10: Types of Information
Part IV: Monitoring Spacecraft Avionics
Chapter 11: Monitoring Terminology
Chapter 12: Kinds of Monitoring
Part V: Unified Monitoring
Chapter 13: Shortcomings of current Monitoring Techniques
Chapter 14: The Idea & Vision of Unified Monitoring
Part VI: The Monitoring Framework
Chapter 15: Requirements on the Monitoring Framework
Chapter 16: Monitoring Framework Prototype
Chapter 17: Monitoring Framework Flight Version
Part VII: Case Studies & Evaluation
Chapter 18: Testing the Monitoring Framework Prototype
Chapter 19: Testing the Monitoring Framework Flight Version
Chapter 20: Evaluation
Part VIII: Conclusions
Summary / Outlook

Figure 4.1: Overview of the Thesis

After the introductory part, Parts II to IV are forming the theoretical background for this thesis. They start with a view on the design and lifecycle of a S/C in general, also taking into account the actors involved in every mission phase (Chapters 5 to 7). Going into more detail in the S/Cs information worlds, within the Chapters 8 to 10 a closer look is taken on the area of information abstraction levels, how these levels correspond to the various information worlds within a space system and what kind of data it takes to build, control and operate a S/C. The theory part closes by defining the term *Monitoring* and describing the monitoring techniques which are currently available (Chapter 11 and 12).

Theory Part

Parts V to VIII contain the main parts of this work. First, in the Chapter 13 and 14 the shortcomings of the current technologies for monitoring S/Cs are summarized and the new idea of Unified Monitoring is presented. Afterwards, Chapter 15 to 17 concentrate on the realization of this idea: the Monitoring Framework. The requirements on the framework are forming the starting point from which the design of the prototype and its successor, the flight version, could evolve. The detailed tests of both frameworks can be found in Chapter 18 and 19. Chapter 20 closes the main part with an evaluation of the thesis' outcome by addressing the current shortcomings identified.

Main Part

At the very end Part VIII summarizes the achievements made and provides an outlook on possible future activities in the field of Unified Monitoring.

Part II

SPACECRAFT AVIONICS SYSTEMS

Because of these considerations, hardware engineers are inclined to suggest that product functionality is best done in software rather than in additional hardware. This is not because they are lazy; it is because a product with more software and less hardware will in most cases be a better product.

— David E. Simon[77]

5 DESIGN OF AVIONICS SYSTEMS

In this part an overview of the State of the Art in the area of S/C avionics is given, as the design of the embedded system is very closely linked to the monitoring capabilities it offers to the user. To give an example, it is obviously far easier to monitor the payload of a satellite if it is controlled completely by the OBC of the satellite bus.

Because the main field of work of this dissertation takes place in the embedded S/W area, the H/W part at the beginning of this chapter will be reasonably short. It is focused on the role of the OBC within the S/C environment, as the OBC is responsible for executing the onboard S/W.

5.1 HARDWARE

Payload and Satellite Bus

Taking a satellite as a typical example for a S/C and also as the main use case environment within this thesis, it can be divided into two major components: the payload compartment and the satellite bus. Both components take over typical task which are listed in Table 5.1 (cf. to [8]).

Payload-Tasks	Bus Tasks
Scientific Instruments	Boardcomputer (OBC)
Meteorological instruments	Power Supply
Navigation	Communication
Communication	Navigation & Orbit Control
Earth Observation	Thermal Control
	Structure

Table 5.1: Examples for Tasks of Payload and Bus

OBDH-only vs. OBDH & PDH

While almost all payloads make use of the bus services like power, thermal control and navigation & orbit control, this differs when it comes to the OBC. Depending on the boundary conditions of the mission (size, mass, power-consumption, . . .) it makes sense to let the OBC of the satellite bus also care for all data handling and communication issues, which are directly linked to payload operation. Only if the payload needs very special and dedicated H/W (which is e.g. optimized for image processing) or has high demands on downlink data rates (e.g. using X-Band), a separation of payload and satellite-bus data handling makes sense. These two concepts described above are also depicted in the Figures 5.1 and 5.2 [8].

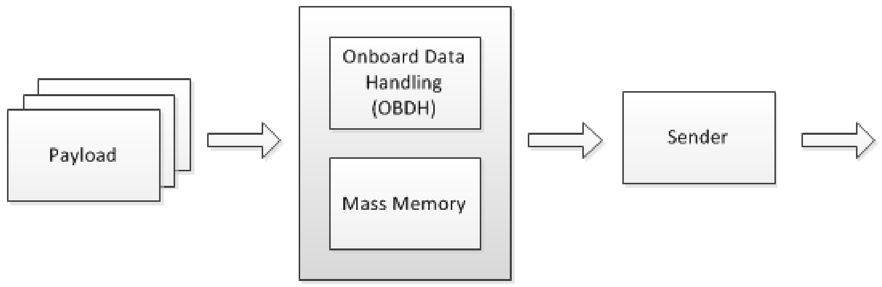


Figure 5.1: The payload using the OBDH of the satellite bus

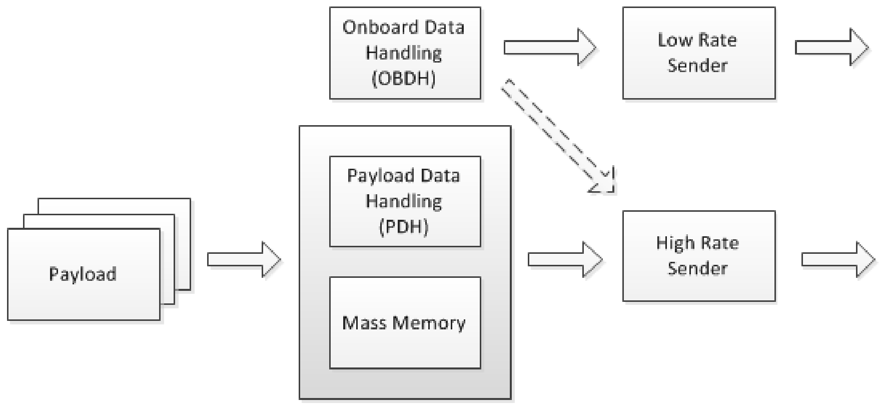


Figure 5.2: Payload with own data handling and high speed downlink capabilities

Nowadays, as electronics get more and more miniaturized, there exist also stages between these two concepts. One example are modern scientific instruments, which are equipped with powerful computers, doing most their operations autonomously, and requiring only very few and high-level commands from the main OBC [7].

Taking the harsh conditions in the space environment and the lack of a possibility to exchange defect H/W components into account, redundancy is a common concept in order to keep the system alive during operation [8]. While Triple Modular Redundancy (TMR) is often used for the internal design of modern space processors, on the level of components *double modular redundancy* is the state of the art. A common scenario are two onboard computers which are running in

Redundancy Aspects

7 The MASCOT mission is a good example for a design, where the intelligent instruments attached to the OBC are taking over the processing of their measured data. Confer to [38] for information about the MASCOT mission, and to [30] for an example of such an intelligent instrument.

8 The other concept is based on choosing high reliable and space-proven H/W components

cold or hot redundant mode and are connected to likewise redundant components like the TM and TC system or the Input/Output (I/O) interfaces of the devices. If there is a separate Payload Data Handling (PDH)-unit and a dedicated and also separated Attitude Control System (ACS)-part, these are also redundant and are connected to the OBC. On the other hand, not all parts of the satellite bus are designed in a redundant way: thermal-control, power-control and of course the structure are typical examples for non-redundant subsystems. Figure 5.3 gives an overview of how the OBC is embedded in the overall H/W system of the satellite bus.

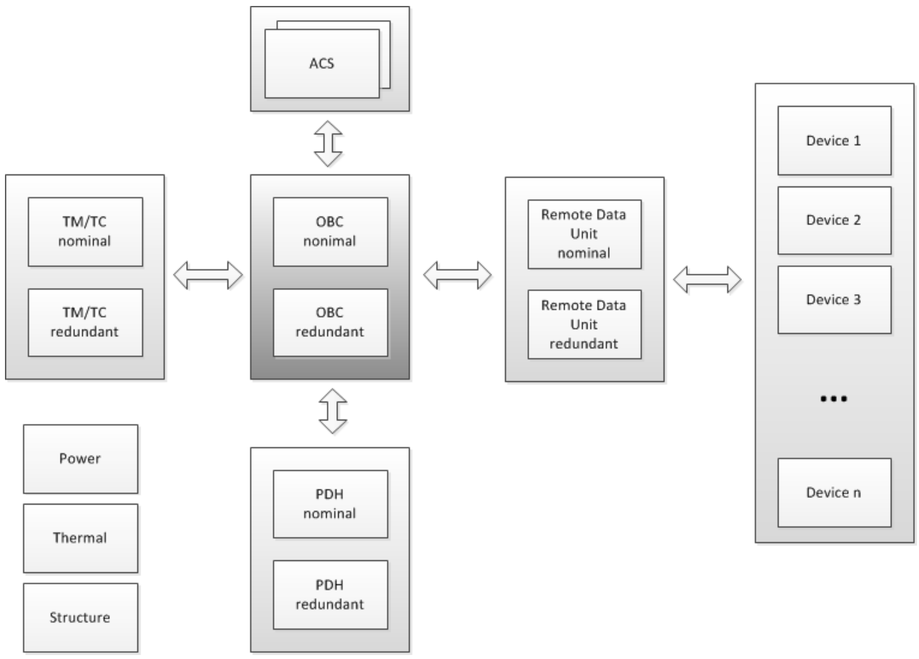


Figure 5.3: The OBC is embedded in the overall system, typically by using redundancy

5.2 SOFTWARE

The S/W running on the board computer of a satellite does not differ much from other embedded S/W, meaning that when designing and implementing the S/W the following qualities have to be taken into account (cf. to [11]):

- Reliability
- Portability
- Maintainability
- Testability
- Re-usability

- Adaptability/Extensibility
- Readability of the source code

What sets the space system apart is its inaccessibility to standard types of maintenance and repair – resulting from the fact, that it must be operated in a complete remote scenario hundreds of miles from earth [35]. Because of this high degree of autonomy of most space missions, special emphasis is put on reliability of the S/W, and – when it comes to functionality of the code – to the S/W update mechanisms. Finally, monitoring of a S/C during its mission is an extremely demanding task which is addressed in the dedicated Chapter 12: KINDS OF MONITORING.

In Space S/W is even more important...

Within a modern S/C various kind of embedded S/W components take over the task of command and data handling. Nowadays systems engineers tend more and more to solutions, where they rather implement main parts of the system's functionality in S/W than using –taking it to the extreme – dedicated H/W. The main advantage behind this paradigm shift is quite obvious: only via S/W uploads it is possible to change, reconfigure or repair a failure after the S/C has been launched. [9]

The architecture of the onboard S/W strongly depends on the S/C it is running on, and is dependent on the functionality it has to provide and the resources which are available and/or needed. E.g. an upper stage differs so much in its H/W architecture from a satellite, that it is barely possible to use the same kind of S/W on both systems.

Therefore the S/W architecture to be selected and implemented for the Onboard Data Handling (OBDH) subsystem of the S/C depends to some degree on the chosen H/W. And as the H/W again is influenced by the resources available (meaning not only the size available in the electronic compartment, but also the available power, the heat dissipation, the shielding, and many other things), the S/W is sometimes also affected by these factors. Taking a nano-satellite as an example, because of its limited resources in term of processing power and because of its limited functionality, these kinds of systems are often equipped with a S/W that just consists of a single big main loop, giving up a lot of additional – but in this case not needed – functionality of e.g. a modern embedded Operating System (OS) could provide.

Common solutions for embedded S/W systems are the usage of (cf. to [77]):

1. Round Robin ("big loop")
2. Round-Robin with interrupts ("interrupt server")
3. Function-Queue-Scheduling Architecture
4. Real-Time Operating System (RTOS) architecture

9 Other important reasons are the price one has to pay for dedicated H/W and the time it takes to procure radiation hardened or fault tolerant H/W components

While solution 1 to 3 are sometimes called *dedicated* S/W, the selection of an RTOS brings more functionality and flexibility to the system, of course with a certain degree of overhead.

As this thesis will concentrate on the category of small to medium sized satellites in the range up to 150 kilograms, the focus will be put on the S/W used for this kind of missions. Within this class of satellites the used embedded H/W is most of the time powerful enough to allow the usage of an embedded OS, and – for inter-task communication – an integrated middleware.

Payload and Bus Software

Taking a satellite as depicted in Section 5.1 as an example, the embedded S/W can be divided into two types:

BUS SOFTWARE The S/W running on the satellite bus has the main purpose to autonomously steer the satellite, handle commands from the groundstation and transfer data down to earth. Only in very rare cases (e.g. if the payload computer does not have enough processing power) the bus S/W sometimes takes over the task of payload data processing, but normally this is done by the payload S/W.

PAYLOAD SOFTWARE The payload S/W is responsible for processing data coming from the instruments. It is sometimes also responsible for storing data (e.g. if the satellite bus does not have enough memory to offer) or for a direct download to earth (e.g. if the satellite bus does not have enough bandwidth).

5.2.1 *Bus Software*

Tasks of the Bus Software

As later on described in the Chapters 14 and 15, the Unified Monitoring approach is based on introducing a general Monitoring F/W in the bus S/W (either RTOS or middleware) of the satellite bus, therefore in the following description we will focus on the tasks typically taken over by the bus S/W. These tasks are often implemented as dedicated S/W applications running on top of the embedded OS. Therefore the terms *task* and *application* will be used in the same manner throughout this thesis, both concentrating on the functionality the piece of S/W is adding to the satellite boot image [10].

Table 5.2 gives an overview of responsibilities the bus S/W takes over, divided in functional task and non-functional requirements.

While the functional tasks mainly concentrate on the steering and control of the satellite, the non-functional tasks are dealing with FDIR

10 Another reason for concentrating on the bus S/W is – besides the focus on the basis of the Monitoring F/W – that the payload S/W strongly depends on the mission. It is hard to find a S/W architecture which is common for the various different missions that exist, though a complex payload S/W often does not differ that much from modern bus S/W architectures.

Tasks (Functional)	Requirements (Non-Functional)
Housekeeping	Fault Tolerance
Attitude Control	Robustness
Navigation	Real-Time
Telecommand & Telemetry	Fast Recovery
Thermal Control	
Power control	
Payload-Data Processing (if necessary)	

Table 5.2: Responsibilities of the Bus Software

techniques like fault tolerance and robustness. Functional and non-functional tasks also differ from the architectural design regarding their implementation within the onboard S/W: while functional tasks are normally concentrated within one single application[11], the responsibility of non-functional tasks is often spread over many – if not all – components of the onboard OS and its services provided.

Bus Software Architecture

Figure 5.4 gives a general overview of the layered architecture of the bus S/W. Similar architectures can be found in most embedded systems (taken from [43]): the target H/W is supported by the Board Support Package (BSP), upon which the RTOS with its kernel runs, accompanied by several other components like protocols, libraries, drivers, etc. The functional tasks described in Table 5.2 belong the topmost application layer. There is a wide range of literature on the topic of embedded S/W design – especially about the design and usage of RTOSs. Herein all of the constituents shown in Figure 5.4 are explained in great detail. Most of them are using a concrete RTOS implementation as an example for explaining the concepts behind. Following this didactic principle, within this thesis we will use the RODOS RTOS as an example (for the reasons behind this choice and details of RODOS please refer to Section 5.2.1.2) [12].

5.2.1.1 *Real-Time Operating Systems*

Even though it is possible to drive a mission with a S/W program implemented as one big main-function (control loop), most of the time the complexity of the system and the number of tasks to be handled by the S/W will quickly become so unmanageable that the use of an embedded RTOS becomes unavoidable [10].

11 Within this dissertation the term *application* is used in order described a S/W entity running on top of the RTOS. In literature other terms are used as well, as there are: components, programs, building blocks, ...

12 For the reader who is interested in the topic area of RTOS design it is recommended to take a look into the books of Q. Li ([43], using VxWorks as example RTOS) and David Simon ([77], using µC/OS as example RTOS).

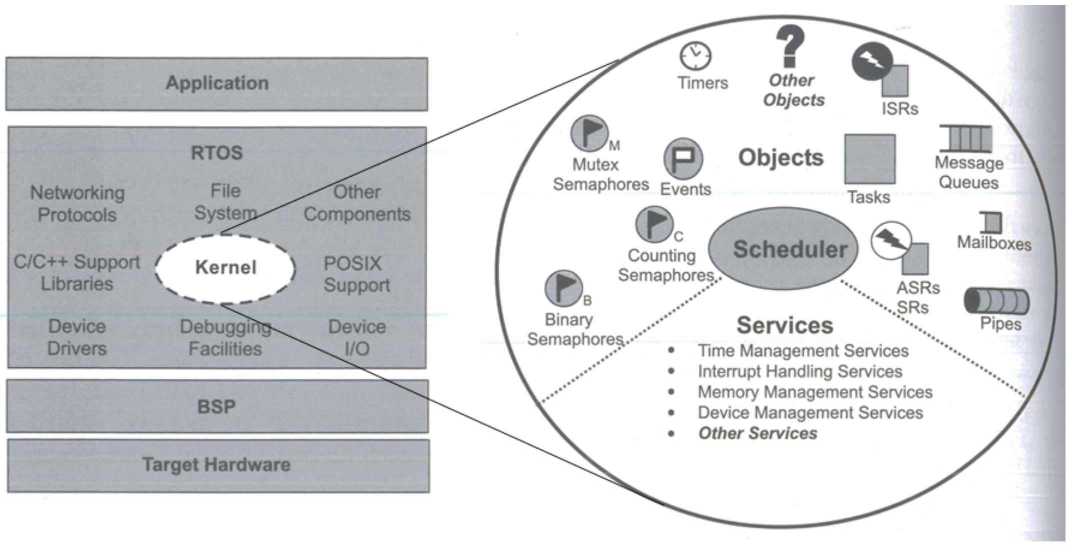


Figure 5.4: Architecture of the Bus Software [43]

There exist several embedded RTOS on the market, and one has to carefully choose an appropriate OS according to the mission needs [7].

In addition to the S/W qualities mentioned at the beginning of this section, also criteria like

- cost/license issues (incl. support),
- availability of sources,
- maturity (e.g. Technology Readiness Level (TRL)-Level [47]),
- performance (e.g. resource consumption),
- functionality (e.g. support for distributed systems), and
- complexity (e.g. easy-to-understand API)

have to be taken into account when it comes to select an appropriate RTOS for a specific mission[13].

DIFFERENCE BETWEEN SOFT- AND HARD REAL-TIME SYSTEMS Broadly spoken, real-time systems are systems which have to react within a certain time on internal and/or external events. If these timing constraints are not met, this will result in malfunction or performance losses. Therefore the accurate functional behavior of the system depends not only on the correctness of the produced results, but also on the point in time when these results are available: a correct

A correct reaction at the wrong time is an error!

13 A comprehensive list of available RTOSs, their properties and an evaluation matrix for comparison was created during a project-internal study within the frame of DLR's *SHARC* project [45].

result at the wrong point in time is as well an error as a wrong result within time – a late reaction to an external event is as wrong as a wrong or even no reaction.

Depending on how strict these time constraints are, one can distinguish between soft and hard real-time systems (cf. to e.g. [43, sec. 1.2.3] or [39, sec. 1.5.1]):

HARD REAL-TIME SYSTEM If a deadline is not met in time, the consequences for the system will be disastrous, therefore these systems are forced to react within the defined time frame. Examples for these kind of systems are flight control systems or airbags.

SOFT REAL-TIME SYSTEM Deadlines are important, but it is not critical if they were not met – the system will continue to work correctly. Examples: Data Acquisition Systems, DVD players, ...

In the satellite domain one can very often find soft real-time systems, and if a deadline is missed most of the time nothing disastrous will happen (e.g. antennas turned on too late for downlink will might cause some data loss, solar cells not shunted in time will might cause some overload of the batteries, ...).

5.2.1.2 *RODOS*

RODOS is the acronym for Realtime Object-Oriented Distributed Operating System. As an embedded OS it is specially designed for space applications, but fits perfectly to all applications demanding high dependability. The RODOS real-time kernel and middleware provide an integrated object-oriented framework to multitasking resource management and to network-based communication infrastructure. Although targeting minimal complexity, no fundamental functionality is missing, as its microkernel provides support for resource management, thread synchronization and communication, input/output, and interrupts management. The system is fully preemptive and uses priority-based scheduling and round robin for threads sharing the same priority level. RODOS is written mainly in the C++ programming language, some H/W dependent parts are written in C and target specific assembly language. Despite RODOS is intended for stand alone use in embedded systems, the user can also run it on-top of Linux as guest OS. It is built as a static library, so that a user can link his code against this library and run the resulting binary.
The overview of the RODOS architecture is shown in Picture 5.5. Besides the Hardware Dependent Layer (HDL), core and management layers RODOS also comes with an (optional) middleware. The users main interface is therefore the application module, which encloses one or more threads. In each application module, the programmer can create an application object that defines an application name and an identification number. By using the publish/subscribe mechanism from the middleware it is easily possible not only to establish an

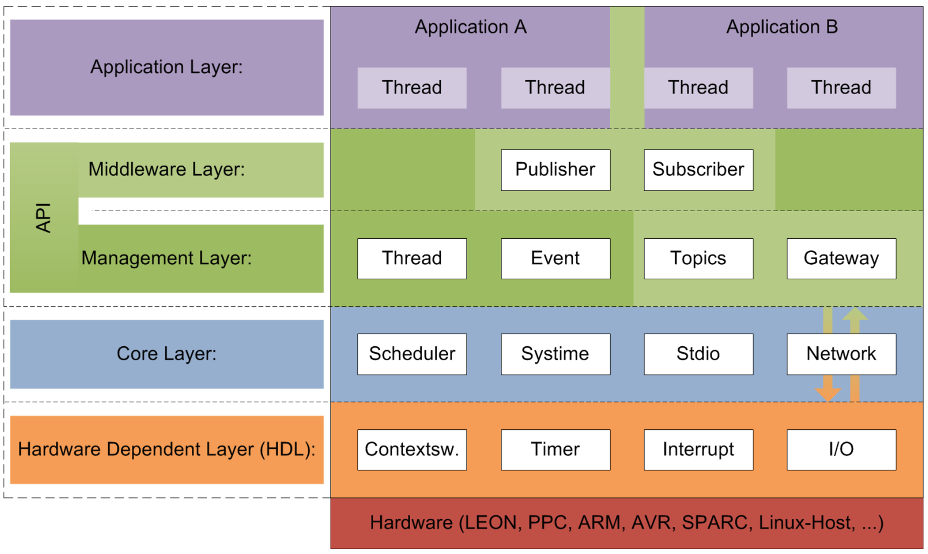


Figure 5.5: RODOS Layers [83]

inter-process communication between applications, but also to communicate with several RODOS computing nodes over the network (Ethernet Transmission Control Protocol/Internet Protocol (TCP/IP), Controller Area Network (CAN) communication) using gateways. A detailed design description of RODOS can be found in [57]. RODOS is OpenSource under the BSD-license and can be obtained through its SourceForge[14] or DLR website[15]. Examples for current space missions flying with RODOS or RODOS-derivatives are *TET* [22, 42], *BiROS* [71] and *TechnoSat* [5].

5.2.2 *Communication Mechanisms*

As described above in Section 5.2.1.2, RODOS offers a middleware which can connect not only S/W, but also devices and S/W together. It distributes messages locally and uses gateways globally. By using the publish/subscribe protocol, publishers can make messages public under a given topic. To establish a communication path, both the publisher and the subscriber must share the same topic which is represented by a topic ID and a data type. Comparable to H/W buses, the middleware implements an array of topics and each time a message is published under a given topic the middleware checks for all subscribers who wish to receive it. The identified subscribers will receive a copy of the message content. Leaving the boundaries of a single computing node, RODOS gateways may read all topics and forward them to a connected network. Further details regarding the RODOS middleware can be found e.g. in Montenegro [56].

14 `http://sourceforge.net/projects/rodos`
15 `http://www.dlr.de/rodos`

Beside this middleware-based high level communication, a number a low level techniques can be found within the wide range of available RTOSs on the market. Just to name the most important, there are

- shared data structures,
- semaphores,
- message queues,
- pipes, and
- mailboxes.

A detailed description of these mechanisms can be found e.g. in Simon [77, chap. 7].

5.2.3 *Embedded Software Development Process*

Within this section the S/W development process which is leading to the final boot image which runs on the embedded target H/W will be shortly described.

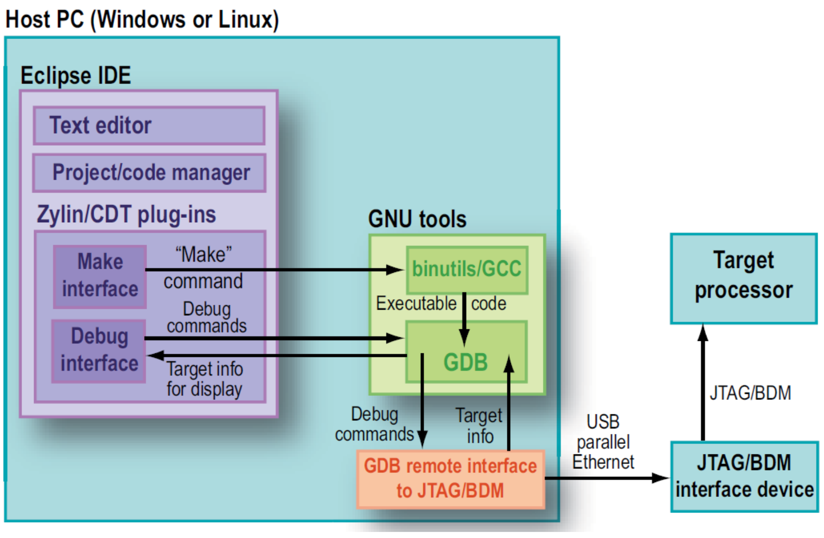


Figure 5.6: Eclipse-based Cross-Development System [33]

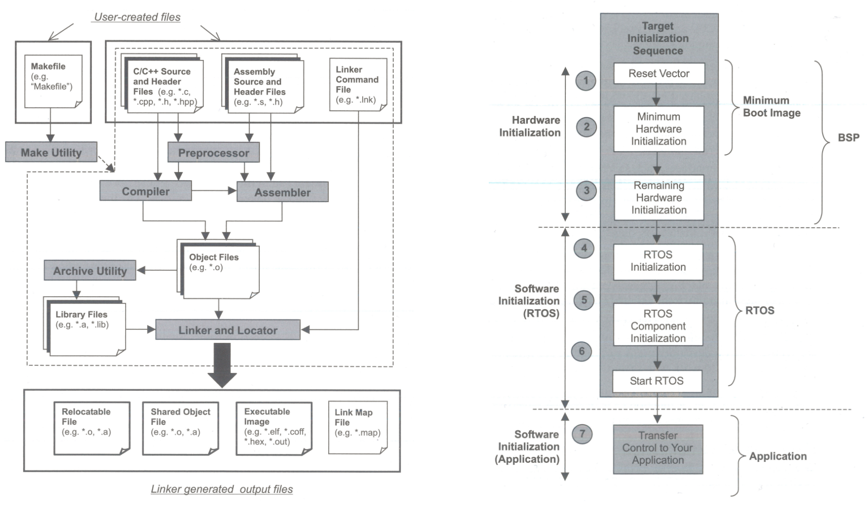
DEVELOPMENT ENVIRONMENT First of all the embedded S/W developer has to set up an appropriate S/W development environment. A number of dedicated tools exist nowadays, the challenge is to integrate them seamlessly in order to create a S/W tool chain which makes embedded development and debugging as easy as possible. Exemplary such an environment is depicted in Figure 5.6: the *Eclipse-*

platform[16] is used for integrating and accessing all relevant tools and especially for invoking the cross-compiler [77, sec. 9.1].

In order to speed up the embedded S/W development process especially in satellite projects in which most of the space-qualified H/W is often only available in later project-phases, the usage of *instruction set simulators* for substituting the real H/W is strongly advised [77, sec. 10.2]. These simulators are running on the host computer and are capable of simulating the microprocessor, memory and communication mechanisms.

It has to be mentioned here, that the final functional and performance testing has to be done on the target H/W anyway. Also during the various tests performed within this thesis it turned out that this is indispensable (cf. e.g. to Section 18.5.6.1 and the Footnote 88 herein).

BOOT IMAGE CREATION & INITIALIZATION ON H/W The process of compilation & linking of the final executable image of the embedded S/W is a well defined process. After having set up a development environment like described in the paragraph above and exemplarily shown in Figure 5.6, the steps depicted in Figure 5.7a must be passed through in order to create the image file to be transfered to the target H/W. Hereafter an initialization process like shown in Figure 5.7b is started in order to finally transfer the control to the application(s).


User-created files
Makefile (e.g. "Makefile")
C/C++ Source and Header Files (e.g. *.c, *.cpp, *.h, *.hpp)
Assembly Source and Header Files (e.g. *.s, *.h)
Linker Command File (e.g. *.lnk)
Make Utility
Preprocessor
Compiler
Assembler
Object Files (e.g. *.o)
Archive Utility
Library Files (e.g. *.a, *.lib)
Linker and Locator
Relocatable File (e.g. *.o, *.a)
Shared Object File (e.g. *.o, *.a)
Executable Image (e.g. *.elf, *.coff, *.hex, *.out)
Link Map File (e.g. *.map)
Linker generated output files
Target Initialization Sequence
Reset Vector
Minimum Hardware Initialization
Remaining Hardware Initialization
RTOS Initialization
RTOS Component Initialization
Start RTOS
Transfer Control to Your Application
Hardware Initialization
Software Initialization (RTOS)
Software Initialization (Application)
Minimum Boot Image
BSP
RTOS
Application
1
2
3
4
5
6
7

(a) Creation (b) Initialization

Figure 5.7: Image File for the Target System [43]

MEMORY MANAGEMENT Efficient memory management is of particular importance taking into account the limited resources of most embedded systems. Especially the correct usage of the *heap* and the

16 https://www.eclipse.org

stack areas of the Random Access Memory (RAM) are essential to the stability and reliability of the system.

In [4] the following definitions of heap and stack are given:

> *Heap*: An area of memory that is used for dynamic memory allocation. Calls to `malloc` and `free` and the C++ operators `new` and `delete` result in runtime manipulation of the heap.
>
> *Stack*: An area of memory that contains a last-in-first-out queue of storage for parameters, automatic variables, return addresses, and other information that must be maintained across function calls. In multitasking situations, each task generally has its own stack.

Stack and heap memory are allocated statically by the programmer by calculating the required space, trying to use as less memory as possible. This is a difficult task which requires a lot of effort in testing. An underestimation of the sizes has a significant impact on the behavior of the system and can lead to serious runtime errors. Therefore heap and stack memory consumption has to be kept constant during runtime, and in most coding standards for embedded systems the dynamic memory allocation *after* finishing the initialization process (see Figure 5.7b) is strictly forbidden. The efficient mastering of heap and stack in order to achieve a reliable and performant system is explained in detail in the article of Lundgren and Frimanson [44].

5.2.4 *Schedulability Analysis*

Throughout the whole S/W development process within a S/C mission scenario – as it is described in Chapter 6 and depicted in Figure 6.1 – the European Cooperation For Space Standardization (ECSS) standards require to monitor the Central Processing Unit (CPU) utilization as well as to perform a schedulability analysis [18]. Both – CPU utilization and schedulability analysis – are based on timing measurements of the embedded S/W which are initially estimated and later on measured on the running S/W. For an overview of the current state of the art within the field of schedulability analysis and the technologies used please refer to [36, sec. 4.3.11] or [65].

Within this thesis we will use various profiling techniques to measure the CPU utilization. For the Monitoring F/W prototype (cf. to Section 16) this is as a first-cut test which ensures that the S/W does not exceed the overall available computational power of the processor and stays within defined margins. For the Monitoring F/W flight version (cf. to Section 17) we will use the schedulability analysis for the final boot image of the satellite in order to verify that the timing requirements associated to the execution of tasks, data flow and events are met – hence proving that the real time characteristics of the embedded S/W.

6 LIFECYCLE OF A SPACECRAFT

The typical lifecycle of a S/C is divided into project phases which have to cover the whole space mission scenario. Herein the development of the S/C plays a major role, but important aspects like interaction with the groundstation, storage and analysis of mission data, etc. have to be planned with the same accuracy.

The phases are standardized due to the according ECSS document [17] where they are described in detail. A nice overview is also given in [19]. Figure 6.1 shows the timing relationships of the phases and the activities to be carried out during each phase. The following acronyms are used:

AR	Acceptance Review	PDR	Preliminary Design Review
CDR	Critical Design Review	PRR	Preliminary Requirements Review
FRR	Flight Readiness Review	QR	Qualification Review
MDR	Mission Definition Review	SRR	System Requirements Review
ORR	Operational Readiness Review	WBS	Work Breakdown Structure

The ECSS project phases can also be mapped to the standard V-model, which is widely used as a standard project management model [see e.g. 40, sec. 2.1.2]). This mapping is shown in Figure 6.2.

As the Monitoring F/W to be developed and described in the Sections 15ff. and shall be used during all project phases, in the following sections the main tasks concerning the project (in our case: the mission) as well as the product (in our case: the satellite) are listed. We will then use these tasks as an input for deriving the requirements of the Monitoring F/W (cf. to Chapter 15).

Model Philosophy

As the main operating range of the Monitoring F/W is the OBC of the satellite, the integration and assembly of the system has to be analyzed. For this we will have a look on the model philosophy and identify the typical and most common models to be developed in each phase [see e.g. 23, sec. 17.8]. Please refer to appendix Section A.2 for a graphical overview of the models to be developed within each level of design (breadboard, engineering and flight) and their integration into the Assembly, Integration and Verification (AIV) plan.

6.1 PHASE 0: MISSION ANALYSIS

Definition of the Mission

Within this starting phase the needs are identified and the intended mission is evaluated in terms of

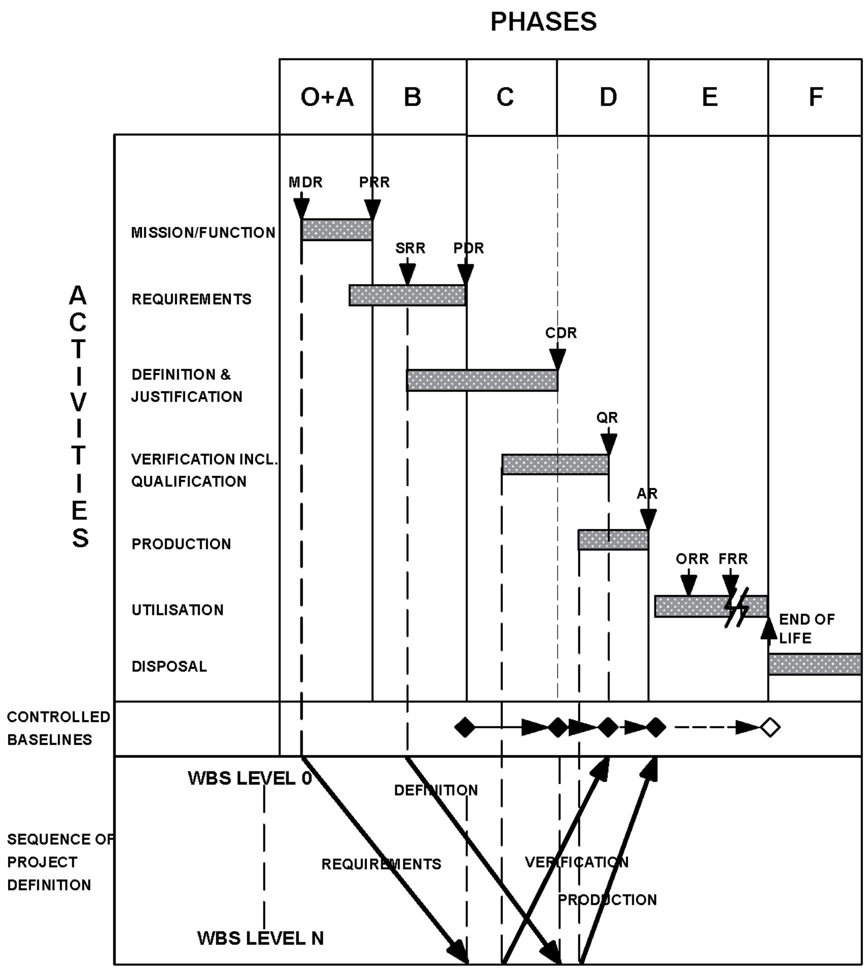


Figure 6.1: S/C project life cycle (due to [17])

- possible system concepts including compliant payload design solutions
- performance, dependability and safety goals
- operating constraints
- organization, costs and schedules

At the end of this phase, a first mission baseline is defined (keeping possible alternatives in mind), and a set of minimum system requirements are identified.

6.2 PHASE A: FEASIBILITY

Feasibility of the Mission

The needs expressed in Section 6.1 are finalized, and the process of responding to this needs is started. The different system concepts identified in phase 0 are analyzed in terms of

- critical elements, uncertainties and risks

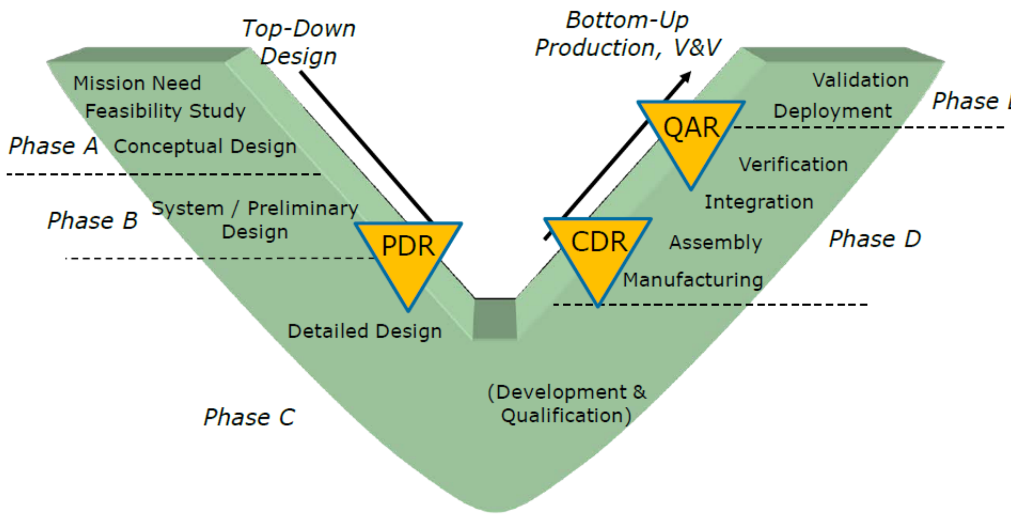


Figure 6.2: ECSS project phases mapped on the classical V-Model (picture from [24])

- technical and industrial feasibility
- constraints relating to costs, schedules, organization, utilization, production and disposal

Finally, a first concept of mission, payload and S/C design is at hand, including a corresponding function tree and requirements for the S/C, the orbit and trajectories.

6.3 PHASE B: PRELIMINARY DEFINITION

Definition of Requirements and preliminary Design

Within this phase a technical solution (=design) for the system concept picked out in phase A will be selected. The following tasks are to be carried out:

- design refinement and verification
- system and equipment specification with special emphasis on interfaces and budgets
- functional algorithm design and performance verification

The phase will be finalized with a consistent set of requirements, which are mapped to a preliminary design (S/C and mission).

6.4 PHASE C: DETAILED DEFINITION

Designed detailed and finalized and the Development starts

Within phase C the design of the system and its components is verified and finalized, first models like the Structure-Thermal Model (STM) and electronic breadboards are build up. Further tasks are:

- subcontracting of component manufacturers
- ordering of long-lead items (e.g. radiation-hardened processors and memory chips)
- start of EGSE development and test
- start of Onboard Software (OBSW) development and verification
- planning of test procedures on unit and subsystem level

Taking the preliminary design defined in phase B into account, H/W and S/W development starts on breadboard level. These breadboards will have the same basic functionality, but use different and non-space-qualified components. Typical models to be build are the STM and the Flat-Sat. Though other models will follow, these breadboards will be used during all mission phases, to used as – taking the Flat-Sat as an example – a testbed for S/W development.

6.5 PHASE D: PRODUCTION

Production, Assembly, Integration and Test

At the end of this phase the flight model has to be delivered. Therefore the following task have to be performed:

- subsystems production
- (sub-)system integration and test
- S/W verification
- validation of operational and functional performance
- finalization of flight procedures

The Engineering Model (EM) which development was already started in phase C is now finalized. Though functionally identical to the flight model, it is only used for functional and performance tests and verification purposes. Having successfully passed all tests, the Flight Model (FM) can be build on the basis of the EM, but now containing full redundancy and high-reliable and space-qualified parts.

6.6 PHASE E: OPERATION PHASE

Spacecraft Operations

The *operation*[17] phase plays a special role, as it is the main usage scenario of the Monitoring F/W. The operation phase E – starting after

17 In literature (cf. to [17], [19]) this phase is often called *utilization*, referring to *operation* as the main task to be undertaken during this phase. As the timeline in Figure 6.1 shows, task- and phase-duration are almost identical during this phase, therefore the name *operation* seems more appropriate to the author. It is also more common in a satellite mission which is the main space-scenario used within this thesis.

the Flight Acceptance Review (FAR) – can be broken down into further details, strongly depending on the type of mission. For the space segment the following generic mission phases[18] can be identified [see 2, sec. 5.1]: *system overview: mission phases and operations*):

- on-ground and pre-launch phase (ground segment validation, operator training, etc.)
- launch and early orbit phase (LEOP) or near-earth commissioning in case of interplanetary missions
- in case of interplanetary missions: cruise phase, including deep space maneuver (planet(s) fly-by(s), orbit corrections, . . .)
- commissioning phase (covering satellite in-orbit verification and payload commissioning (e.g. calibration and performance measurements))
- exploitation phase
- end-of-life disposal phase (focusing on controlled de-commissioning, de-orbiting, transfer to graveyard-orbit, etc. of the S/C)

The chronological order of these phases is also shown in the following Figure 6.3:

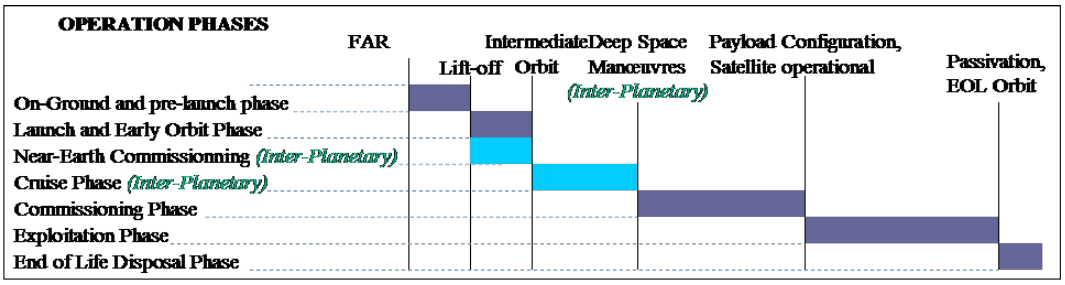

OPERATION PHASES
FAR
Lift-off
Intermediate Orbit
Deep Space Manoeuvres (Inter-Planetary)
Payload Configuration, Satellite operational
Passivation, EOL Orbit
On-Ground and pre-launch phase
Launch and Early Orbit Phase
Near-Earth Commissionning (Inter-Planetary)
Cruise Phase (Inter-Planetary)
Commissioning Phase
Exploitation Phase
End of Life Disposal Phase

Figure 6.3: Generic Operation Phases (from [2])

18 These phases are considered to be template phases, they have to be be refined for or tailored to each specific mission.

7 INVOLVED ACTORS

The identification of the involved actors has to be done in two steps:

1. getting an overview over the involved engineering domains, and

2. showing the temporal distribution of how the different domains are involved in the build process of a S/C (as depicted Figure 6.1).

The engineering domains which are involved in the S/Cs build process and therefore need access to the systems engineering data are depicted in Figure 7.1 [24]. There is a direct relationship between these domains and the actors(=engineers) working in them. Nevertheless it has to be mentioned that in practice (especially within smaller projects), it turns out that one engineer is working in different domains at the same time (e.g. by doing additional performance engineering while mainly being involved in software engineering).

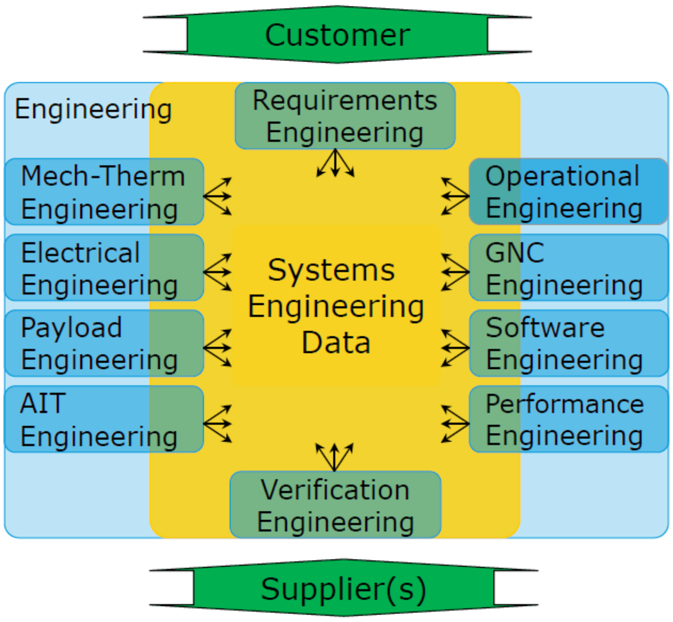


Figure 7.1: Data necessary for S/C Engineering and the involved Engineering Domains which are using the Data [24].

The degree of work load within each domain depends on the project phase. Figure 7.2 [19] tries to address this issue by showing the tempo-

19 The graphic is based on `http://commons.wikimedia.org/w/index.php?title=File:RUP_disciplines_greyscale_20060121.svg&oldid=108540496`

ral distribution of the domain's workload. Due to the reasons already described at the very beginning of Section 5.2.1 the payload engineering domain is not taken into account.

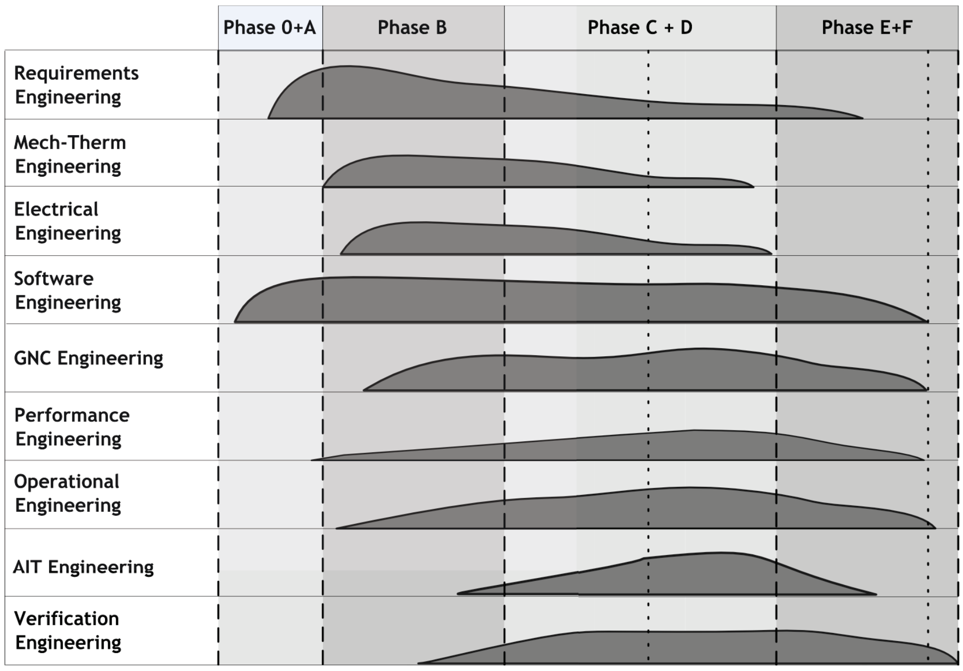


Figure 7.2: Temporal Distribution showing how the different Engineering Domains are involved in the Build Process of a S/C

Part III

SPACECRAFT INFORMATION TYPES

A detailed look into the various information types inside a spacecraft is provided within this part.

8 SOFTWARE ABSTRACTION LEVELS

The selection of a suitable monitoring tool (or a set of tools) from the techniques presented in Chapter 12 clearly depends on the type of objects to be monitored. Then again the type of monitored objects depend on the level of abstraction which the user is interested in. Within this chapter we will first give a short introduction to the different abstraction levels associated with execution of a S/W program within a S/C and then map them to the area of S/C avionics in the Chapter 9.

The information of interest differs in the level of detail. In order to reduce the information quantity and provide only the relevant information to the user, S/W execution can be monitored at different levels of abstraction [76, sec. 2.4.1]. At each level of abstraction the object to be monitored can be understood as a black box, hiding its implementation to the user.

8.1 SYSTEM LEVEL

Within this level only information is provided which is related to the *real-world* or *users view* of the system. Typically, the entities where the monitored information originate from are the subsystems of a satellite which directly influence the real-world. Given two examples, this could be the thermal-control or the power-control system. Within the S/W architecture these physical systems are often directly linked to their corresponding S/W applications which are responsible to control them (cf. to the corresponding Section 17.1.1: The Compact Satellite S/W Architecture). The status information to be monitored contains no details about the implementation of these subsystems.

8.2 PROCESS LEVEL

Having introduced the S/W applications as the steering entities of their subsystems in Section 8.1, the next step of doing a more in-depth monitoring would be to look at the information regarding the processes [20] of these applications. Typical process level events are

20 The terms *process* and *thread* are often used in parallel, but they actually mean different things: a process provides the resources needed to execute a program which is the reason why different processes have to run in separate memory spaces. With each process a single thread (the *primary thread*) is started, but additional threads can be created as well. Threads are OS specific features and represent the entities within a process which can be scheduled. Therefore threads can be understood as subset of a process. Threads belonging to the same process are running in a shared memory space. Further details about threads can be found e.g. in [54, chap. 4].

- process creation & termination,
- process state changes ("running", "waiting", "ready"),
- process synchronization,
- inter-process communication,
- external interrupts, and
- I/O operations.

8.3 FUNCTIONAL LEVEL

The Monitoring F/W will operate in the Functional Level

In terms of the monitoring, next step after having examined the process level would be to look into the details of process execution which will result in looking into the implementation details of the processes by monitoring the function calls, their return values and the parameters passed between functions. From within this level it is also possible to cover also the system and process levels above in which the user wants to gain insight into the S/W applications controlling their corresponding subsystems. The functional level is therefore chosen as the level in which the Monitoring F/W will operate in (cf. to Section 14.1).

Before runtime the monitoring task on this level is typically performed by doing step-by-step debugging inside the functions on the development host. However, *at runtime* the monitoring of *each* executed function is most of the time not possible since it uses very much CPU performance of the system being monitored, and – in addition – the collected amounts of traced events are too huge to be of any practical use.

A common solution to this problem at runtime is to insert output statements at dedicated locations inside the source code (e.g. in conditional branches or when the state of a process changes). These output statements can be either print statements (current State-of-the-Art) or the use of the logging statements of the Monitoring F/W (beyond the State-of-the-Art). Details about print statements are contained in Section 12.1.1, their disadvantages are outlined in Section 13. For the Monitoring F/W refer to Sections 16 and 17.

9 TYPES OF INFORMATION WORLDS

Figure 9.1 gives an overview of the different information worlds within a space system. For each information world different levels of abstraction are needed, some of them are overlapping each other. Within this section we will describe these information worlds, their monitoring needs and the levels of abstraction needed. The idea of this thesis is to combine all these worlds into one single information world, therefore concentrating on a level of abstraction from which it is possible to cover the monitoring demands for all of them.

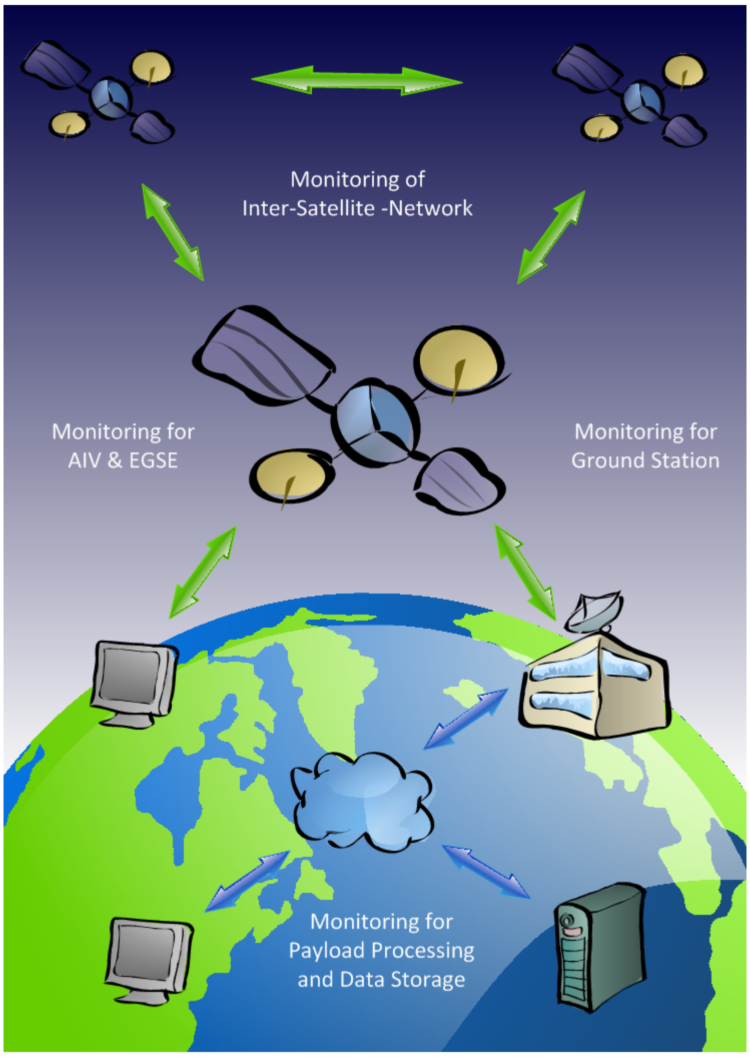


Figure 9.1: Different Information Worlds within a Space System

9.1 MONITORING OF INTER-SATELLITE-NETWORKS

Satellite networks are based on having several distributed S/Cs which together fulfill one common mission goal. The traditional way to distribute S/Cs in orbit is called *satellite formation flying* and can be subdivided into the following categories:

- cluster formations,
- trailing formations [71], and
- satellite constellations (like e.g. the GPS).

While in all of the above categories rather similar S/Cs are distributed in space, the new idea of *Fractionated S/Cs* is based on placing several different satellites into orbit, spreading the traditional satellite subsystem functionalities of one satellite over several satellites. The handling of all of these distributed S/Cs is still an area of active research[21][32].

In terms of monitoring, a typical satellite formation consisting of technically similar satellites does not differ much from the monitoring of a single satellite – every satellite has to be monitored on its own. The major difference is that typical S/C information like telemetry has to be assigned also to the S/C it is coming from (e.g. by using the S/C identifier (ID) as an additional parameter). For a fractionated architecture where the whole satellite swarm appears and acts as one virtual satellite, the monitoring may concentrate on one special satellite which is acting as the communication interface to the groundstation, collecting and downloading information from his accompanying satellites as well.

The abstraction levels needed for the monitoring of inter-satellite networks are the same as presented in the upcoming Section 9.3.

9.2 MONITORING FOR AIV AND EGSE

The monitoring for AIV and EGSE concentrates on the information needed by system engineers during the final steps of the development process, resulting in the end-to-end testing of the satellite on ground. For this mainly monitoring on the system level of abstraction is needed, sometimes going down to monitoring on process level as well (e.g. when a concentration on I/O operations and external interrupts is needed). Only for debugging purposes in case of a faulty behavior of the system a deeper insight into the functional level may be necessary. Given an example, the reception of a command to activate the heaters of the satellite's thermal control systems would be considered as a typical system-level event.

21 One research area is the communication between the S/Cs through inter-satellite networks (e.g. finding a suitable network topology) [86]

Figure 9.2 shows where to bring in the system-level probes to monitor these events, using another example of a typical physical S/C avionics implementation from [3] (which is to some degree more detailed than the overview presented in the Hardware Section 5.1).

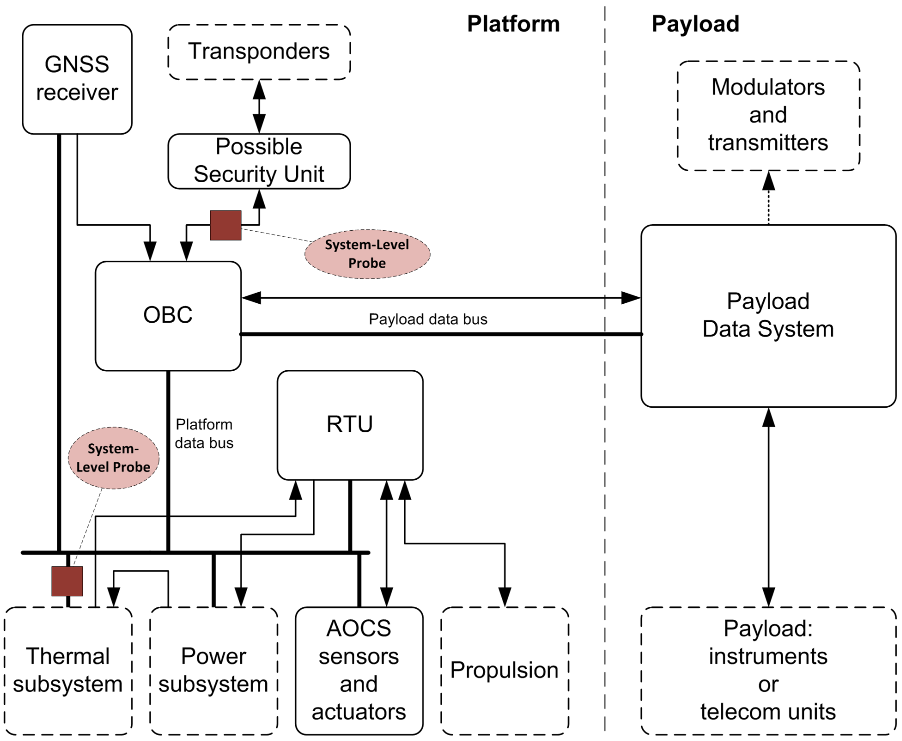


Figure 9.2: System-Level Probe insertion in a typical S/C avionics implementation (here: Probes for the Thermal Control System) [3]

For our example these probes can be the listening to the acknowledge of a received and executed TC (e.g. "turn heaters on") and the observation of a certain relay-switch (e. g. "heaters were switched on") afterwards.

9.3 MONITORING FOR GROUND STATIONS

Monitoring for groundstations is typically done by collecting information on board the S/C and sending this information to the groundstation as TM. The information to be provided usually resides in the system level of abstraction (cf. to Section 8.1) and can be divided into two main categories [87]: housekeeping and (science) data (the latter is explained in Section 9.4). Housekeeping data contains information about the S/C's health- and safety-state.

In this section the C&DH system to be monitored is illustrated exemplarily for a small satellite which is representative for other S/Cs as well. As depicted in Figure 9.3, housekeeping data is collected from all S/C devices and applications (e.g. for thermal control, the Attitude Determination and Orbit Control System (AOCS), or the Electrical Power System (EPS)) upon request by a S/W application normally called *Housekeeper*. The Housekeeper provides the collected data in a frequent manner to the corresponding TM application which transfers them to the groundstation as historical or extended TM. The housekeeping data which is not already transferred to ground is stored within the mass memory of the OBC until the next contact – typically within in a ring-buffer structure where the oldest TM is overwritten after some time. Depending on the criticality, parts of the information is may be also sent using the S/C's real-time TM.

Gathering housekeeping information is most important to ensure a successful mission. On board the satellite the housekeeping data is needed by the surveillance application to trigger FDIR activities. On ground the information is used by mission engineers to check that everything works correctly and by the science staff for doing additional analysis, like proving the quality of science data, instrument and device performance.

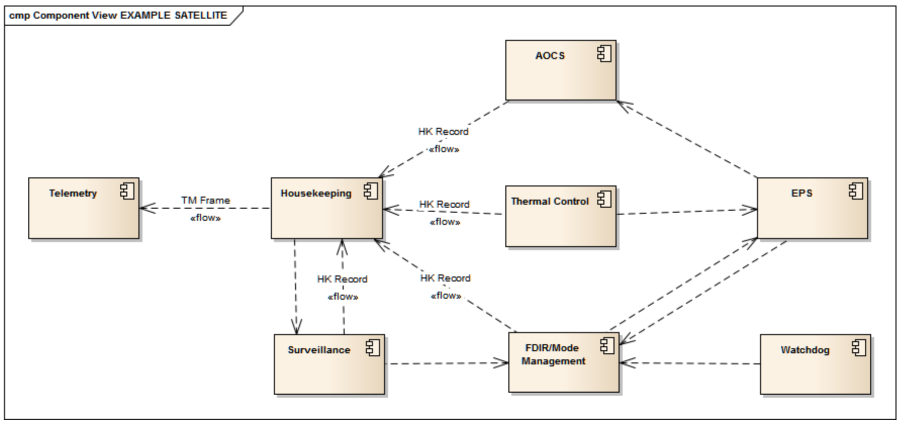


Figure 9.3: Housekeeping capability of the onboard S/W (example; simplified view)

To transfer as much housekeeping data as possible special emphasis has to be put on optimization techniques for the transfer via the radio link, because typically the downlink capacity has to be shared among the S/C platform (satellite bus) and payload TM (cf. to Section 5.1). Responsible for the handling of housekeeping data is a special PUS service called "Housekeeping and Diagnostic Data Reporting Service" (no. 3 [20]). This service also allows to configure the sampling rates of the housekeeping data to be collected.

Beneath the housekeeping capability, extensive debug information is needed in order to do a thorough monitoring of the S/C. This is partially achieved by two PUS services [20]:

- PUS Service no. 8 ("Function Management Service") is a dedicated service to provide limited access to the functional abstraction level of the S/C's onboard S/W during operation. The service offers the possibility to do some sort of remote method

invocation using a function ID and the corresponding parameters.
It has to be mentioned that e.g. print statements (cf. to Section 12.1.1) are typically encapsulated within other functions, therefore they cannot be called directly which makes these kinds of methods unreachable for this PUS service.

- PUS Service no. 12 ("Onboard Monitoring Service") offers the possibility to monitor a list of parameters on board the S/C and the possibility to downlink an event report (e.g. in case a parameter exceeds its pre-defined range)

Non of the above mentioned services targets the collection of low-level debugging information on the functional level which is needed for AIV/Assembly, Integration and Test (AIT) purposes for doing in-depth debugging of the internal state and control flow of the S/C's boot image. Currently there is also no other widespread technology available which would provide this information on ground during the operational scenario. In addition, as the debug statements within the S/W source code are not used during operation they are often removed before the launch. Consequently, the observability into the data handling system is extremely reduced during the operational phase of the mission.

9.4 MONITORING FOR PAYLOAD PROCESSING AND DATA STORAGE

Also for the monitoring of the payload instruments the system level of abstraction is needed. Like described in Section 5.1, if the control of the payload is overtaken by the OBC of the satellite bus the same techniques which are needed by the groundstation in order to operate and monitor the S/C are used here (see section above). Likewise, if the S/C is designed as such, that there is a separate data handling unit responsible for payload control similar techniques are used as the information needed for the monitoring of the payload computer are usually akin to the monitoring of the data handling unit of the satellite bus.

As the operational scenarios and their monitoring needs of most missions are somehow similar to each other, this differs when it comes to storage and processing of the science data received from the payload. For this area of monitoring located in the lower part in Figure 9.1 we will have an exemplary view on the setup of the science ground segment of the *Planck*-project (cf. [46]). A detailed description of this setup is given in the article of Texier [79] which also includes the following figures: Figure 9.4 gives on overview of all elements of the ground segment, in particular it shows how the science ground segment is connected to the ground station of the operations ground segment. More details about this connection and its interfaces used during operations are then contained in Figure 9.5.

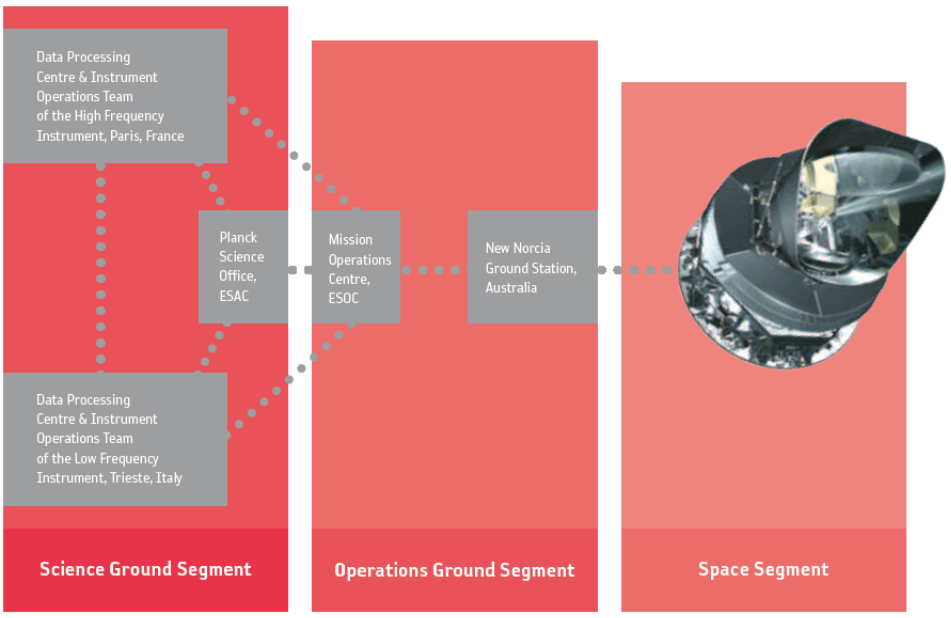

Data Processing Centre & Instrument Operations Team of the High Frequency Instrument, Paris, France
Planck Science Office, ESAC
Data Processing Centre & Instrument Operations Team of the Low Frequency Instrument, Trieste, Italy
Mission Operations Centre, ESOC
New Norcia Ground Station, Australia
Science Ground Segment
Operations Ground Segment
Space Segment

Figure 9.4: Elements of the *Planck* Ground Segment (from [79])

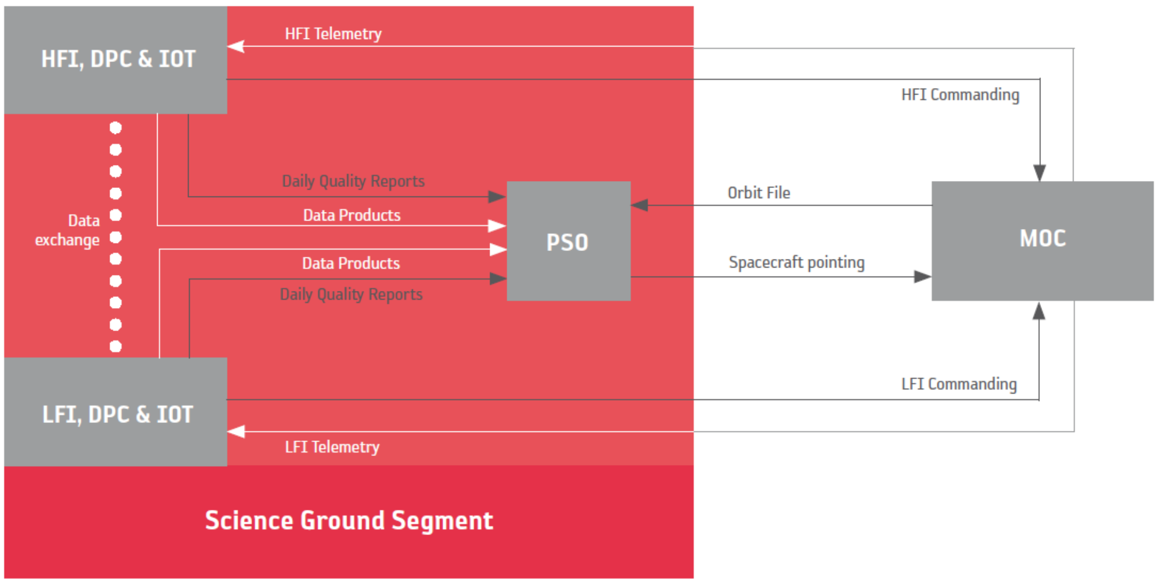


Figure 9.5: Interfaces in the *Planck* Science Ground Segment during Operations (from [79])

10 TYPES OF INFORMATION

Within this chapter we will have a detailed look on the different types of S/C information which is available during build, integration, test and operation. This information can be categorized into information which is holding the state of the system, information to be exchanged system-internal and the TCs to be exchanged with the groundstation.

10.1 INFORMATION HOLDING THE STATE OF THE SATELLITE

The Table A.1 lists some exemplary telemetry values of the TET-1 satellite [61]. These values are chosen by the system team to be sent to the ground station as real-time TM (cf. to Section 10.2). Together with the corresponding analogue values in Table A.2 they reflect the internal state of satellite in enough detail for the system engineers to decide whether everything is running fine in order to fulfill the mission goals, or – in case of a failure – if some kind of intervention from ground is needed.

10.2 INFORMATION TO BE EXCHANGED ON THE SOFTWARE BUS

During the operation of a satellite, the communication system of the onboard computer can be characterized by a few central messages, which are present in most missions in one or other form:

COMMANDS to be distributed to all applications (tasks) running on top of the OBC's RTOS

STANDARD HOUSEKEEPING REQUESTS to collect housekeeping data from all applications

"I AM ALIVE"-MESSAGES to check if vital applications are still running

ANOMALY REPORTS from applications to report anomalies and error which they have detected

REAL-TIME TELEMETRY from applications to provide extended housekeeping which will be sent immediately

HISTORICAL TELEMETRY from applications to provide extended housekeeping data to history which will be sent upon request

These messages will normally frequent the S/W bus[22] of a satellite, making use of the implemented S/W communication structures, for example by using a middleware (cf. to Section 5.2.1).

COMMANDS A typical satellite application like thermal control receives commands from the TM/TC-application which are normally coming directly from the ground station. It is also possible that some commands are coming from other applications. Here are two examples:

- An application which is responsible for handling lists of commands to be executed at a certain point in the future. These lists are sent by the ground station, but afterwards managed onboard of the satellite using internal lists. These lists can be edited (e.g. insert or delete commands) by special TCs. If their time of execution has arrived the commands will be distributed to the corresponding applications.
- An application which is responsible for providing autonomy e.g. in case of loss of control from the groundstation. It is responsible for executing predefined lists of TCs until the connection to the satellite is set up again[23].

STANDARD HOUSEKEEPING REQUESTS The application which is responsible for collecting housekeeping information from all other applications. This is typically done by issuing housekeeping requests, asking the other tasks to provide information about their internal state.

"I AM ALIVE"-MESSAGES All applications shall send periodically messages in order to inform the application which is responsible for supervising the health status of the satellite, that they are still vital (running and able to react). The supervising application of often referred to as S/W-*Watchdog*, whereas the messages themselves are called *I Am Alive* messages. If any application is not publishing these messages anymore, the watchdog assumes some kind of crash and executes a pre-defined action (like bringing the satellite into safe mode, do a switchover to a redundant computer or initiate a restart).

ANOMALY REPORTS In contrast to the above mentioned "I Am Alive" messages where each application is "asked" for reporting its

22 The term *software bus* originates from its similarities to bus systems connecting H/W components to each other which are characterized by their shared communication channels.

23 For deep space mission this may be the default communication scenario, as commanding by ground is not practiced because of the long time it takes for a command to reach the system. For this reason an interplanetary probe has to have a high degree of autonomy.

status, the application itself has also the possibility to report a critical situation (an anomaly or an error) [24]. One of the receivers of these messages is the Housekeeper, another receiving application will be some kind of FDIR component being responsible for initiating appropriate actions in order to prevent a crash of the whole system (going into safe mode would be appropriate in most cases).

REAL-TIME & HISTORICAL TELEMETRY All S/W components can (but are not forced to) provide extended telemetry in addition to the normal housekeeping information they have to provide. This kind of information can either be published as real-time or historical TM. The difference between these two is, that in case of contact with the ground station the real-time telemetry will be forwarded immediately to the earth, whereas historical telemetry will only be sent if requested from the groundstation (e.g. in the case of failure analysis after a crash).

10.3 INFORMATION TO BE EXCHANGED WITH THE GROUNDSTATION

The information to be exchanged between the S/C and the groundstation can be extracted from what is listed in the preceding Sections 10.1 and 10.2: while most of the information holding the internal state of the satellite will be committed during a downlink connection to earth as real-time or historical telemetry, an important reason for establishing an uplink connection to a satellite is the sending of TCs.

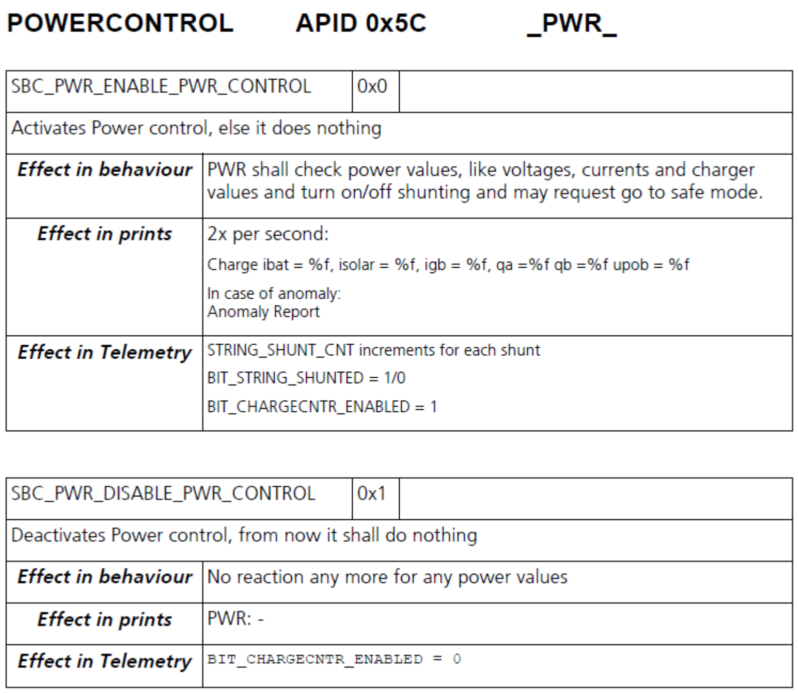
Using again the TET-1 satellite as an example, we will forbear to list all available TCs from this mission. In general it can be stated that the majority of the TCs have the purpose to either initiate a certain behavior of the satellite (e.g. go to safe mode), to change the binary TM in Table A.1[25], or to set the minimum and maximum ranges (e.g. battery loads or temperatures) needed by the applications in order to fulfill their steering and control tasks. The verification of the correct reception and successful execution on board the satellite can be done looking into the TM which was received after the TCs were committed[26]. Taken the TET `PowerControl` application as an example, the table in Figure 10.1 shows the necessary commands for activation and deactivation of the application, whereas the table in Figure 10.2 is showing the TC for changing the minimum and maximum values the application needs in order to trigger the resulting actions when the

24 The application ideally does this right before it eventually crashes where only the absence of the "I Am Alive" message is signaling that something went wrong.

25 The binary TM in Table A.1 can be recognized by the preceding `BIT_` (like `BIT_-TEMPCNTR_ENABLED`) and can only hold the values 0 or 1.

26 If the project adheres to some kind of standard like for example ECSS, there are special protocols (e.g. the CCSDS TM/TC protocols) and services (like e.g. the PUS service [20]) which have to be used for this verification process.

values are exceeded. Besides the command name, its parameters and a short description of its functionality the tables show the expected effects in the behavior of the corresponding subsystem, the console printouts and the TM to be received after the correct execution of the command.



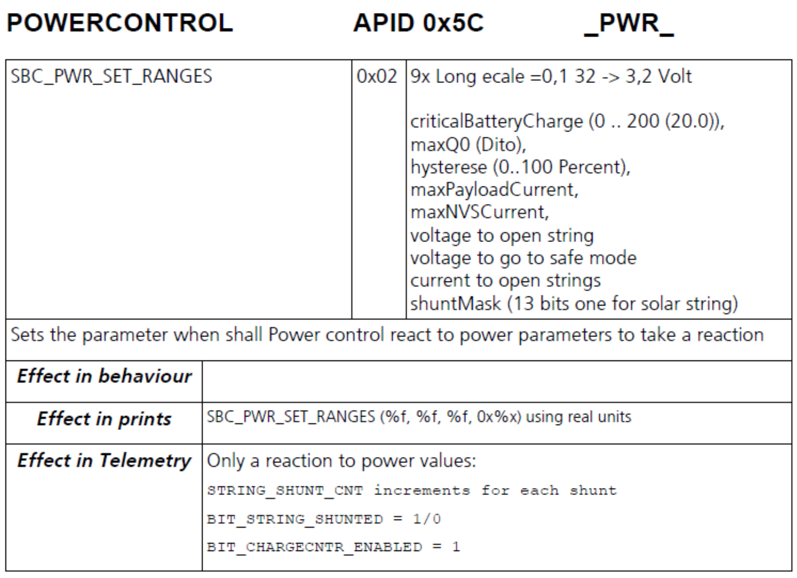
POWERCONTROL **APID 0x5C** **_PWR_**

SBC_PWR_ENABLE_PWR_CONTROL	0x0	
Activates Power control, else it does nothing		
Effect in behaviour	PWR shall check power values, like voltages, currents and charger values and turn on/off shunting and may request go to safe mode.	
Effect in prints	2x per second: Charge ibat = %f, isolar = %f, igb = %f, qa =%f qb =%f upob = %f In case of anomaly: Anomaly Report	

Effect in Telemetry	STRING_SHUNT_CNT increments for each shunt
	BIT_STRING_SHUNTED = 1/0
	BIT_CHARGECNTR_ENABLED = 1	

SBC_PWR_DISABLE_PWR_CONTROL	0x1	
Deactivates Power control, from now it shall do nothing		
Effect in behaviour	No reaction any more for any power values	
Effect in prints	PWR: -	
Effect in Telemetry	BIT_CHARGECNTR_ENABLED = 0	

Figure 10.1: TET Telecommands for enabling and disabling the `PowerControl` application



POWERCONTROL **APID 0x5C** **_PWR_**

SBC_PWR_SET_RANGES	0x02	9x Long ecale =0,1 32 -> 3,2 Volt criticalBatteryCharge (0 .. 200 (20.0)), maxQ0 (Dito), hysterese (0..100 Percent), maxPayloadCurrent, maxNVSCurrent, voltage to open string voltage to go to safe mode current to open strings shuntMask (13 bits one for solar string)
Sets the parameter when shall Power control react to power parameters to take a reaction		
Effect in behaviour		
Effect in prints	SBC_PWR_SET_RANGES (%f, %f, %f, 0x%x) using real units	

Effect in Telemetry	Only a reaction to power values:
	`STRING_SHUNT_CNT increments for each shunt`
	`BIT_STRING_SHUNTED = 1/0`
	`BIT_CHARGECNTR_ENABLED = 1`	

Figure 10.2: TET Telecommand for setting the ranges of `PowerControl` application

Part IV

MONITORING SPACECRAFT AVIONICS

This part gives an overview of the different techniques of monitoring an embedded system inside a spacecraft.

*The argument that system monitoring is just a nice to have,
and not really a core requirement for operational readiness,
dissipates quickly when a critical application goes down with no warning.*

— Larry Dragich

11 MONITORING TERMINOLOGY

In this section, the basic terminology of monitoring is introduced. The term *monitoring* is often used in parallel with other terms, mostly the term *logging*. But logging is only one technique of monitoring a system. Monitoring a technical system has a much wider meaning, and there is no common definition of this term. Therefore, within this section we will explain our understanding of monitoring with respect to the area of S/C construction and operation.

Recording information, observing a system and doing some kind of surveillance – all these tasks belong to the monitoring terminology. They are applied on a recurrent basis in order to make assumptions about the past, present and future of the system. One of the biggest motivations to put emphasis in the monitoring process is to recognize errors in the running system as early as possible. As it is shown in Figure 11.1, an error can be caused by a fault or by a critical state during operation [1]. An example for a fault can be a defect of a component or a S/W bug which can either still be dormant in the system or have already become active and leading to consecutive errors. A critical state can be a low battery power or a temperature which rises too high, both leading to a faulty behavior of the subsystem they occur in. As it is impossible to eliminate all faults within a system [55] an important goal is to detect an error before it leads to a failure of the whole component/subsystem and propagates to a higher level of abstraction (cf. to Section 8.1), causing subsequent errors which may lead to the failure of the complete system.

In order to achieve this goal, for the embedded parts of the Monitoring F/W to be developed as the foundation for the new idea of Unified Monitoring (cf. to the next Parts v and vi) it is important to be included in the lowest abstraction level possible. Only from here it is possible to monitor higher abstraction levels as well and the F/W can be used to do an in-depth analysis of the running system in case of an error as early as possible.

Of course it is even better to detect errors and remove the faults in the phase of building and integration, before the system goes into normal operation mode. This process – which is normally described by the verb *debugging* – and can also be tremendously improved by

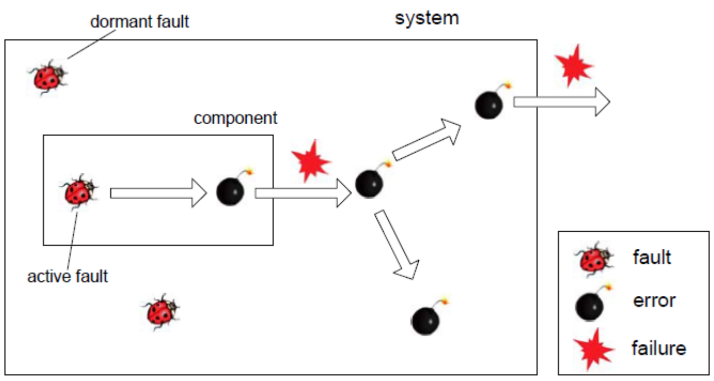


Figure 11.1: Error Propagation in an Embedded System [1]

establishing monitoring capabilities as early as possible in the development process.

Monitoring is especially demanding in the case of an embedded system like a satellite, where not only the "normal" housekeeping data has to be monitored, but also things which are not meant to be visible during normal operation, like the state of variables, which lines of code are actually being executed, or whether certain assertions hold across a complicated data structure [70].

Therefore, from what has been explained so far we derive our definition of monitoring which is to get to know the current system state – and what has led it – as good as possible, whereas *state* means

Detailed knowledge of state of the system at any point in time would be the ideal monitoring.

- the internals (e.g. variable values), and
- the execution flow within the system which led to this internals,

and *good* means

- to have an adequate time resolution, and
- to be as detailed (fine granular) as possible.

When we talk about the state of a system, formally we would have to distinguish between the state of the different H/W components on the one hand, and the state of the embedded S/W on the other hand. For direct H/W debugging purposes there exist a number of proven and well-understood techniques like volt meters, ohm meters, oscilloscopes, logic analyzers and in-circuit emulators (cf. to [77, sec. 10.4]). But as already described in the MOTIVATION Chapter at the beginning of this thesis, getting visibility into H/W gets more and more difficult. And even if we put a lot of effort into developing new and more sophisticated H/W debugging tools, this would only help us to better understand and build these systems *on ground*: as all of these techniques require direct access to the H/W, obviously they cannot be used in the remote operational scenario of a S/C. Looking into the

H/W State is reflected in the S/W State

state of the S/W is the only possibility to get to know the H/W state during runtime (operation) of the satellite.
Therefore we concentrate on the goal to equip the S/C to be developed with methods (sophisticated sensors, intelligent devices, etc.) to map the state of the H/W as good as possible to corresponding S/W entities. As a result, the running S/W image of a satellite contains both, the H/W state as well as the S/W state, together reflecting the state of the whole system.

Due to [70], the common ways of looking into the innards of an executing S/W program can be categorized as follows:

USING A DEBUGGING TOOL This is a non-intrusive technique and reveals the internal status of the system (values of variables etc.). Examples are tools like GRMON [25] or GDB [33]. They will be further explained in Section 12.2.1.

PRINTLINING This technique is highly intrusive as it makes temporary modifications to the program, typically adding lines that print information out. It is meant to reveal the execution flow and – in addition – also the internal status. An example is printf-debugging which will be explained in Section 12.1.1.

LOGGING This technique is about creating a permanent window into the programs execution in the form of a log. The program has to be written in a way that it can produce a configurable output log describing its execution and – in addition – also the internal status. Examples are real-time or historical TM. This intrusive technique will be explained in Section 12.1.2.

Together, these ways form a rich set of monitoring techniques which should ideally be combined into one tool which is known as the *system monitor*.

11.1 THE SYSTEM MONITOR

The system monitor is a specific process in a system mainly used for collecting and storing state data, but can be used for other purposes as well [27]. The design of the system monitor is driven the following aspects:

First, the system monitor should fulfill its task in a way, that the functional and timing behavior of the system is least influenced. This is especially important for embedded systems like satellites. Ideally, each application shall contain a tiny fraction of the monitor, being supplemented by a central monitoring instance. This makes the overall monitoring component highly configurable (e.g. regarding the level of detail), while keeping the communication overhead (e.g. by sending around monitoring messages within the system) at a minimum level.

In addition, the design of the the system monitor shall be as such that it is most performant in terms of

TIMING: monitoring information should be recorded within a desired period as well as recording frequency. To achieve this, the monitoring can be implemented using different modes: "monitor poll", "agent push" or a hybrid mode.

CONFIGURATION: even during operation the system should be configurable.

DATA STORAGE: a maximum amount of monitoring data shall be stored (maybe with compression) to be able to analyze the system as detailed as possible.

DATA ACCESS: monitoring data shall be accessed via all normal communication paths: e.g. UART and TCP/IP during integration and the satellite antennas for download to earth.

PROTOCOL: the system monitor should be most flexible in terms of communication: system-internally e.g. method invocation and middleware shall be supported, whereas the downlink to earth should take place using CCSDS-protocols.

Further requirements and details about the design and implementation of the Monitoring F/W as a realization of a system monitor for space applications like satellites can be found in the Chapters 15, 16 and 17.

12 KINDS OF MONITORING

Within this chapter the different kinds of monitoring information existing in the S/C domain will be addressed. Even though in the embedded world there exist several ways of monitoring a system, the space environment has special requirements that have to be taken into account when designing and implementing the monitoring techniques the system will provide. Especially

- the remote operational scenario, and
- the real-time requirements

are restriction criteria for the selection of appropriate techniques. Just to give an example, the usage of step-by-step debugging e.g. with GDB is not an option because

- it is technically not possible during the mission, and
- during monitoring on ground it would violate the realtime constraints of the system as the timing of the boot image would not reflect the behavior of the system during operation [27].

In order to avoid any possibility of confusion, a word about testing and operating of an embedded system inside a S/C – in comparison to the term *monitoring* of this kind of system – shall be mentioned: *Testing* of a S/C mainly concentrates on the validation and verification, that the S/C to be developed will operate within in the limiting values it is designed for. In contrast, *operating* a S/C has the goal to ensure, that system does not leave these limits during operational scenario. Both – testing as well as operating – need to utilize the *monitoring* capabilities from the S/C which provides them with as much and as detailed information about the state of the S/C as it is needed in order to fulfill their tasks.

Testing, Operating & Monitoring

Therefore, if we want to design a common toolset for realizing our vision of Unified Monitoring (see Chapter 14), we have to take into account and thoroughly analyze the tools and processes used to gather the S/C's status in both fields: testing as well as operating. This is the motivation for and the content of the following chapters.

27 It should be noted that there are approaches to do step-by-step debugging in RTOSs. The approaches are mostly based on the concept of working with different times: the system time itself and the time on the local debugging machine. Taking both times into account, it is possible to reconstruct a real-time scenario like acting natively on the H/W platform, on which the debugging should take place. But, this is still an active research topic and not used yet. Please cf. to [60] for one of the research papers regarding this topic.

12.1 MONITORING WITH SOURCE CODE INSTRUMENTATION

12.1.1 *printf-Debugging*

There exist several techniques to debug the program execution within an embedded system. A very simple and widely used one is to include print-statements in the source code of a S/W program. Debugging with print-statements is often referred to as *printf-debugging* [15]. The name arises from the printf statement in the C programming language, but similar statements can be found in all major high-level programming languages used for embedded systems. Also within RODOS the internal debugging statement is named with a capitalized `PRINTF`.

The main intention behind the usage of these kind of statements is to track the control and data flow during the execution of the code. This is done by adding printf-statements directly in the code of the applications. For this purpose RODOS offers methods like `PRINTF` and `ERROR`. As shown in Picture 12.1, these methods can be used by application developers by simply including `debug.h` in their application. In this typical debugging scenario, the developer starts a console on his host computer, establishes a serial connection to the target platform (here: the OBC of a satellite) and is afterwards able to watch the `PRINTF`-output on the terminal using a standard UART-interface.

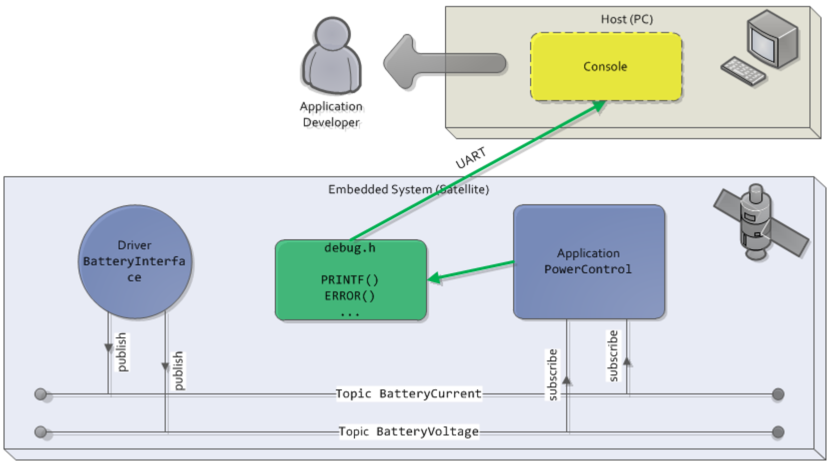


Figure 12.1: RODOS debugging using `PRINTF`-statements

Referring to Figure 5.5, there are two kinds of users who are using the debugging-technique with print-statements: first, there is the RODOS user who wants to use the RTOS in order to operate the OBC of his specific mission. He is only working in the application layer, dealing with the RODOS API from the middleware- and management-layers. Debugging with `PRINTF` will help him finding errors during the time of developing and testing his applications and will finally end up with an error-free boot image for the onboard computer. Sec-

Listing 12.1: TET-Code-Example using PRINTF statements

```
 1 bool ApplicationInterfaceTimeControl::executeCommand(Command &cmd
       ) {
 2     ...
 3     case TelecommandCodes::SBC_TIM_SET_UTC:
 4         PRINTF("TIM:       -> SBC_TIM_SET_UTC (%ld seconds, %ld
               milliseconds)\n",
 5                paramSeconds,
 6                paramMillisecs);
 7         sysTime.setUTC(paramSeconds*SECONDS
 8                        + (paramMillisecs % 1000));
 9         return true;
10     ...
11     default:
12         PRINTF("TIM!       -> bad telecomand code\n");
13         errorCounters.timBadTcc++;
14         return false;
15     }
16     return false;
17 }
```

ond, the RODOS developer itself makes extensive usage of the PRINTF-debugging capabilities while working on all layers that are located underneath the application layer.

In Listing 12.1 a usage example from the TET-1 mission is depicted: the method executeCommand of the TimeControl-application interface is called in order to synchronize the onboard time of the satellite with the time on ground. The corresponding telecommand is SBC_-TIM_SET_UTC. Within this method the PRINTF-function is used to display the correct setting of the onboard time (Listing 12.1, Line 4) or – in case of a not-recognized TC – an error taken place (Listing 12.1, Line 12). The output of these PRINTF-statement are usually displayed on the computer terminal of an S/W or AIV/AIT-engineer, intermixed with the output of all other S/W applications and/or threads running at the same time on top of the used RTOS.

Real-World-Example from TET-1 Satellite

How confusing such an output is can easily be seen when looking at the short terminal output snippet from one of the TET satellite integration sessions, shown in Listing 12.2.

In Line 1 and Line 43 the current TET-1 onboard time before and after setting the Coordinated Universal Time (UTC) time is being printed by the surveillance-application. The output of the PRINTF-method from Listing 12.1, Line 4, can be seen in Line 37. It is prefixed by the TIM-abbreviation, which stands for TimeControl.

In order to get rid of the PRINTF-messages when producing the final boot image or doing performance tests on the target H/W, these code fragments are often surrounded by macros. During compilation the

Surrounding printf Statements with Pre-Processor Macros

Listing 12.2: TET-1 PRINTF Terminal Output

```
SUR: SW-Ver=1671 @ 885.008000 Utc= 885.008 == 1.1.2000
    0:14:45.008
GIF: -
OP: no commanded OOP elems
ONS OP:r_WGS= 1013669.69892, 253474.92344, -6868032.61369
ONS OP:y_OOP= 995207.52008, 318335.96608, -6868032.61369
ONS: CIF
NVS: => cmd(0400:128.131) ok
TIM: Sync 886,0 @1PPS 0.000
HKE: send dmablock 18, writeindex = 2204, in block 36
HKE: + VC0
DOW: dma(0x145399, nr 18) firstUtc = 883
PWR: -
RED: Mode=2 Node=0 IWorker=1 partnerPwr=1, not-resp=0
RED: Sync
MMM: -
NVS: Msgs=0, hk=0
NVS: in-msg-len = 0
CMD: safelst=a051200
HKE: -
NVS: => cmd(0400:128.132) ok
WAT: 8 threads
RLY: -
RLY: cmdDec 5f c5 13 c0 30 00 00 86 19 06
SUR: read-analogs
DOW: wait for DMA
GIF: -
OP: no commanded OOP elems
ONS OP:r_WGS= 1012835.06067, 249758.01008, -6868292.06953
ONS OP:y_OOP= 994602.91758, 314609.21800, -6868292.06953
CCS: CLCW=01020000
CMD: ok-modem 0 1
MMM: -
NVS: Msgs=0, hk=0
NVS: in-msg-len = 0
CMD: safelst=a051200
CCS: cmd 0x23 apid 0x60
TIM: -> SBC_TIM_SET_UTC (340735380 seconds, 0 milliseconds)
HKE: -
NVS: => cmd(0400:128.132) ok
RLY: -
RLY: cmdDec 5f c5 13 c0 30 00 00 86 19 06
SUR: read-analogs
SUR: SW-Ver=1671 @ 886.008000 Utc= 340735380.128 == 18.10.2010
    16:43:0.128
```

C/C++ pre-processors ensures that – if the macro is set correctly – all printf messages are compiling to no code. For example, the line containing the printf statement in Listing 12.3 will only be executed if the compiler is started with `"g++ -DDEBUG file.cpp"`.

Listing 12.3: printf Statement surrounded by a Pre-Processor Statement

```
#if DEBUG
printf("something here");
#endif
```

12.1.2 *Debugging using Telemetry Messages*

This debugging technique is equivalent to write out log files on the embedded system and transfer them to the host computer (cf. to Chapter 5). Often not only the most recent log records can be received, but also historical and more detailed data: in space jargon these data types are referred to as *real-time*, *historical* or *extended* TM.

As it was already explained in Section 9.3, the onboard TM/TC functionality is used in order to get status information from the S/C. From ground, engineers can send specific TCs in order to receive exactly that kind of TM messages which they need for system analysis in case of an error or malfunction.

Taking again the application `TimeControl` from Section 12.1.1 as an example, if one wants to specifically request from ground the current onboard time of the TET satellite, a TC will be send to the satellite, causing the onboard S/W to invoke the `getHkData`-function which is inherited from the `ApplicationInferface`-class (cf. to Listing 12.4, Line 5).

12.2 NON-INTRUSIVE MONITORING TECHNIQUES

Within this section techniques will be described which allow the monitoring of the boot image of a satellite without modification of the source code. This technique is often used for *black box testing* where the S/W can be tested without having any special knowledge about its internal structure.

12.2.1 *Software-only Monitors*

A very comfortable way for a programmer to debug his embedded S/W is to use a dedicated debug port on the target H/W in order to communicate with a S/W component which resides in a section of the target ROM. This component is sometimes referred to as a *monitor* (cf. to Simon [77, sec. 9.3, p. 280].
Communication with the monitor is normally established using some

Listing 12.4: TET Source Code for requesting TM information

```
class ApplicationInterfaceTimeControl : public
    ApplicationInterface {
public:
    ApplicationInterfaceTimeControl() : ApplicationInterface("
        TimeControl", APID::TIMECONTROL) { }
    ...
    bool getHkData(unsigned char* buff) {
        longToBigEndian(&buff[CurrentHK_Fields::LOCAL_TIME],
                        (long)(NOW()/SECONDS));
        longToBigEndian(&buff[CurrentHK_Fields::UTC],
                        (long)(sysTime.getUTC()/SECONDS));
        return true;
    }
    ...
};
```

kind of serial port. Via the monitor it is hereafter possible to load the boot image into RAM and start it. Moreover, the monitor has also debugging and profiling capabilities: it can show the symbols used by the embedded S/W or the functions called during execution, display memory and register values and allows to set breakpoints[28].

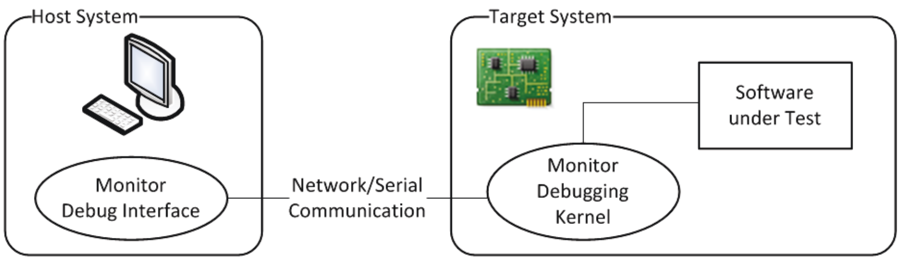


Figure 12.2: S/W-Only Monitors

Monitors are a part of the BSP and normally provided by H/W vendors. They consist of two parts (cf. to Simon [77, sec. 10.4, p. 323 ff]) which are also shown in Figure 12.2, namely the *debug interface*, residing on the host computer, and the *debugging kernel*, residing on the target system. As all parts of the monitor are implemented only in S/W, no additional H/W modification is necessary. Therefore, when running the S/W on the final flight H/W – which does not offer these debug capabilities – no additional errors will occur. This monitoring technique is therefore especially useful in the implementation and integration phase in the early stages of a satellite project where an

28 It has to be remembered, that using this breakpoint functionality violates the real-time constraints of the system (see Footnote 27).

end-to-end test is not yet possible (cf. to Chapter 6 for details about the various phases during the lifecycle of a satellite).

An example of such a monitor is *GRMON* offered by the company *Aeroflex Gaisler*, which is available for all LEON-based systems and therefore widely used in the space sector (see [25]). GRMON's internal profiling feature is the main tool used for the testing scenarios in Chapter 18.

GRMON Profiler

The GRMON profiler offers also the possibility of doing a statistical profiling: the profiling function collects samples of which code is currently executing and derives statistical information on the amount of the execution time spend within each function. It does not take into consideration if the current function is called from within another procedure. The GRMON profiling is non-intrusive and collects samples as often as possible. The only bottleneck is the debug link. Therefore it is of advantage to use the fastest of the debug-links that are supported by the system on which the embedded S/W will run on.

12.2.2 *Monitors using the JTAG target interface*

A common way to debug embedded S/W is to use a special debug port offered by the target H/W. Widespread on most development boards is the Joint Test Action Group (JTAG) port [29]. Initially invented by H/W developers to verify the developed electronics, the JTAG port is nowadays widely use by S/W developers as a back door for S/W debugging [see 77, sec. 10.4]. Similar to the S/W-only monitors explained in the above Section 12.2.1, there exists a piece of S/W on the host which serves as the debug interface and controls the target microprocessor through a dedicated cable. JTAG is supported by most of todays microprocessors and does not use any valuable target resources. Like debugging using a serial or Ethernet connection also JTAG usually requires dedicated H/W.

12.2.3 *Logic Analyzers*

Another possibility to monitoring and debug the execution of an embedded S/W image is to use a logic analyzer ([77], p.317ff). With such kind of tool it is possible to capture an execution trace during runtime, containing the list of instructions the microprocessor executed during a certain time period. Dis-assembling this trace displays the assembly instructions of the boot image, some logic analyzers also

29 This capability is also sometimes also referred to as Background Debug Monitor (BDM).

offer the possibility to link the trace against the corresponding lines of the source code[30].

12.2.4 *Oscilloscopes*

The main purpose of an oscilloscope is to graph voltages versus time [see 77, sec. 10.4]. As an analog device it detects the exact voltage of a signal. When it comes to testing and debugging the embedded S/W on the target H/W, oscilloscopes can be very useful to find errors which might occur during operation of the S/W, but have their origin in testing faulty H/W circuits.

As an example it can be mentioned that during debugging of a non-functioning wireless communication link in use case 3 of the Monitoring F/W Prototype (cf. to Section 16.4.3) it was found with the help of an oscilloscope, that the sent out TCs were received correctly by the experimental *FloatingSat* satellite. From this we were able to conclude, that the physical transmission was working fine, and that the error is most likely located within the embedded S/W part which is responsible for further processing the received messages on board of the satellite.

30 For the latter one the trace of the logic analyzer has obviously to be downloaded/-transfered to the development host computer

Part V

UNIFIED MONITORING

If I had asked people what they wanted,
they would have said faster horses.

— Henry Ford

13 SHORTCOMINGS OF CURRENT MONITORING TECHNIQUES

Within this chapter the main problems of the State-Of-The-Art in the area of monitoring embedded S/Cs will be explained, leading to the new idea of Unified Monitoring which is explained in Chapter 14. We will refer to the techniques described in Section 12: KINDS OF MONITORING and address their shortcomings, ordered by their type.

The monitoring techniques described in Chapter 12 provide an insight into the different kinds of information worlds of a S/C which often hold the same kind of content. They coexist with often no interchange of information at all, although there is just as often a significant intersection of the content that has to be provided. This situation has several disadvantages, some of them are quite crucial. In the following paragraphs we will describe the most important ones – many of them have been the motivation for starting this thesis. The problems can be divided into three different categories: functional , performance and usage shortcomings[31].

13.1 FUNCTIONAL SHORTCOMINGS

13.1.1 *Limited operational Usage*

The majority of the (low-level) debugging techniques presented in Chapter 12 is not meant to be used in the operational scenario, as most of them require a direct physical connection to the embedded system. This also inhibits access to the lower levels of abstraction during operation which is needed for debugging purposes in case of FDIR activities (cf. to Chapter 8). Taking printf as an example, it is typically used to print out messages to the terminal. It is not designed in a way to easily add more backends (sinks), and – most important – it is not meant to be used in a remote scenario like satellite operations. printf debugging is dedicated for AIV/AIT purposes, using the EGSE and check-out equipment in the laboratory (cf. to Chapter 6). Even though it would be valuable to use printf debugging also during the operational phase in orbit, this is currently not practiced during most missions – although technically this would be possible to a certain degree[32].

31 Some of the shortcomings can be sorted in several categories at the same time. If this is the case, the most significant category is chosen.

32 The printf output meant to be printed out on a terminal by directly accessing the satellite could be redirected and wrapped into TM frames to be transferred to ground.

Leaving out problems e.g. with time delays because of signal transmission delays to ground, the groundstation responsible for operating the satellite would normally also not allow a direct – e.g. terminal-based – access to the S/C, mainly because of safety reasons. However it has to be mentioned that there exist dedicated PUS services which offer limited insight into the functional abstraction level of the S/C (cf. to Chapter 8 and Section 9.3).

Currently, if one wants to get a deeper insight into the internal runtime behavior of the S/C and its onboard S/W, one has to analyze the gathered information after transmission to ground which is often referred to as *offline checking* of S/C data. For this, complex techniques like data mining have to be used to analyze the stored housekeeping and TM data [75].

13.1.2 *System Crash Limitations*

Some tools have the disadvantage that they are not usable in case of a complete crash of the embedded system. The logic analyzer (cf. to Section 12.2.3) is a good example for this inability: in case of a crash all information regarding memory, registers, etc. is gone.

13.1.3 *Fixed Granularity*

printf-messages are hard-coded in the application code and therefore have almost no options to be configured at runtime. This is especially important for the granularity of the printf-messages. The user gets all information (infos, warnings, etc.) from all applications – whether he is interested in them or not (cf. to Listing 12.2 for a good example). Resulting from this behavior, it can be very difficult, time consuming and error prone for the user to find the needed information in log files where the printf-messages could be stored in. In addition, as it can be seen in Figure 3.2 an increase of the granularity of monitoring information from a faulty subsystem may become necessary in case of a FDIR activity.

13.1.4 *Violation to Space Coding Standards*

The usage of pre-processor macros (e.g. for surrounding printf statements, see Section 12.1.1) is often forbidden in coding standards applicable for space industry.

13.1.5 *Static Presentation of Information*

Not only the granularity, but also the display and presentation of the information is often not very flexible in terms of configuration (see Section 13.3.4 below).

13.1.6 *Insufficient Traceability during Operation*

The origin of the information and the processes/threads/applications involved in information handling and/or processing can not – or only hardly – being traced back in the remote operational scenario. The same holds for the reconstruction of the temporal order of occurrences of events (time events or interrupts) and their correlation to the corresponding behavior of the onboard S/W in terms of state changes or method invocations. But just the latter point is most important for the operations team in order to find the error in the system which is responsible for the misbehavior or failure of the S/C.

13.2 performance shortcomings

13.2.1 *Memory Consumption*

By holding redundant information not only the size of the source code and the boot image will be larger than necessary, also the memory consumption at execution time will most likely be bigger than having only one single framework for information processing.

13.2.2 *Processor Load*

In the case, that printf statements used for debugging purposes are not deactivated or removed before launch, they lead to an unnecessary processor load. One reason behind this behavior lies in the implementation of how printf's are send over a UART-port from the embedded H/W to a connected development host: the string contained within a printf message is divided into its single characters, and afterwards character by character (one at a time) is send out. This is typically done within one big loop which is invoked also in the case if there is no physical connection to the UART port[33].

13.2.3 *Reaction to Interrupts*

An immediate consequence of the unnecessary processor consumption described in Section 13.2.2 is, that the reaction to interrupts is also affected: the reaction time to occurring interrupts (coming from either in- or outside the embedded system) is prolonged by spending time for formatting and sending printf-statements e.g. via UART. Therefore, within the time of treating this kind of debug statements the functional behavior of the system is impaired.

33 To give an rough approximation of the time consumed by this process, the transfer of 10 chars over a 115kbps-fast UART connection will take approx. 1ms

13.2.4 *Bandwidth Utilization*

As already addressed in the MOTIVATION Chapter 1, for all communication taking place on the S/W bus of the satellite and – even more important – on the radio link to the groundstation, the saving of bandwidth is a critical point. Obviously the harshness of this requirement differs from mission to mission, depending on what kind of transmission technique is used (S-band, X-band, etc.). But also for missions with a great bandwidth it is fact that the less housekeeping information is sent to ground, the more bandwidth is available for the mission and/or scientific data [34]. Current approaches to get rid of all unnecessary and redundant information within transmission records concentrate on the avoidance of always repeated and never changed types of messages (like e.g. the ones containing strings) [35].

Static and irrelevant Information is transmitted to Ground

What is completely missing today is to concentrate the download to ground only on the relevant information which strongly depends on what the user is most interested in. Taken an extreme example, in case of an FDIR activity the operational engineers are only interested in the detailed information regarding the affected subsystem(s), so all bandwidth should be reserved for this kind of data – leaving away everything else (including payload data) in order to safe the mission.

13.2.5 *Violation of Observability Principle*

All of the shortcomings regarding performance issues described in the subsections above signalize one major disadvantage of monitoring techniques which utilize the source code (like e.g. printlining): they are a violation against the *Principle of Observability* described in Section 14.3 and lead to the – so called – *probe effect* (cf. to [39], section 12.2, p. 293). By changing the source code in order to monitor or debug the system, the functional and temporal behavior is also influ-

34 An exception are smaller scientific and/or technology demonstration missions which are not bound to a standardized communication protocol (like e.g. CCSDS) and have therefore more freedom in terms of which communication techniques they use. One possibility for these kinds of missions is to store a dedicated debug boot image on board which contains all printlining information needed for debugging. In case of an FDIR activity this image can be booted and the debugging data normally outputted to a terminal is redirected to the antenna downlink and send down to earth as TM, leaving away the science data completely. As a drawback, this kind of images often need more onboard memory when stored on board due to the contained debugging data. Another possibility is to invoke a dedicated debugging shell directly on the satellite and send appropriate shell commands over the normal TM chain.

35 Currently this is a very active research topic, as there are no efficient techniques available for filtering out this kind of information – neither within the inter-task communication on board nor when using the radio link to the ground station. Though it has to be mentioned that a the time writing this thesis, a bachelor thesis was published which is addressing this issue within the S/W environment of the AOCS subsystem (cf. to [72])

enced. This may introduce new and hide old errors which has to be avoided.

13.3 USAGE SHORTCOMINGS

13.3.1 *Effort Overhead*

13.3.1.1 *Development Overhead*

Often the same kind of information has to be provided to both information worlds: the debugging and the TM world. This is an unnecessary overhead. Figure 13.1 shows an example of the information worlds holding the state of the OBC. These kind of intersection diagrams can be drawn for all other subsystems – like ACS, Onboard Navigation System (ONS), etc. – as well.

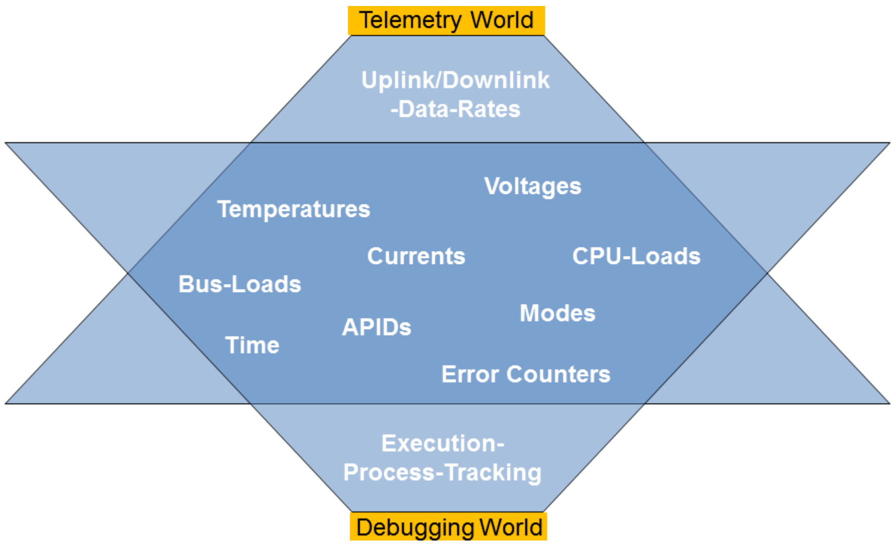


Figure 13.1: Redundant Information Worlds (here: OBC-Status Information) [14]

13.3.1.2 *Increased Test Effort*

As a result from the overhead when writing code and providing the same kind of information to two information worlds, the correct reception and execution of TCs has also be verified in both worlds. As an example please cf. to Figure 10.2 in Section 10.3 where both – effects in prints and TM – have to be controlled after sending the command to the S/C.

In addition, like described in Section 12.1.1 print statements are often surrounded by pre-processor macros. One of difficulties resulting from this technique is, that higher effort has to be put into making the code testable. Given an example, the well know test-tool *Polyspace*

[49] – which is often used to perform static code analysis [50] – can not handle these statements. The developer has to define each single macro within the tool to make the tests run through.

13.3.1.3 *Increased Change Effort*

Like the additional efforts described above when providing the same information to two information worlds (see Section 13.3.1.1) and testing them accordingly (see Section 13.3.1.2), also the effort when changes have to be done is twice as high because both information worlds have to be adapted.

13.3.2 *Error Prone*

As a direct consequence of the increased test effort described in Section 13.3.1.3 above, in case of a change both information worlds have to be adapted at the same time which is obviously error prone. More errors can result from the removal of used pre-processor macros for getting rid of printf statements at compile time (see Section 12.1.1): just one typo within e.g. the macro definition can lead to an unwanted code fragment remaining in the final flight version of the OBSW.

13.3.3 *Code Readability*

printf and TM/TC messages providing the same kind of information for two different information worlds will lead to an unnecessary confusion for someone who has to look for errors and/or has to understand the source code. The simpler information is provided within the actual implementation to the information worlds the better the code can be understood. Two times the same information within the same source code will mess things unnecessarily up.

13.3.4 *Learning Curve for New Tools*

For subsystem engineers and especially S/W developers of the embedded satellite S/W it is often confusing – and may result in additional cost and time-consuming training sessions – if they have to monitor their subsystem e.g. in the control room of the groundstation, looking for the information they need by using tools and/or displays they are completely unfamiliar with. In the extreme case, an engineer working at his subsystem console in the integration room may oversee important status and/or error-messages, or has major difficulties to track the execution steps of the S/W. Therefore it would be highly advantageous if the display (incl. filtering) of information from the satellite can be adapted in a way that e.g. the S/W engineer has the same debug output at hand that he is used to – be it from his development computer or from the EGSE tools used in the integration room.

14 THE IDEA & VISION OF UNIFIED MONITORING

Within this chapter the main idea behind the Unified Monitoring approach is presented. The technical solution accompanying this idea will be presented in the following chapters (starting with the identification of essential requirements in Chapter 15), introducing the so called Monitoring F/W.

14.1 THE VISION OF UNIFIED MONITORING

The vision of Unified Monitoring is to cover all monitoring abstraction levels described in Chapter 8 and provide this information during building the satellite as well as during the operational mission scenario. Everybody who is involved in building and operating the S/C shall be equipped with an infrastructure and methodology which embeds a unified access to S/C status information within the development process.

In order to achieve this goal the technology which brings this idea into practice – the Monitoring F/W – has to be included in an abstraction level from which it is possible to cover all other abstractions as well. Therefore the lowest level of abstraction was chosen which is the functional level described in the Section 8.3. The Monitoring F/W introduced in the subsequent Chapters 16 and 19 brings together the techniques described in that section, it can therefore be seen as a powerful tool which combines step-by-step debugging and the capabilities of a profiling tool.

Monitoring F/W combines Step-by-Step Debugging with Profiling

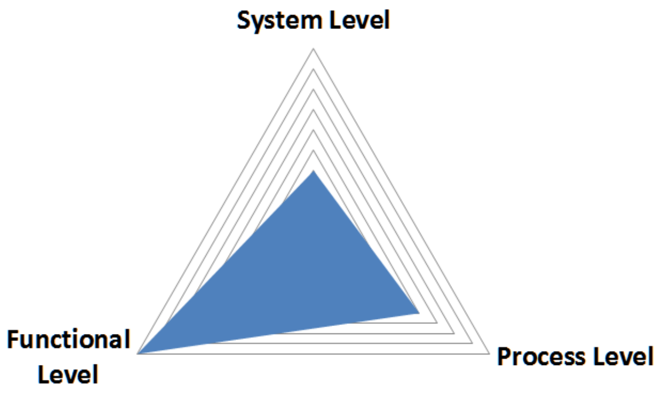


Figure 14.1: Focus of the Unified Monitoring in terms of Abstraction Levels

While the Unified Monitoring technology is rooted in and has its focus on the functional level, the abstraction levels stacked above are covered as well – however with a declining focus. Figure 14.1 depicts the levels of abstraction and the degree of how much the Unified Monitoring puts focus on them.

14.2 THE MAIN IDEA: SEPARATION OF CONCERNS

The goal of monitoring a S/C is to get as much detailed information about the current status of system as possible in order to make reliable predictions for the future of the system. The goal of Unified Monitoring of S/Cs is to hold all this information within one single information world so that all actors dealing with the system satellite within its development process have access to exactly that kind of information they need in order to fulfill their job. To achieve this goal and to get rid of most of the problems stated in the above Chapter 13, the information worlds described in Chapter 9 have to be combined. In addition – as already mentioned at the beginning of Chapter 11 – the integration of the embedded parts of the Monitoring F/W have be done at the lowest abstraction level possible (with respect to the onboard S/W).

Combination of Information Worlds and Integration in Functional Abstraction Level

In order to address these two key requirements we have to separate the different functionalities of the system by decoupling the information sources and sinks into the following functionalities [36]:

1. logging of information directly in their corresponding sources,
2. sending them to connected sinks, and
3. displaying and formatting them accordingly.

This kind of separation is called *Separation of Concerns*-principle [37], it is depicted in Figure 14.2.

The Monitoring F/W – introduced from Chapter 15 on – is designed in a way that it takes this principle into account which becomes more and more important especially for distributed embedded systems (cf. also to [6]). In order to use and evaluate the F/W it is at a first included in the RTOS RODOS (cf. to Section 5.2.1).

14.3 FURTHER DESIGN PRINCIPLES

The Separation of Concerns principle presented in Section 14.2 is only one principle to be taken into account for the design of the Monitor-

36 In contrast to e.g. printlining where – even if the output may be redirect to a different sink – the final formatting options are already contained in the statement itself.

37 In order to illustrate this principle, the LaTeX document preparation system which was also used to prepare this thesis can be taken as a good example: the actual content together with some basic formatting options is stored within a `.tex`-file. From this content the actual formatting (article, book, ...) is as well separated as the final type of the target file (PDF, Postscript, HTML, ...).

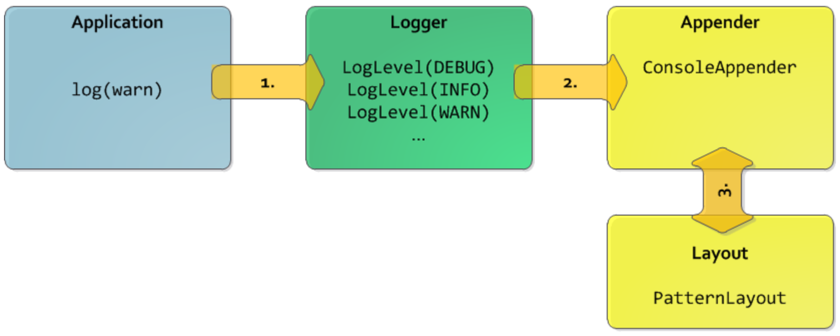


Figure 14.2: Separation in the Data Flow of Monitoring Information (Example based on the Monitoring F/W Prototype)

ing F/W. Further proven principles are explained in the book *Real-Time Systems* from Herrmann Kopetz ([39], section 2.5, p. 45 ff):

- Principle of Abstraction (components as basic structural units)
- Principle of Separation of Concerns (also called Principle of Partitioning)
- Principle of Causality (targeting for a deterministic behavior)
- Principle of Segmentation (enabling a sequential step-by-step analysis of the S/W)
- Principle of Independence (e.g. of components)
- Principle of Observability (achieved by visible communication channels between components, using a multicast topology).
- Principle of a Consistent Time (using a global time base – especially important for distributed embedded systems)

Beside the Separation of Concerns principle, for the design of the Monitoring F/W the Principle of Causality and the Principle of Observability are of special importance: we follow the Principle of Causality because we want to know where the monitoring information is coming from. The tracing of information is very important, especially for failure analysis and detection. The Principle of Observability is taken into account in order keep the *probe effect* – which is caused e.g. by using printlining debugging techniques (cf. to Sections 13.2.5 and 12.1.1) – as small as possible. Finally, as the Monitoring F/W will enhance the traceability of execution steps of the running onboard S/W also the Principle of Segmentation can be considered as fulfilled.

Part VI

THE MONITORING FRAMEWORK

The following part concentrates on the technical component for bringing the idea of Unified Monitoring into practice: the Monitoring Framework. Starting with the collection of requirements, first a laboratory prototype will be developed, upon which a flight version evolves.

15 REQUIREMENTS ON THE MONITORING FRAMEWORK

The idea behind Unified Monitoring was explained in detail in Chapter 14. In order to put this idea into practice, an embedded S/W F/W is introduced which main purpose is to take over the functionalities of the different information worlds from Chapter 9. The F/W is intended to be used through all development phases (cf. to Chapter 6) by all involved actors (cf. to Chapter 7), preparing the necessary information for the user as needed in each life cycle of the S/C. This design-idea for the F/W is oriented towards the system monitor introduced in Section 11.1. The system context diagram in Figure 15.1 shows the environment the Monitoring F/W is acting in and the boundaries of the F/W. Please cf. to appendix Section A.1 for the notation used herein.

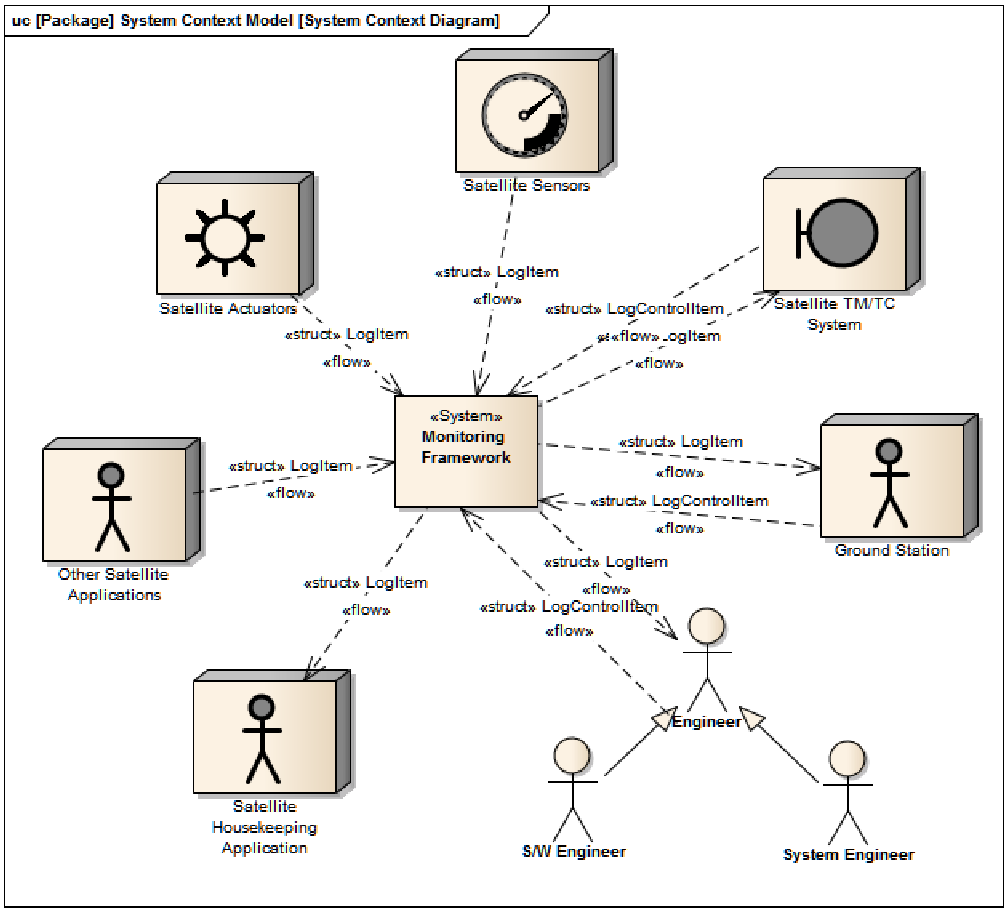


Figure 15.1: System Context diagram of the Monitoring F/W

The main focus lies on the replacement of the internal debugging features of the system on the one hand, and on the other hand sub-

Monitoring F/W = Debugging + Housekeeping

stituting the system's Housekeeping application. The fact that the housekeeping application is still shown as a user system in the context diagram is due to the design goal that the usage of the Monitoring F/W shall not forbid the existence of the classical Housekeeping application. One reason behind this is, that e.g. in the case of the F/W prototype a complete replacement would mean to also switch over to the publish/subscribe based middleware as the communication platform of the system. This is on the one hand not always possible and on the other hand not always the best communication solution for all embedded systems. In this sense, the Monitoring F/W shall leave the final decision of completely replacing the Housekeeping application as a design decision to the S/W engineer. The same argumentation as for the Housekeeping application also holds for the system's TM/TC application. While the Monitoring F/W is strongly coupled to TM as well as TC by sending the monitoring information and receiving steering and control commands over the antennas, they both coexist in separate functional components on board of the satellite. One of the main engineering reasons behind this design is to develop a most flexible and portable system, in which the monitoring is not dependent on a special TM/TC implementation.

Like shown in Figure 15.1, the structure `LogItem` holds the monitoring information, whereas the `LogControlItem` structure is used for sending control messages to the F/W. Monitoring information have exactly one reproducible source, but can be distributed to many sinks and formatted in many ways.

When including the F/W into the onboard S/W and making it a library to be linked – together with the RTOS – into the final boot image, it is one of the design goals to keep the F/W's influence on the functional and timing behavior of the overall system on a minimal level. In addition, the F/W has to provide very simple interfaces to make it very easy for the embedded S/W developer to use.

15.1 GENERAL REQUIREMENTS

When designing the new Monitoring F/W, besides the common requirements valid for the development of every bus S/W (see Section 5.2.1 and esp. the non-functional requirements in Table 5.2) there are several general requirements which have to be taken into account – following the idea of a requirements driven design as described in [59].

First, the general idea of Unified Monitoring which is presented in Chapter 14 leads to the following high-level requirements:

RECORD ALL KINDS OF INFORMATION The Monitoring F/W shall be a single source of information: it shall be used during all phases of development (cf. to Chapter 6 and holding all kinds of information regarding the satellite state (cf. to Chapter 10).

SIMPLE AND FAMILIAR USAGE The usage of the F/W should be as simple as possible and oriented towards e.g. the familiar printf statements.

RTOS INTEGRATION The technique behind the new logging mechanism shall utilize the communication techniques offered by the underlying RTOS (e.g. a middleware), using the preferred technique for all inter-process communication inside the embedded system (see Section 5.2.2)

SIMPLE AND APPROVED DESIGN The F/W design shall be oriented towards the architecture of the *log4j* logging F/W[38] which is widely used for Java-based applications. The design of log4j is very simple and similar to what is depicted in figure 14.2, nevertheless it is a powerful and mature logging F/W, offering features like classifying messages by categories, filter these categories accordingly and freely adjust its output [see 63, sec. 2.6.] [39].

LOW MEMORY AND POWER CONSUMPTION The integration of the Monitoring F/W into the onboard S/W shall result in a minimum increase of resource consumption.

DIVIDING INPUT FROM OUTPUT Due to the resource constraints mentioned above, the user has to be able to deactivate the message outputs (sinks) at runtime.

VARIABLE OUTPUT Output shall be send to various message sinks (console, file, network, ...) which can be connected and deconnected to the F/W at runtime.

By using the RODOS middleware it shall be possible for the F/W prototype to send logging information over the network to any other RODOS instance using gateways, like depicted for the RODOS RTOS in Figure 15.3. The formatting of the received messages on the host computer shall be done by appender and layout modules. They should be configurable in a way, that – if requested – the console output is similar to what is produced by the printf statements described in Section 12.1.1, but any other output shall also be possible.

In order to facilitate the first realization of the idea of Unified Monitoring by developing a prototype of the Monitoring F/W, the initial high level requirements are specified in more granularity within the SysML requirement diagram[40] shown in Figure 15.2. This diagram is explained in more detail the appendix Section A.4.

38 http://logging.apache.org/log4j

39 Because of its high adaption levels log4j was chosen as the prime candidate for the logging component of the *European Ground Systems - Common Core* (EGS-CC). This initiative has the goal to develop a common infrastructure to support space systems monitoring and control in pre- and post-launch phases for all mission types [66, 67]

40 See [84, sec. 4.3] for details regarding this type of diagram.

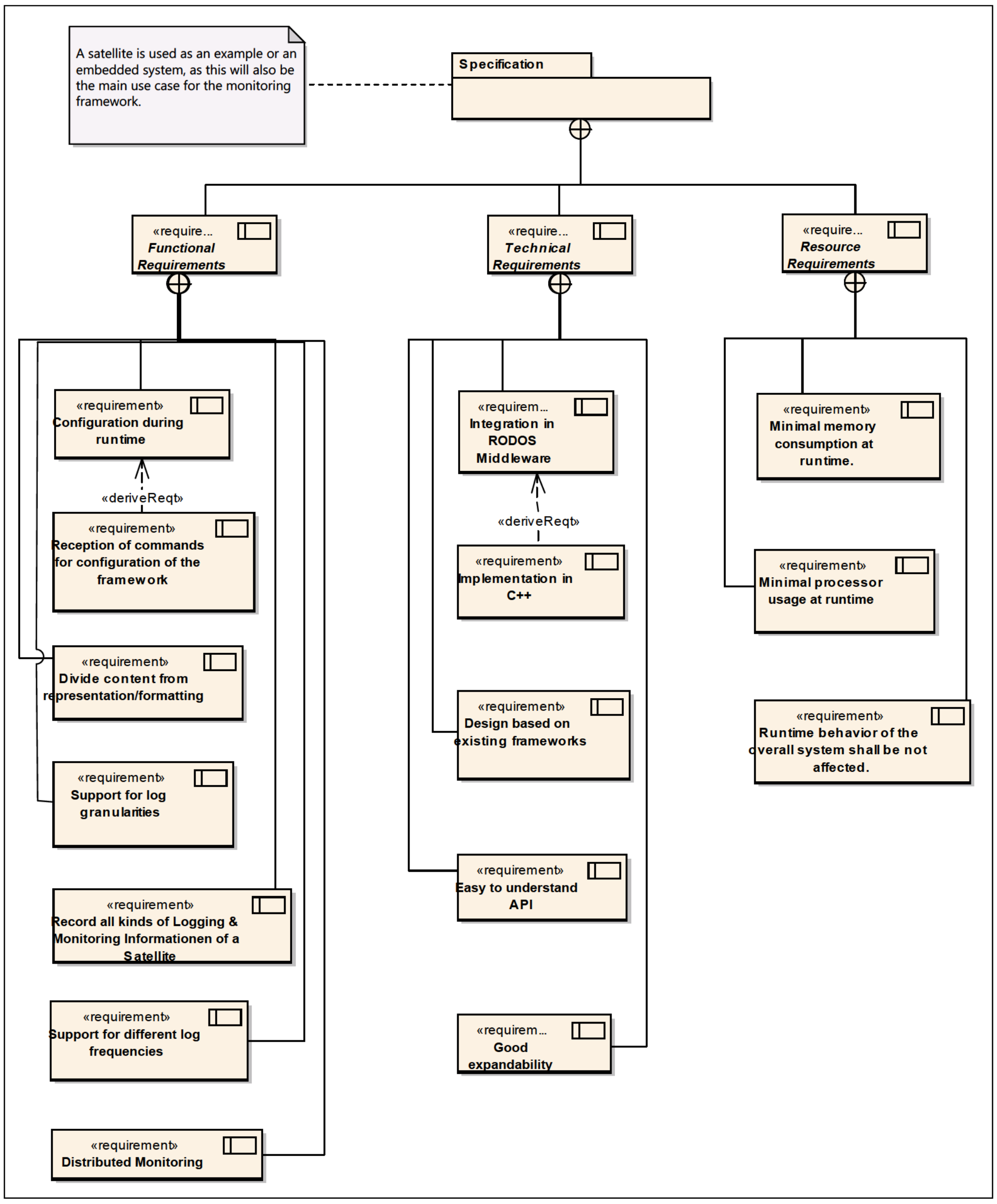

A satellite is used as an example or an embedded system, as this will also be the main use case for the monitoring framework.
Specification
«require... Functional Requirements
«require... Technical Requirements
«require... Resource Requirements
«requirement» Configuration during runtime
«deriveReqt»
«requirement» Reception of commands for configuration of the framework
«requirement» Divide content from representation/formatting
«requirement» Support for log granularities
«requirement» Record all kinds of Logging & Monitoring Informationen of a Satellite
«requirement» Support for different log frequencies
«requirement» Distributed Monitoring
«requirem... Integration in RODOS Middleware
«deriveReqt»
«requirement» Implementation in C++
«requirement» Design based on existing frameworks
«requirement» Easy to understand API
«requirem... Good expandability
«requirement» Minimal memory consumption at runtime.
«requirement» Minimal processor usage at runtime
«requirement» Runtime behavior of the overall system shall be not affected.

Figure 15.2: Functional, technical & resource Requirements on the Monitoring F/W

These requirements are the input and served as the starting point for the design and implementation of the Monitoring F/W prototype which was performed within the scope of a diploma thesis [63]. For this thesis special emphasis was put on following the paradigms of Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) [64]. Therefore, in [63] Section 4.1.1 starts with the identification of functional requirements coming from the actors dealing with the F/W and the definition of appropriate use cases hereafter. Afterwards the non-functional requirements which correspond to the technical and resource requirements shown in Figure 15.2 are addressed in Section 4.1.2.

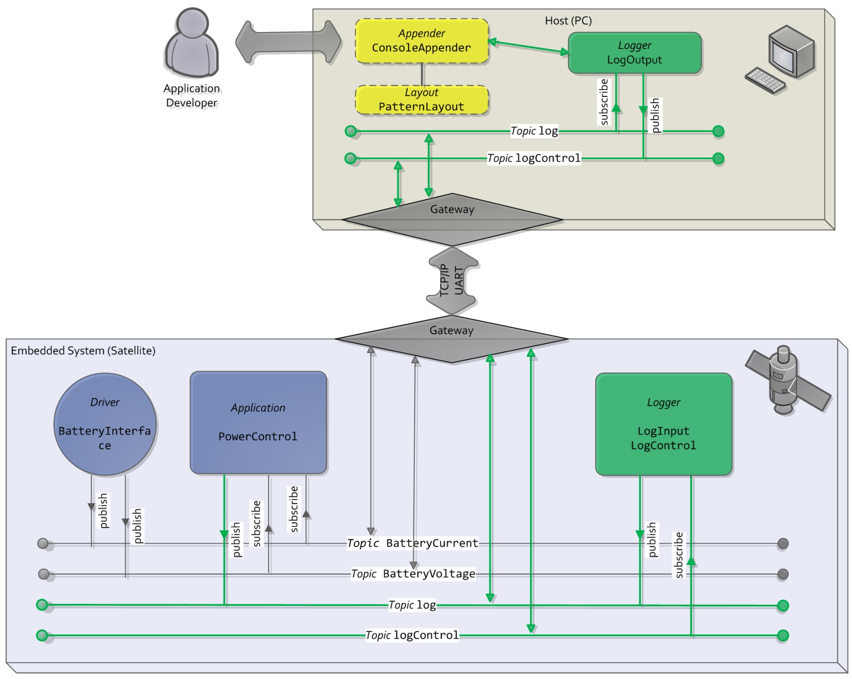


Figure 15.3: RODOS Debugging using the Monitoring F/W

15.2 DESIGN-IMPACT FROM INFORMATION WORLDS

The decision into which place the embedded parts of Monitoring F/W have to be integrated is driven by the issues addressed in Chapter 8 (and esp. Section 8.3 herein) and at the beginning of Chapter 11: the Monitoring F/W has to be included in the lowest abstraction level – the functional level – in order

F/W Integration into Functional Level

1. to cover all abstraction levels and provide monitoring for all information worlds, and therefore
2. to be able to detect errors before they propagate to a higher level of abstraction within the system.

The interface of the Monitoring F/W prototype has to be designed as such that is similar to printf (cf. to Section 12.1.1), meaning the integration of the corresponding F/W methods to be invoked directly within the functions of the application S/W (cf. to the Sections 16.1.2 and 17.2.3 regarding the F/W's user interfaces). This design makes the Monitoring F/W prototype an ideal candidate for replacing the traditional printf debugging which is the focus of the prototype's use cases (see Sections 16.4.1,16.4.2 and 16.4.3) and also of the first flight version tests performed in Section 19.

15.3 DESIGN-IMPACT FROM PROJECT-PHASES

In the following text we will show how the development of the Monitoring F/W is connected to the temporal aspects of the development of a S/C with respect to the different project phases (cf. to Chapter 6). The spatial aspects in terms of the different S/C information worlds were already addressed in the above Section 15.2).

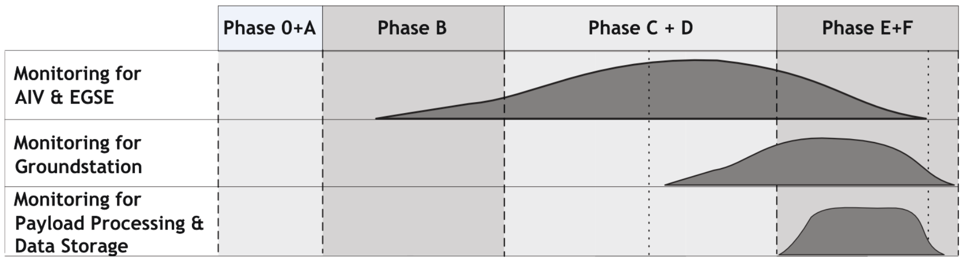


Figure 15.4: Temporal Distribution showing how the Monitoring Load within the different Information Worlds are mapped on the Build Process of a S/C [24]

Picture 15.4 [24] shows how the work load in terms of monitoring within the various S/C information worlds is mapped on the timing of the S/C project phases[41]. As it can be seen, the work within the different S/C information worlds – although spread over several project phases – can be assigned to one major project phase when looking at their peak work load. This differs from the mapping of involved actors to project phases, as it is depicted in Figure 7.2 in Chapter 7: the involvement of the actors and their corresponding engineering domains are widely spread over the different the project phases. Because of this it is absolutely sufficient to derive the temporal impact on the design of the Monitoring F/W by mainly looking at the project phases (instead of focusing on the involved actors):

S/C Project Phases driving the F/W Design

PHASE 0 From the monitoring point of view, in this early planning phase a tailoring of the monitoring needs within the course of the project has to take place. The following boundary conditions and parameters can be identified:

- monitoring needs from the mission/payload
- performance of the monitoring component (frequency & granularity)
- constraints regarding transmission bandwidth
- constraints regarding storage capabilities on board and on ground

41 As mentioned in Chapter 9, the *Monitoring of Inter-Satellite-Network* was let out of the diagram because it is almost identical to the *Monitoring for Groundstation*.

PHASE A Keeping the monitoring constraints identified in phase 0 in mind, the Monitoring F/W's envisaged capabilities can be identified, resulting in the following decisions:

- make-or-buy
- re-use of an existing Monitoring F/W
- needed extensions of an existing Monitoring F/W

An extensive study of existing Monitoring F/Ws which are possible candidates on which a re-use decision can be based upon can be found in [63, chap. 3]. Within the frame of this thesis it was decided to start the implementation of the Monitoring F/Ws (both prototype and flight version) from scratch. Nevertheless the design of the F/W prototype was strongly influenced by this study, whereas the design of the flight version is then again based upon the F/W prototype's tests and lessons learned.

PHASE B Also for the monitoring component the integration phase has started:

- interfaces to other components and sub-systems – both H/W and S/W – have to be defined
- depending on the decisions from phase A, a first high-level design of the monitoring component will be determined

PHASE C, D, E & F From phase C on the Monitoring F/W must be ready to be used and integrated in the S/W source code of the flight S/W. Any requirements coming from these phases should have been already addressed in earlier phases, so that the F/W can be used for

- the breadboards to be developed in phase C,
- the EM and FM models to be developed in phase D,
- the FM model in the operational and disposal phases E and F.

(cf. to A.2 for the Eu:CROPIS model philosophy)

16 MONITORING FRAMEWORK PROTOTYPE

This chapter is dedicated to the development of the Monitoring F/W prototype. After describing its design and implementation we will show the compile- and run-time configuration options and depict some possible use cases which will be used in Chapter 18 for testing and evaluation purposes. From the experiences and results gained from the prototype the flight version of the F/W will evolve.

16.1 DESIGN

Taking the requirement of a – possibly middleware-based – message exchange using the underlying RTOS features for inter-task communication as a stimulus (cf. to Section 15.1 in previous Chapter 15) for finding a first simple design from which a Monitoring F/W could evolve, the introduction of an error task being connected by message queues to the rest of the system is a possible solution.
Chapter 7 of [77] (p. 173 ff) introduces a simple example of two tasks writing their error reports to a message queue[42] from which a dedicated error task reads and reacts to them – for instance – by increasing an error counter and writing into a shared data structure like an error log.

Furthermore, the design of the Monitoring F/W is driven by the major requirement that it shall be used by all actors in all phases holding all kinds of information within one single information world (cf. the requirements Section 15). In order to illustrate the concepts and relationships regarding this information world, we will use an Unified Modeling Language (UML) communication diagram based on the Entity–Control–Boundary pattern [43]. Within the diagram presented in Figure 16.1 only essential attributes and relationships are shown, the involved operations will be described later on in separate sequence diagrams for every control. For a description of the diagram symbols please refer to the appendix Section A.1.

Data entities correspond to the information to be recorded by the Monitoring F/W. As shown in Chapters 9 and 10 these are mainly information from debugging statements and the satellite housekeeping whose content is provided by terminal output (using `PRINTF` or `ERROR` methods) and the TM to be send to the EGSE or groundstation. The core library of the Monitoring F/W which will run as embedded

42 Cf. to Chapter 5.2.2 for other possible communication mechanisms alternative to message queues

43 The Entity–Control–Boundary pattern is a simplification of the Model–View–Controller Pattern (cf. to [26])

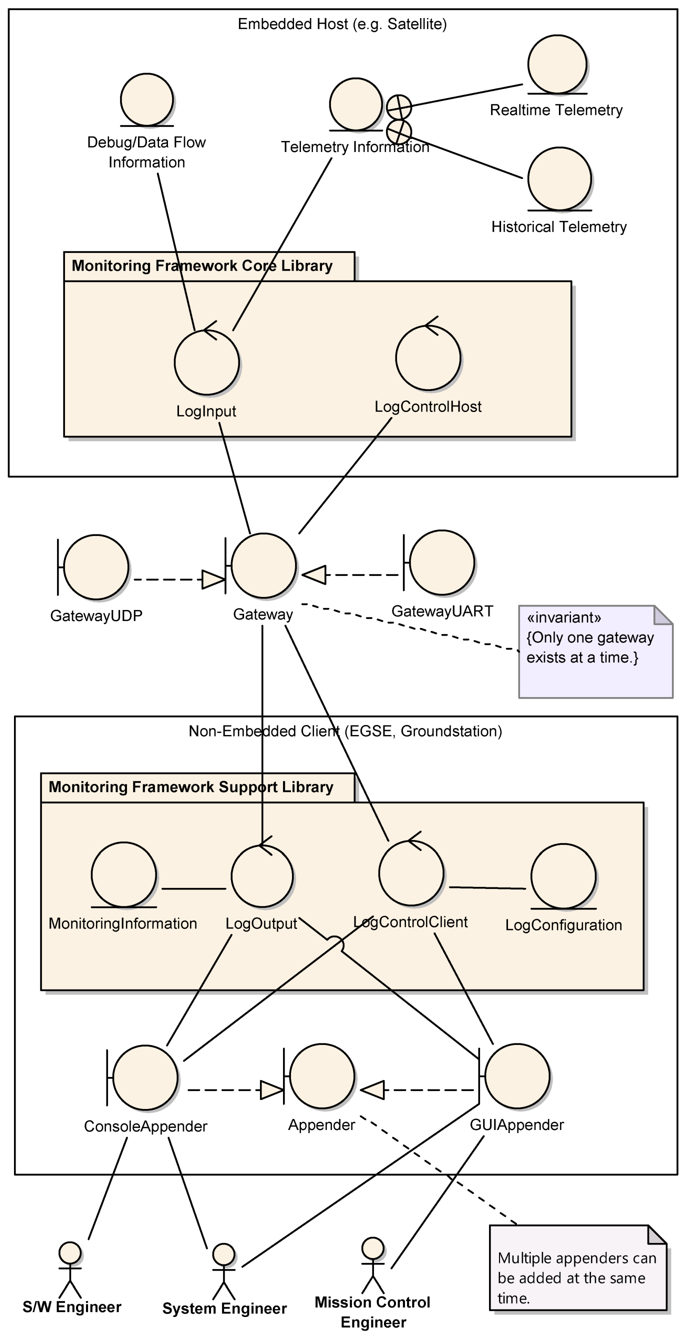

Embedded Host (e.g. Satellite)
Debug/Data Flow Information
Telemetry Information
Realtime Telemetry
Historical Telemetry
Monitoring Framework Core Library
LogInput
LogControlHost
GatewayUDP
Gateway
GatewayUART
«invariant» {Only one gateway exists at a time.}
Non-Embedded Client (EGSE, Groundstation)
Monitoring Framework Support Library
MonitoringInformation
LogOutput
LogControlClient
LogConfiguration
ConsoleAppender
Appender
GUIAppender
Multiple appenders can be added at the same time.
S/W Engineer
System Engineer
Mission Control Engineer

Figure 16.1: Communication Diagram of the Monitoring F/W Prototype

S/W on the OBC takes care of handling these request via the *LogInput* controller. Additionally, the *LogControlHost* controller will take care of the runtime configuration of the Monitoring F/W.

In order to send the monitoring information from the embedded target platform to a non-embedded system used by the main actors in this scenario (S/W-, system- and/or mission control engineer, cf. to Chapter 7), a RODOS gateway (cf. to Section 5.2.1.2) will be used as the system's boundary. Currently there are two concrete implementations for this gateway: the *GatewayUDP* for the primary RODOS development platform Linux and the *GatewayUART* for the LEON2 and LEON3 development boards used for the tests of the prototype (cf. to Section 5.2.3 for general information on the development environment typically used for embedded S/W).

On the non-embedded/client side which is usually a desktop computer running Linux the monitoring information coming from the satellite's OBC or its correspondent simulator are stored by the *LogOutput*-Control into the *MonitoringInformation*-entity. The actual *LogConfiguration*-entity holds the status of the Monitoring F/W, the *LogControlClient*-controller takes care of feeding in user requests and forwarding them forwarding them to the *LogControlHost*.

Several appenders realizing the *Appender*-boundary will serve as the interfaces to the actors. At the same time more than one appender can be connected to the Monitoring F/W, each one fulfilling the needs of one or more target users. The *ConsoleAppender* is primary targeted towards the S/W engineer, while the *GUIAppender* is intended for the usage in the control center by the mission control engineer. In addition, both appenders maybe also used by the system engineer.

Associated with every control is a sequence diagram representing the control's interactions with boundaries, entities, and other controls. These diagrams are shown in Section 16.2. To give a better overview, the controls are grouped together with respect to their functional purpose:

FIGURE 16.3 is showing showing the dynamic behavior of sending of monitoring messages to ground, and

FIGURE 16.4 depicts the dynamic configuration of the F/W at runtime.

16.1.1 *Integration into RODOS*

As the main field of application of the Monitoring F/W is to offer extensive logging functionality to the applications and threads residing within the application layer (see Figure 5.5), the parts of the F/W supposed to run on the embedded side (represented in Figure 16.1 by the

Monitoring Framework Core Library-box) will be integrated within the management layer of RODOS[44].

In addition, the non-embedded parts of the F/W (represented in Figure 16.1 by the *Monitoring Framework Support Library*-box) will deployed as a RODOS support library.

16.1.2 *User Interface*

The user interface of the Monitoring F/W prototype is driven by the replacement of the debug messages, mainly appearing in the form of `PRINTF`-messages within the source code (cf. to the corresponding Section 12.1.1). This design driver is reflected in two requirements regarding the user interface of the F/W

- The design shall be based on existing F/Ws.
- The user interface/API shall be easy to understand.

Confer to the requirements Chapter 15 and also the corresponding Appendix A.4 for further details.

As the initial design is strongly oriented towards the embedded logging F/W of M. Schulze ([73]), the selection was made for the C++ standard output streams `cerr` and `cout`. To these streams characters can be written as formatted data using the insertion operator «[45]. The usage of these programming language constructs is well known to every C++ programmer, like to following code snippet shows:

```
this->log(info) « "Battery Current (mA): " « batteryCurrent «
endl; 46
```

This user interface has several advantages, not only from the user point of view, but also on how information is being put into the Monitoring F/W: Monitoring messages are stored and transported using `LogItems`. Before a `LogItem` is sent over the middleware within a middleware message, it is being constructed out of – so called – `PreLogItems`. This can be seen in Figure 16.3 (for the description of the main constituents of the prototype please cf. to the following implementation Section 16.2). By using the « operator, the creation of `PreLogItems` is implicitly done during processing of the corresponding source code line:

44 If – in the future – the monitoring of the RTOS core may become necessary, some elements of the F/W may be also integrated within the core layer of RODOS (cf. to Section 22.2.1.2 for this future use case).

45 Insertion of unformatted data is also possible with these streams, e.g. by using member functions such as `write`. But as explained in the text, this would have the drawback of losing the information of the content of the data – which is crucial for proper handling of monitoring messages within the F/W (e.g. for serialization process before sending messages over the middleware)

46 `endl` is an object of the class `Endl`.

1. The first `PreLogItems` are being put to the `LogInputBuffer` by the `LogInputter` when it is first created by the `Logger`-method `log`: one `PreLogItem` with a pointer to the corresponding log level and one `PreLogItem` with a time stamp.

2. Afterwards, every part between each «-operator is packed as a separate `PreLogItem` and handed over to the `LogInputBuffer`. With respect to the content, there has to be a dedicated operator function of the «-operator for each data type.

3. As every operator function terminates by returning the initially created `LogInputter` as a self-reference, operator concatenations like shown in the code snippet above become possible.

16.1.2.1 *Handling of Strings*

Strings are handled slightly different than the other data types. On the one hand they have maximum length – like application- and thread-name – which can be adjusted by setting the corresponding parameter by using macros or CMAKE (cf. to Section 16.3). But as this length is valid per `LogItem`, long strings will be split over several `LogItems` and hence over several middleware messages.

16.1.3 *Buffering*

Like described in Section 16.1.2, monitoring messages inserted into the system are divided into several `PreLogItems` and finally `LogItems` before sending them over the RODOS middleware to other computing nodes (like a development host) using the `LogTopic`. Afterwards – on the receiver side – the monitoring messages have to be re-constructed out of the incoming `LogItems`. For this process buffering is needed on both sides which is done within the `LogInputBuffer` on the embedded side as well as in `LogOutputThread` on the receiver side using `SyncFifo` buffers (cf. to Section 16.2).

For the performance of the embedded system the design and configuration of the `LogInputBuffer` is most important. This buffer is explained in detail in [63, sec. 4.2.1.7], its main task is to collect and store the `PreLogItems` which are afterwards converted to `LogItems` by the `LogInputThread`. When an interruption from the scheduler occurs (e.g. in the case of a context switch) while log messages are being written to the buffer, the construction of `LogItems` out of `PrelogItems` is continued afterwards using the corresponding application- and thread-IDs. Problems may occur, if there exists more than one log

Context Switching

statement within the same thread, while at the same time the buffer is already filled with `PrelogItems`[47].

Buffer Size

One of the most important settings to control the runtime behavior of the Monitoring F/W prototype is the size of the `LogInputBuffer` which can be set using the `LOG_BUFFER_SIZE_INPUT` parameter (cf. to Listing 16.1). If the size it too small many `PreLogItems` (the latest ones) will be thrown away and the monitoring messages will be broken apart on the receiver side. If the size is too large too much memory will be consumed.

47 This problem can be solved by the introduction of a unique ID for every monitoring message. Cf. to the Sections 18.3.3 and 18.5.6.2 for the problems which occurred during testing and which could be avoided having such a feature integrated into the F/W.

16.2 IMPLEMENTATION

Figure 16.2 depicts the implementation of the Monitoring F/W prototype which is available for the RODOS RTOS (cf. to Section 5.2.1.2). The implementation and usage of this prototype is already explained in detail in [63] and [14]. Within this section we will therefore concentrate on the major constituents. One remark on the general design: as the interested reader can easily notice, most of the F/W's classes are implemented using the *Singleton*-control pattern [26] which ensures, that classes which build the core of the F/W exist only with one instance within the system. This prevents an unintended misuse and also ensures fast and lean implementation of the F/W.

The F/W can be divided into three major parts, which are clearly separated in Figure 16.2: the *Input-*, the *Output-* and the *Control*-part[48].

The Embedded Part

The *Input* part is meant to run only on the embedded side of the system to be developed, in general this is the OBC of the satellite. All of the classes contained here belong to the *Embedded Host*-part displayed in the upper part of Figure 16.1.

In the following the main classes of the *Input*-part and their purposes are summarized:

LOGGER The main interface class of the embedded application:

- has a unique application ID, ensuring that there is only one central logger in the embedded system[49]
- must be inherited by all applications which want to use the Monitoring F/W prototype

`LogItem`s contain the Monitoring Data for Downlink

LOGITEM contains the logging information data to be sent over the RODOS middleware to the non-embedded part for the Monitoring F/W:

- each `LogItem` contains the name of the protocoling application and thread
- a big monitoring message (containing e.g. long strings) can be spread over several `LogItems`

LOGLEVEL level of criticality of each message to be sent to the Monitoring F/W:

- to be defined by each application
- can be changed at runtime
- currently implemented levels: "debug", "info", "warn", "error" and "off"

48 Not shown within the class diagram is the *Meta*-part as it contains only elements which are not part of the prototype, but are used to debug the F/W itself. This so-called *MetaLogger* can be seen as a mini-logging F/W which is introduced in order to avoid the usage of the unwanted `PRINTF`-messages. It is not subject of this thesis, its main usage scenario was within the development of the prototype itself.

49 This corresponds to the *error task* introduced at the beginning of Section 16.1.

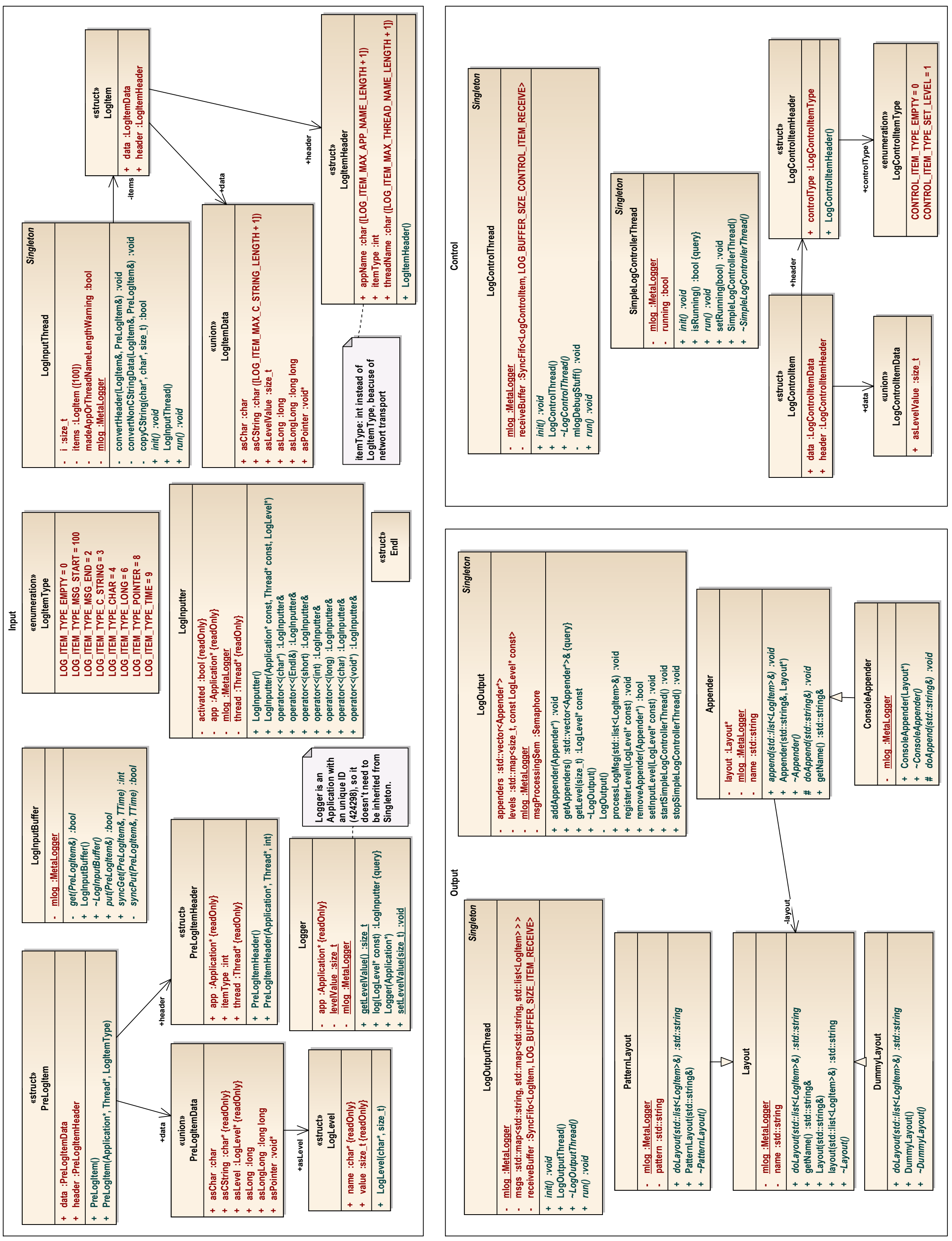

Input
«struct» PreLogItem
«union» PreLogItemData
«struct» PreLogItemHeader
«struct» LogLevel
Logger
LogInputBuffer
«enumeration» LogItemType
LogInputter
«struct» Endl
LogInputThread
«struct» LogItem
«union» LogItemData
«struct» LogItemHeader
itemType: int instead of LogItemType, because of network transport
Logger is an Application with an unique ID (424298), so it doesn't need to be inherited from Singleton.
Control
LogControlThread
SimpleLogControllerThread
«struct» LogControlItem
«struct» LogControlItemHeader
«union» LogControlItemData
«enumeration» LogControlItemType
Output
LogOutputThread
LogOutput
Appender
ConsoleAppender
PatternLayout
Layout
DummyLayout
Singleton

Figure 16.2: Class Diagram of the Monitoring F/W Prototype

- setting to "off" means deactivating the Monitoring F/W at runtime
- the user (e.g. the application programmer) can define his own log levels at compile-time

LOGINPUTTER handles the log requests for each `application`:

- will be created for each application by the `Logger` class
- gets information about the calling `application` thread and the `LogLevel` of every log request
- contains all operator functions for overloading of the «-operator, one for each data type to be sent as a monitoring message

LOGINPUTBUFFER "last instance" of the embedded part of the logging F/W before the Monitoring F/W sends messages over the RODOS middleware:

- configurable buffer size (via CMAKE, cf. to Section 16.3.1) enables scaling to limited H/W resources
- collects all log messages coming from the applications in a ring buffer structure
- inherits from RODOS `SyncFifo`-buffer (see Section 5.2.1.2)

LOGINPUTTHREAD Responsible for creating `LogItems` ready to be sent over the RODOS middleware using the corresponding `LogTopic`:

- reads all `PreLogItems` stored within the `LogInputBuffer`
- processes `PreLogItems` to `LogItems` and replaces all included pointers herein by the actual data
- sends the `LogItems` over the RODOS middleware using the `LogTopic`

As can be easily seen, the embedded part of the F/W is designed in a way, that the collection of messages coming from the applications is clearly separated from providing messages to the outside world by sending them over the middleware. The main reason behind this design and for not letting the `Logger` directly send messages over the network is an important real-time aspect: if the `Logger` itself is blocked (e.g. by a context-switch or an occurring interrupt), the `LogInputThread` can still hand over the monitoring messages to the middleware. In this way, a satellite using the Monitoring F/W can send messages to the ground station in order to initiate FDIR activities even in the case of a system blockade. In case the Monitoring F/W is already used within the early integration phases of the satellite taking place on ground this is a great advantage over common debugging techniques like e.g. using a logic analyzer (cf. to Section 13.1.2).

Satellite FDIR possible even if the Systems "hangs"

The second part of the Monitoring F/W is meant to run only in the non-embedded world (e.g. an EGSE system or a groundstation). It is marked as *Output* in the lower left corner of Figure 16.2, its main classes are:

The Non-Embedded Part

LOGOUTPUTTHREAD Opposite part to `LogInputThread` and `LogInputBuffer`:

- receives the `LogItems` coming from the middleware over the `LogTopic` (inherits from RODOS `Thread` and `Subscriber` classes, cf. to Section 5.2.1.2)
- if a monitoring messages is spread over several `LogTopics`, `LogOutputThread` is responsible for waiting until the last one is received before passing the message to `LogOutput`. For synchronization purposes a `SyncFifo`-buffer of RODOS is used
- like with `LogInputBuffer`, the buffer size can be adjusted using a macro or CMAKE settings

LOGOUTPUT responsible for receiving log messages from `LogOutputThread` and handing them over to all connected `Appenders`:

- receives the `LogLevel` set by the user and sends them to the embedded part of the F/W over the RODOS middleware
- `Appenders` can be added and removed to the `LogOutput` at runtime
- in order to ensure that a monitoring messages is forwarded to all registered `Appenders` at the same time, this task is done within a a critical section which cannot be interrupted

APPENDER Abstract base class, from which other classes have to inherit in order to write out monitoring messages:

- receives `LogItems` from `LogOutput` and transforms them to strings
- as an example a `ConsoleAppender` is implemented which prints out monitoring messages to a console on the development host

LAYOUT Responsible for doing the actual formatting of monitoring messages

- used by `Appender` do the actual formatting
- the formatting strongly depends on the chosen `Appender`, therefore it is implemented as an abstract class, from which concrete `Appender` instances have to inherit
- exemplarily, a `Patternlayout` to be used by the `ConsoleAppender` was implemented which formats the strings to be printed out on the console of the development host

Special emphasis in the *Output* has been put on the handling of big monitoring messages which are split over several middleware messages (e.g. if a monitoring message contains long strings, cf. to Section 16.1.2.1 for further details). If such a message is received completely by the `LogOutputThread` it will be handed over to `LogOutput` for re-constructing the initial monitoring message. Within this process no interruption is allowed, which leads to the fact, that this part of the implementation (a loop) is protected in a critical section which cannot be interrupted. This can be also seen in the sequence Diagram 16.3.

The F/W's Configuration Part

The third part – visible in the lower right corner of Figure 16.2 – is meant to *Control* the behavior of the Monitoring F/W at runtime. Within the prototype implementation the functionality implemented here is the changing of the `LogLevels` which can even lead to a complete deactivation of the F/W in the case of setting the level to "off". The Control part consists of the following classes:

LOGCONTROLTHREAD runs on the embedded *Input* part and hands over received middleware messages containing control settings to the `Logger` class for controlling the Monitoring F/W at runtime (cf. to Section 16.3.2:

- inherits from `Subscriber` (cf. to Section 5.2.1.2) and "listens" to `LogControlItems` being published on the `LogControlTopic`
- collects control messages in a buffer (`SyncFifo`, see Section 5.2.1.2) which can be adjusted using a macro or using CMAKE (cf. to Section 16.3.1)

SIMPLELOGCONTROLLERTHREAD On the non-embedded *Output* part this class is a very simple implementation of a control possibility of the F/W's behavior. It accepts inputs from the user on the console and hands these inputs over to `LogOutput` for forwarding them to the embedded part.

`LogControlItem` contains the Control Data for Uplink

LOGCONTROLITEM contains the log control data to be sent over the RODOS middleware to the embedded part of the Monitoring F/W.

While the class diagram in Figure 16.2 was very useful in order to explain the different classes existing within the F/W, the dynamic behavior of our prototype is best shown by having a look at two sequence-diagrams[50]: Figure 16.3 shows the dynamic behavior for the sending of `LogItems`, while Figure 16.4 does the same for `LogControlItems`.

50 Cf. e.g. to [84] for a general description of "reading" sequence diagrams and how to interpret the used symbols.

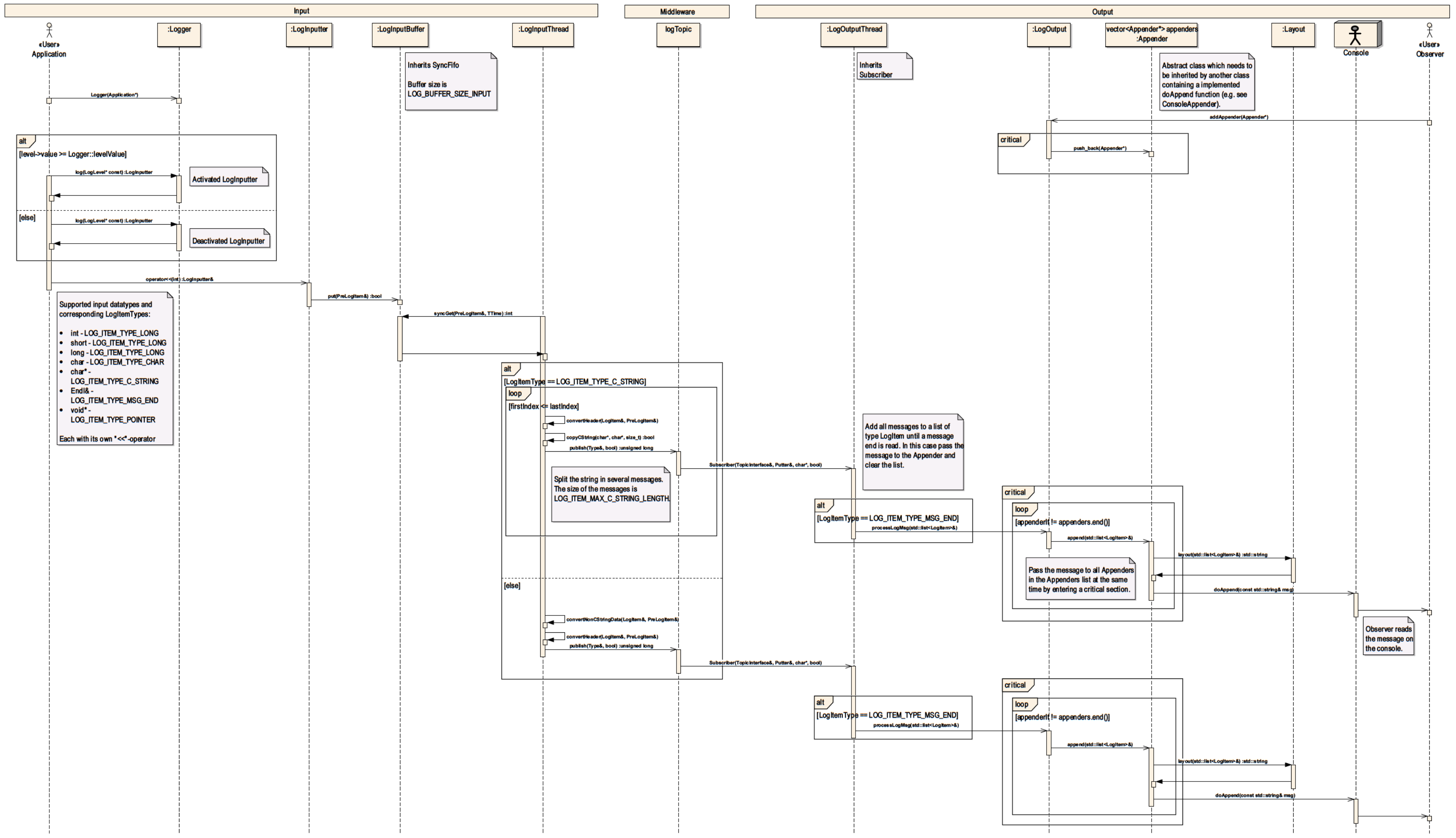

Input
Middleware
Output
«User» Application
:Logger
:LogInputter
:LogInputBuffer
:LogInputThread
logTopic
:LogOutputThread
:LogOutput
vector<Appender*> appenders :Appender
:Layout
Console
«User» Observer
Inherits SyncFifo
Buffer size is LOG_BUFFER_SIZE_INPUT
Inherits Subscriber
Abstract class which needs to be inherited by another class containing a implemented doAppend function (e.g. see ConsoleAppender).
Logger(Application*)
addAppender(Appender*)
critical
push_back(Appender*)
alt
[level->value >= Logger::levelValue]
log(LogLevel* const) :LogInputter
Activated LogInputter
[else]
log(LogLevel* const) :LogInputter
Deactivated LogInputter
operator<<(int) :LogInputter&
put(PreLogItem&) :bool
syncGet(PreLogItem&, TTime) :int
Supported input datatypes and corresponding LogItemTypes:
• int - LOG_ITEM_TYPE_LONG
• short - LOG_ITEM_TYPE_LONG
• long - LOG_ITEM_TYPE_LONG
• char - LOG_ITEM_TYPE_CHAR
• char* - LOG_ITEM_TYPE_C_STRING
• Endl& - LOG_ITEM_TYPE_MSG_END
• void* - LOG_ITEM_TYPE_POINTER
Each with its own "<<"-operator
alt
[LogItemType == LOG_ITEM_TYPE_C_STRING]
loop
[firstIndex <= lastIndex]
convertHeader(LogItem&, PreLogItem&)
copyCString(char*, char*, size_t) :bool
publish(Type&, bool) :unsigned long
Split the string in several messages. The size of the messages is LOG_ITEM_MAX_C_STRING_LENGTH.
[else]
convertNonCStringData(LogItem&, PreLogItem&)
convertHeader(LogItem&, PreLogItem&)
publish(Type&, bool) :unsigned long
Subscriber(TopicInterface&, Putter&, char*, bool)
Subscriber(TopicInterface&, Putter&, char*, bool)
Add all messages to a list of type LogItem until a message end is read. In this case pass the message to the Appender and clear the list.
alt
[LogItemType == LOG_ITEM_TYPE_MSG_END]
processLogMsg(std::list<LogItem>&)
alt
[LogItemType == LOG_ITEM_TYPE_MSG_END]
processLogMsg(std::list<LogItem>&)
critical
loop
[appenderIt != appenders.end()]
append(std::list<LogItem>&)
layout(std::list<LogItem>&) :std::string
doAppend(const std::string& msg)
Pass the message to all Appenders in the Appenders list at the same time by entering a critical section.
Observer reads the message on the console.
critical
loop
[appenderIt != appenders.end()]
append(std::list<LogItem>&)
layout(std::list<LogItem>&) :std::string
doAppend(const std::string& msg)

Figure 16.3: Sequence Diagram of the Monitoring F/W Prototype - showing the dynamic Behavior for the Sending of `LogItems`

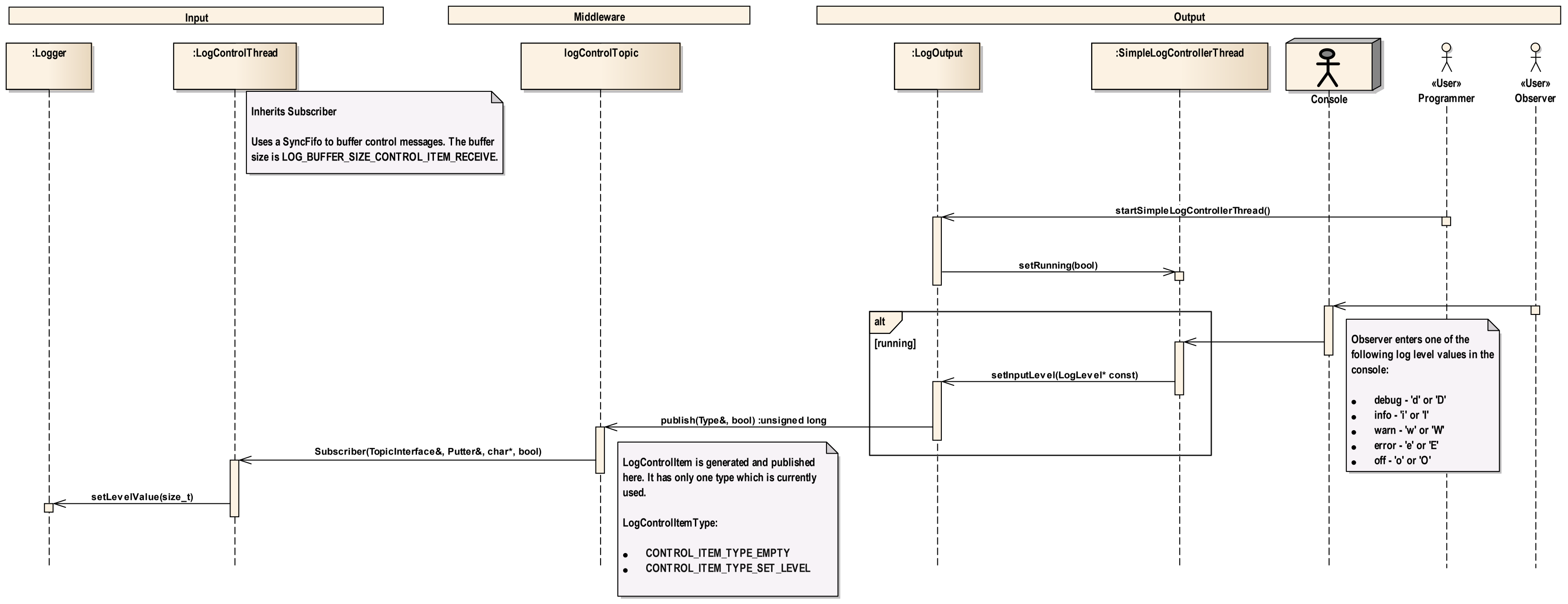

Input
Middleware
Output
:Logger
:LogControlThread
logControlTopic
:LogOutput
:SimpleLogControllerThread
Console
«User» Programmer
«User» Observer
Inherits Subscriber
Uses a SyncFifo to buffer control messages. The buffer size is LOG_BUFFER_SIZE_CONTROL_ITEM_RECEIVE.
startSimpleLogControllerThread()
setRunning(bool)
alt
[running]
Observer enters one of the following log level values in the console:
debug - 'd' or 'D'
info - 'i' or 'I'
warn - 'w' or 'W'
error - 'e' or 'E'
off - 'o' or 'O'
setInputLevel(LogLevel* const)
publish(Type&, bool) :unsigned long
Subscriber(TopicInterface&, Putter&, char*, bool)
setLevelValue(size_t)
LogControlItem is generated and published here. It has only one type which is currently used.
LogControlItemType:
CONTROL_ITEM_TYPE_EMPTY
CONTROL_ITEM_TYPE_SET_LEVEL

Figure 16.4: Sequence Diagram of the Monitoring F/W Prototype - showing the dynamic Behavior for the Sending of `LogControlItems`

16.3 CONFIGURATION

Monitoring of embedded S/W applications is a typical cross-sectional task. This is reflected by the fact, that the monitoring functionality is typically not only concentrated in one central application, but aspects of it are scattered about all other S/W components as well (similar to e.g. the fault tolerance functionality [1]).

Scalability of Performance

As a consequence, a badly performing monitoring F/W would immediately result in a misbehavior of the whole S/W image: in a real-time system even a small time delay can result in a failure of the whole system if a deadline is missed (cf. to Section 5.2.1.1). Therefore, the Monitoring F/W must not only be tailored by configuration to the limited resources of the S/C avionic (like it is true for all constituents of the embedded S/W), but also tailored in terms of not influencing the other parts of the S/W system (this can be verified by a schedulability analysis as described in Section 5.2.4). The scaling of the Monitoring F/W in terms of performance is mostly done at compile-time.

Scalability of Functionality

Besides the design goal of being as less intrusive as possible – which is common for all embedded S/W and their supplementary libraries – we put strong emphasis on a novelty in this field of work: the functional scaling of the Monitoring F/W during runtime. The main reason behind this design driver is quite obvious and has its roots in the initial motivation of the universal tool for the human body introduced in the IDEA & VISION Chapter 2: in case of an occurring failure (e.g. in the critical operational phase of the satellite) FDIR-specific actions need as much detailed information about the subsystem(s) of the satellite where the error is supposed to come from. This would be an easy thing to do if we had unlimited resources available on board which would allow a very detailed in-depth recording of all relevant information. But this is most of the time not the case, as the majority of space missions are not only restricted in terms of processor performance, bus bandwidth and storage capacities, but also limited in the bandwidth which would be necessary to transfer all the needed monitoring data to ground[51].

In order to solve this problem, the Monitoring F/W was designed in a way that it is possible even at runtime to change its configuration. Foreseen properties to be adaptable are

- the depth of information (e.g. gathering of monitoring information from one application, one thread or even one specific method),
- the frequency of recording the monitoring data,
- the criticality of the data to be monitored

51 It should be noted that even if the mission design has the capabilities of solve these technical restrictions, the free resources are used in order to fulfill important tasks like onboard data processing or transferring larger chunks of payload data to ground.

- the resolution of analog or digital values,
- and many more.

The scaling of the monitoring F/W in terms of functionality is mostly done at run-time.

16.3.1 *Configuration at Compile-Time*

Scaling the Monitoring F/W in terms of performance at compile-time is integrated in the build process of the embedded S/W (cf. to the *Boot Image Creation* paragraph in Section 5.2.3). It is often done by defining appropriate configuration macros which can take place in two ways:

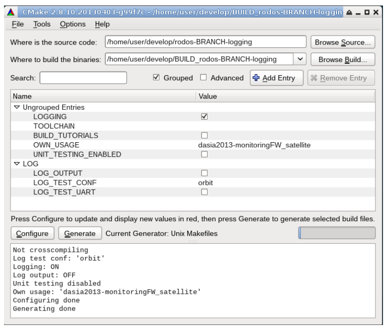
A single header file (usually called `param.h`) is used in order to define the macros within conditional groups. As an example the `param.h` file for the Monitoring F/W prototype is given in Listing 16.1, it contains the values used for testing the monitoring F/W in the first two uses case (cf. to Sections 18.2 and 18.3). Using the preprocessor's `#ifndef` directive ensures, that the macros will be included in the output of the preprocessor if and only if they are not already defined.

Listing 16.1: `param.h` File of the Prototype of the Monitoring F/W

```c
/*
 * param.h
 */

#ifndef _LOG_PARAM_H_
#define _LOG_PARAM_H_

// set default values for monitoring framework prototype

#ifndef LOG_APP_ID
#define LOG_APP_ID                                    424298
#endif

#ifndef LOG_THREAD_PRIO_CONTROL
#define LOG_THREAD_PRIO_CONTROL                       50
#endif

#ifndef LOG_THREAD_PRIO_INPUT
#define LOG_THREAD_PRIO_INPUT                         100
#endif

#ifndef LOG_ITEM_MAX_C_STRING_LENGTH
#define LOG_ITEM_MAX_C_STRING_LENGTH                  16
#endif

#ifndef LOG_ITEM_MAX_APP_NAME_LENGTH
#define LOG_ITEM_MAX_APP_NAME_LENGTH                  16
#endif

#ifndef LOG_ITEM_MAX_THREAD_NAME_LENGTH
#define LOG_ITEM_MAX_THREAD_NAME_LENGTH               16
#endif

#ifndef LOG_BUFFER_SIZE_INPUT
#define LOG_BUFFER_SIZE_INPUT                         100
#endif

#ifndef LOG_BUFFER_SIZE_CONTROL_ITEM_RECEIVE
#define LOG_BUFFER_SIZE_CONTROL_ITEM_RECEIVE          20
#endif

#ifndef LOG_LEVEL_DEBUG_VALUE
#define LOG_LEVEL_DEBUG_VALUE                         10
#endif

#ifndef LOG_LEVEL_INFO_VALUE
#define LOG_LEVEL_INFO_VALUE                          20
#endif

#ifndef LOG_LEVEL_WARN_VALUE
#define LOG_LEVEL_WARN_VALUE                          30
#endif

#ifndef LOG_LEVEL_ERROR_VALUE
#define LOG_LEVEL_ERROR_VALUE                         40
#endif

#ifndef LOG_LEVEL_OFF_VALUE
#define LOG_LEVEL_OFF_VALUE                           50
#endif

#ifndef LOG_META_LEVEL_VALUE
#define LOG_META_LEVEL_VALUE                          50
#endif

#ifndef LOG_BUFFER_SIZE_ITEM_RECEIVE
#define LOG_BUFFER_SIZE_ITEM_RECEIVE                  100
#endif

#ifndef LOG_APP_ID_OUTPUT
#define LOG_APP_ID_OUTPUT                             424299
#endif

#ifndef LOG_THREAD_PRIO_SIMPLE_CONTROLLER
#define LOG_THREAD_PRIO_SIMPLE_CONTROLLER             50
#endif

#ifndef LOG_THREAD_PRIO_OUTPUT
#define LOG_THREAD_PRIO_OUTPUT                        100
#endif

#endif /* _LOG_PARAM_H_ */
```

A more comfortable way of configuring the parameters of the F/W in the process of compilation is to use a universal build tool like

CMAKE (cf. to [48] for further details). CMAKE is cross-platform and runs on quite a number of operating systems. It uses simple platform and compiler independent configuration files and is therefore suitable for almost every development host computer (cf. to the *development environment* paragraph in Section 5.2.3). It comes with a GUI which offers different options of how many of the available configuration details are visible to the user.



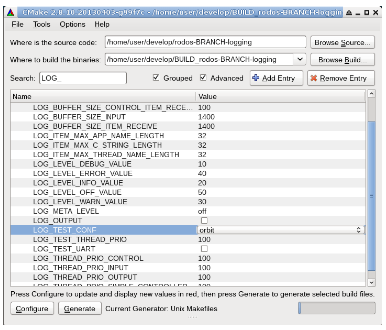
Where is the source code: /home/user/develop/rodos-BRANCH-logging
Where to build the binaries: /home/user/develop/BUILD_rodos-BRANCH-logging
Search:
☑ Grouped ☐ Advanced

Name	Value
▽ Ungrouped Entries	
LOGGING	☑
TOOLCHAIN	
BUILD_TUTORIALS	☐
OWN_USAGE	dasia2013-monitoringFW_satellite
UNIT_TESTING_ENABLED	☐
▽ LOG	
LOG_OUTPUT	☐
LOG_TEST_CONF	orbit
LOG_TEST_UART	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.
Current Generator: Unix Makefiles

```
Not crosscompiling
Log test conf: 'orbit'
Logging: ON
Log output: OFF
Unit testing disabled
Own usage: 'dasia2013-monitoringFW_satellite'
Configuring done
Generating done
```

(a) Simple View



Where is the source code: /home/user/develop/rodos-BRANCH-logging
Where to build the binaries: /home/user/develop/BUILD_rodos-BRANCH-logging
Search: LOG_
☑ Grouped ☑ Advanced

Name	Value
LOG_BUFFER_SIZE_CONTROL_ITEM_RECE...	100
LOG_BUFFER_SIZE_INPUT	1400
LOG_BUFFER_SIZE_ITEM_RECEIVE	1400
LOG_ITEM_MAX_APP_NAME_LENGTH	32
LOG_ITEM_MAX_C_STRING_LENGTH	32
LOG_ITEM_MAX_THREAD_NAME_LENGTH	32
LOG_LEVEL_DEBUG_VALUE	10
LOG_LEVEL_ERROR_VALUE	40
LOG_LEVEL_INFO_VALUE	20
LOG_LEVEL_OFF_VALUE	50
LOG_LEVEL_WARN_VALUE	30
LOG_META_LEVEL	off
LOG_OUTPUT	☐
LOG_TEST_CONF	orbit
LOG_TEST_THREAD_PRIO	100
LOG_TEST_UART	☐
LOG_THREAD_PRIO_CONTROL	100
LOG_THREAD_PRIO_INPUT	100
LOG_THREAD_PRIO_OUTPUT	100
LOG_THREAD_PRIO_SIMPLE_CONTROLLER	100

Press Configure to update and display new values in red, then press Generate to generate selected build files.
Current Generator: Unix Makefiles

(b) Advanced View

Figure 16.5: The different CMAKE Views for the Prototype of the Monitoring F/W

For the prototype of the Logging F/W, this feature was used in order to define two configuration views depicted in Figure 16.5:
A screenshot of the *simple view* can be seen in Figure 16.5a. Here only very basic options like turning on or off the Monitoring F/W and a general test configuration (e.g. "orbit" or "ground") containing predefined values can be chosen. On the other hand, the *advanced view* visible in Figure 16.5b lets the user configure the F/W in every possible detail. For example, the priorities of the threads described in the implementation Section 16.2 can be set here (e.g. `LOG_THREAD_-PRIO_INPUT` for the `LogInputThread` priority).

This is of particular importance for "trimming" the F/W later on when performing the schedulability analysis during the integration phase with other (S/W) subsystems (see Section 5.2.4). Though, the initial setup regarding priorities of threads within an embedded application can be based upon the findings from the priority settings experiments of the F/W prototype (cf. to Section 18). These experiments also triggered the initial priority setup of the flight version of the Monitoring F/W (cf. to Section 17).

16.3.2 *Configuration at Runtime*

As it was explained at the beginning of this chapter, it shall be possible to configure the F/W even at runtime to perform some kind of functional scaling. For the prototype it was decided to concentrate on the runtime manipulation of the criticality of information to be recorded and transfered to the development host, the EGSE equipment or the groundstation. For this purpose the adjustment of the `LogLevels` of the monitoring `F/W` was introduced (cf. to Section 16.2).

Initially – at compile-time – the global log levels are defined by the following values:

- LOG_LEVEL_DEBUG_VALUE (integer value 10)
- LOG_LEVEL_INFO_VALUE (integer value 20)
- LOG_LEVEL_WARN_VALUE (integer value 30)
- LOG_LEVEL_ERROR_VALUE (integer value 40)
- LOG_LEVEL_OFF_VALUE (integer value 50)

The prototype offers the possibility to add further, more detailed log level values later on.

The user is then able to adjust the `LogLevel` also at runtime: the changed value is sent inside a `LogControlItem` over the RODOS middleware using the `LogControlTopic`-type `CONTROL_ITEM_TYPE_SET_-LEVEL`. On the embedded *Input* part these messages are received by the `LogControlThread` which in turn invokes the corresponding `Logger`-method `setLevelValue()` (cf. to the configuration sequence diagram in Figure 16.4.

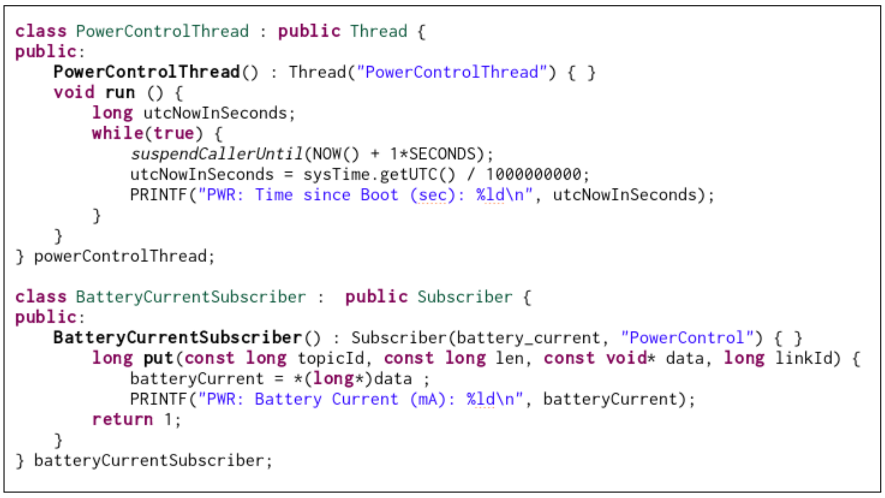
After this kind of runtime re-configuration the `Logger` behaves different in terms of what kind of monitoring messages are sent to the *Output* part, dependent on the new log level: every time an application uses the `log`-method in order to send out monitoring information, the global log value is compared with the one handed over by the `log`-method. If the corresponding integer value is smaller than the global one, `log` returns an deactivated `LogInputter` which discards these input messages (cf. to the corresponding section in the upper left corner of Figure 16.3).

16.4 USE CASES

16.4.1 *Use Case 1: PRINTF-Replacement (Simple)*

16.4.1.1 *Use Case with normal PRINTF*

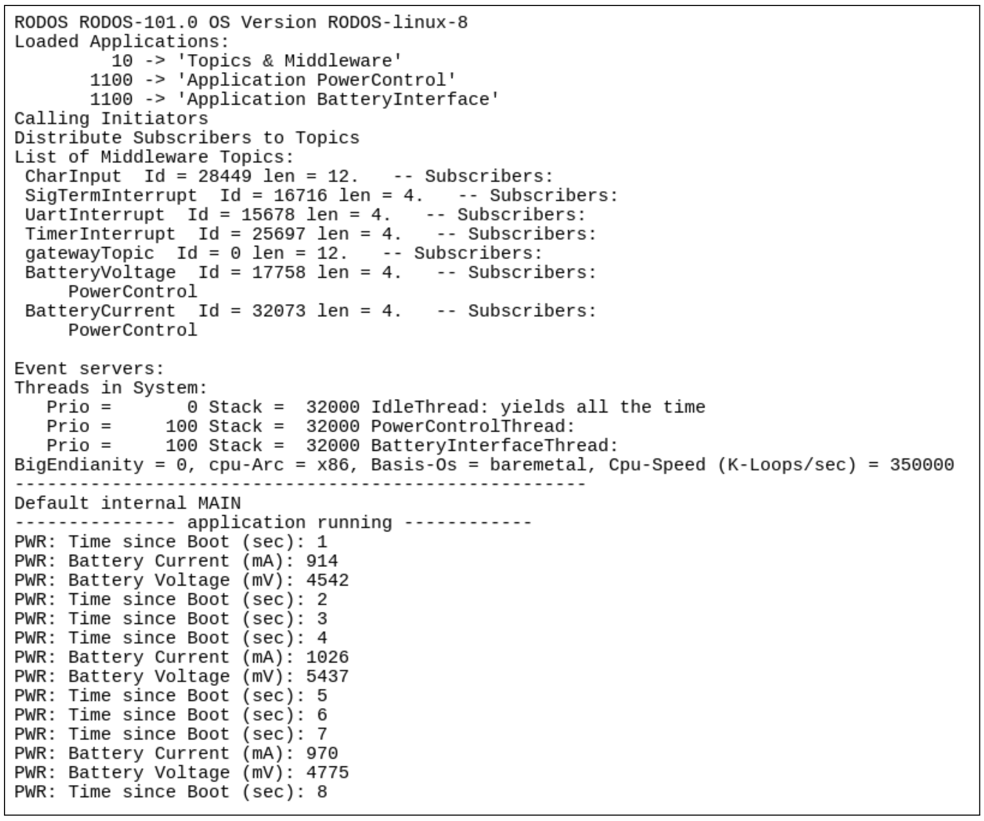
In order to demonstrate the flexibility of the new Monitoring F/W a minimal working example has been prepared. It has the primary goal to configure the Monitoring F/W in a way that it can be used similar to printf debugging – not only in terms of usability but also regarding the functional behavior. For the minimal working example we use the same scenario as depicted in Figure 12.1 where we only have two applications running on top of RODOS on the side of the satellite: a `BatteryInterface`-application which serves as an interface to the real H/W, and the `PowerControl`-Application which is responsible for controlling the power management system of the satellite. In this case, `PowerControl` subscribes itself to the middleware topics `BatteryCurrent` and `BatteryVoltage` which are published to the system by the `BatteryInterface`. In order to debug the system and ensure, that all information is correctly received by `PowerControl`, the application developer uses RODOS `PRINTF` statements like shown in Figure 16.6: every three seconds the `BatteryCurrentSubscriber`-thread prints out the current and voltage of the battery, and – as an additional debug information – the `PowerControlThread` is printing every second the current onboard time, counted from the boot up of the satellite. The output is formatted in a similar way like it was done in the TET-1 project (cf. to Listing 12.2).



```
class PowerControlThread : public Thread {
public:
    PowerControlThread() : Thread("PowerControlThread") { }
    void run () {
        long utcNowInSeconds;
        while(true) {
            suspendCallerUntil(NOW() + 1*SECONDS);
            utcNowInSeconds = sysTime.getUTC() / 1000000000;
            PRINTF("PWR: Time since Boot (sec): %ld\n", utcNowInSeconds);
        }
    }
} powerControlThread;

class BatteryCurrentSubscriber :  public Subscriber {
public:
    BatteryCurrentSubscriber() : Subscriber(battery_current, "PowerControl") { }
        long put(const long topicId, const long len, const void* data, long linkId) {
            batteryCurrent = *(long*)data ;
            PRINTF("PWR: Battery Current (mA): %ld\n", batteryCurrent);
        return 1;
    }
} batteryCurrentSubscriber;
```

Figure 16.6: Sourcecode of the `PowerControl`-Application using `PRINTF`-Statements

After compiling these two applications and linking them together with the RODOS library, the final boot image is deployed to the target H/W. Normally the Linux operating system is used for simulating the



```
RODOS RODOS-101.0 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
       1100 -> 'Application PowerControl'
       1100 -> 'Application BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =     100 Stack =  32000 PowerControlThread:
   Prio =     100 Stack =  32000 BatteryInterfaceThread:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-------------------------------------------------
Default internal MAIN
--------------- application running ------------
PWR: Time since Boot (sec): 1
PWR: Battery Current (mA): 914
PWR: Battery Voltage (mV): 4542
PWR: Time since Boot (sec): 2
PWR: Time since Boot (sec): 3
PWR: Time since Boot (sec): 4
PWR: Battery Current (mA): 1026
PWR: Battery Voltage (mV): 5437
PWR: Time since Boot (sec): 5
PWR: Time since Boot (sec): 6
PWR: Time since Boot (sec): 7
PWR: Battery Current (mA): 970
PWR: Battery Voltage (mV): 4775
PWR: Time since Boot (sec): 8
```

Figure 16.7: Console Output of the `PowerControl`-Application using `PRINTF`-Statements

behavior of the embedded H/W, as Linux is also the usual development platform for RODOS. However, for the test and evaluation of the F/W prototype real (not simulated) H/W is used. Deployment takes place by connecting to the embedded system using a UART-interface and transferring and starting the boot image. Afterwards the output shown in Figure 16.7 can be observed on the local Linux terminal of the developer.

16.4.1.2 *Use Case with Monitoring Framework*

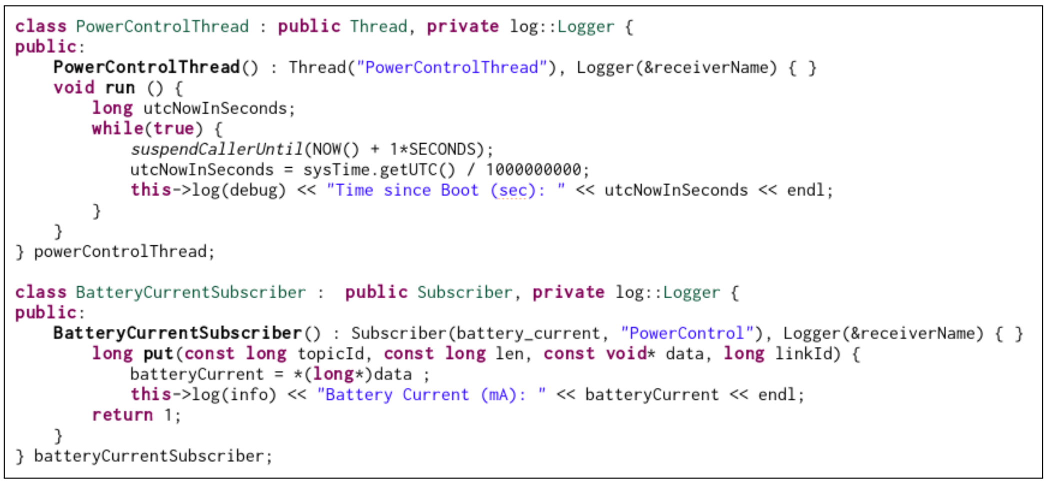
In order to simulate the printf debugging by using the Monitoring F/W we have to change the setup used in the example scenario described above: the information provided to the user (in our case: the S/W developer) by printing them to the terminal using RODOS `PRINTF` statements will now have to be provided by the Monitoring F/W. The resulting setup is depicted in Figure 15.3, in order to easily identify the newly added components and their roles taken over (Logger, Appender, Layout), the same colors like in Figure 14.2 are used (green for components related to the core of the Monitoring F/W, yellow for supplementary components like appenders and layouts).

The scenario on the embedded target platform was extended in three ways:
First, the two additional logging topics introduced to the middleware can be seen in Figure 15.3 (please cf. also to description of the constituents of the F/W prototype in Chapter 16.1). The application `PowerControl` is using the `log`-Topic in order to publish not only its own status information, but also – being a subscriber to the topics `BatteryCurrent` and `BatteryVoltage` – the information coming from the battery interface. The code section in Figure 16.8 shows the `PowerControl`-application, where the calls to `PRINTF` described in Section 16.4.1.1 are replaced by calls to the Monitoring F/W.

Components of the Monitoring F/W are needed on both sides: the embedded Target-Platform and the development Host-Platform

NOTE:

In Figure 15.3 it is depicted that the `PowerControl` application is publishing log messages using the normal publish/subscribe mechanism of the RODOS RTOS (e.g. by using the `puplish()`-method). But a comparison with the code Snippet 16.8 reveals that within the actual implementation the `log`-method of the Monitoring F/W is invoked instead. The fact that within the figure the applications appear as a direct publisher of log messages is for the sake of simplicity. The same simplification will be used in the schematic overviews within the Figures 16.13 and 16.18 of use case 2 and 3.



```
class PowerControlThread : public Thread, private log::Logger {
public:
    PowerControlThread() : Thread("PowerControlThread"), Logger(&receiverName) { }
    void run () {
        long utcNowInSeconds;
        while(true) {
            suspendCallerUntil(NOW() + 1*SECONDS);
            utcNowInSeconds = sysTime.getUTC() / 1000000000;
            this->log(debug) << "Time since Boot (sec): " << utcNowInSeconds << endl;
        }
    }
} powerControlThread;

class BatteryCurrentSubscriber :  public Subscriber, private log::Logger {
public:
    BatteryCurrentSubscriber() : Subscriber(battery_current, "PowerControl"), Logger(&receiverName) { }
        long put(const long topicId, const long len, const void* data, long linkId) {
            batteryCurrent = *(long*)data ;
            this->log(info) << "Battery Current (mA): " << batteryCurrent << endl;
        return 1;
    }
} batteryCurrentSubscriber;
```

Figure 16.8: Source Code of the `PowerControl`-Application using the `log`-Method of the Monitoring F/W prototype

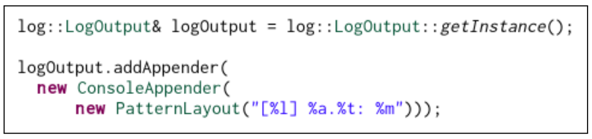
The `logApp` as the subscribing application collects this information, and is – on the other hand – also able to configure and control the `PowerControl` application by utilizing the `logControl` topic to which the `PowerControl` application is subscribed.
Second, to be able to send middleware information to external systems, the embedded satellite S/W image was configured with a gateway which is connected to the middleware. It is capable to send the

requested middleware topics over the network to other RODOS instances. In this first use case this is the host Personal Computer (PC) we use in order to debug the embedded system. Having these two gateways connected via a UART connection[52], `log` and `logControl` topics can be exchanged between both systems.
Third, on the host PC we need a similar setup as on the target computer of the satellite in order to receive and print out the received monitoring information and – if necessary – to configure the monitoring F/W on the embedded target computer. Again, in Figure 15.3 we can see two additional logging topics, which are – in this case – the only topics needed for debugging. The `logOutputApp` as the subscriber to the `log` topic receives the monitoring information which contains the status information from `PowerControl` and `BatteryInterface`. The `logOutputApp` is then using a `ConsoleAppender` configured with a `PatternLayout` in order to print out the information to the console of the application developer. Figure 16.9 shows the corresponding code section, whereas in the configuration of the `PatternLayout` "l" is the log-level, "a" & "t" the application and thread the information is coming from and "m" the content of the message [see 63, sec. 4.2.1.11 and 4.2.1.12].

NOTE:

On the Host Platform Coding Standards may be violated if necessary.

> Careful readers surely noted, that in this code snippet the `new`-operator was used for creating objects and dynamically allocate memory. In most embedded systems – especially safety-critical once – this is strictly forbidden and explicitly stated in the applicable coding standards for the specific domain[53]. Nevertheless, in this case the created image is only dedicated to run on the debugging host of the S/W developer which is normally a desktop computer running Linux oder Windows. Therefore this violation is acceptable within these implementation parts of the monitoring F/W.



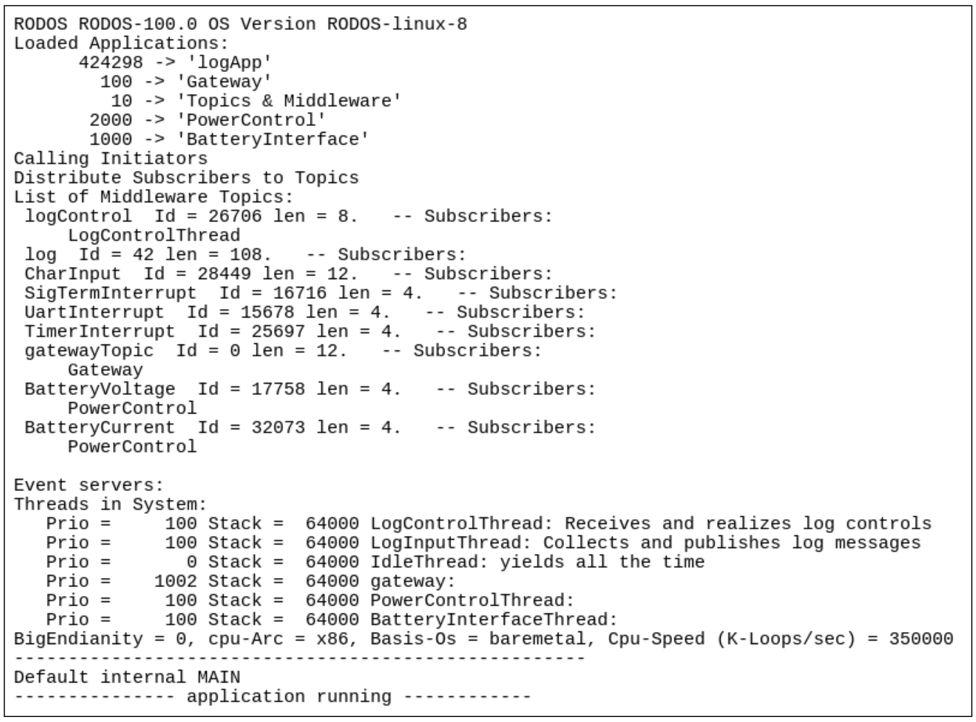
```
log::LogOutput& logOutput = log::LogOutput::getInstance();

logOutput.addAppender(
  new ConsoleAppender(
      new PatternLayout("[%l] %a.%t: %m")));
```

Figure 16.9: Source Code Section showing the Configuration of the Monitoring Output

After deployment to the target H/W, in order to control the correct execution of the boot image we connect again to the embedded system using a UART-interface and start the boot image. The output

52 TCP/IP would also be possible.
53 To give some examples, for the military aircraft domain cf. to JSF rule AV 206 [11], for the automotive industry this is specified in the MISRA C standard, rule 118 [53]

shown in Figure 16.10 looks like expected: the RODOS RTOS starts and prints out its normal startup status messages using `PRINTF` (until "application running" appears as the last message on the screen). Then the output stops, because all following messages are sent over the middleware to the host computer. Figure 16.11 shows what is printed on the console of the host computer. As expected, also here the normal RODOS startup prints are shown, but followed by the received middleware messages coming from the embedded target computer. The messages are formated according to the configuration of the `PatternLayout` shown in Figure 16.9.

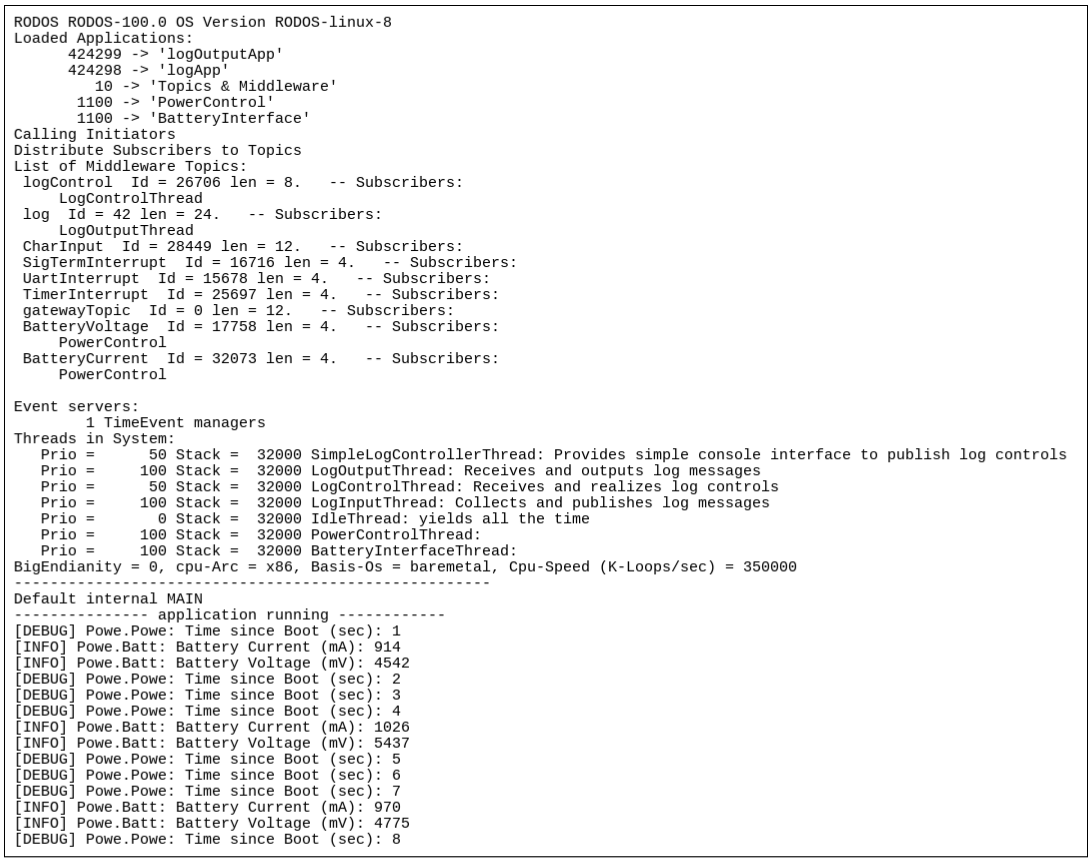


```
RODOS RODOS-100.0 OS Version RODOS-linux-8
Loaded Applications:
      424298 -> 'logApp'
        100 -> 'Gateway'
         10 -> 'Topics & Middleware'
       2000 -> 'PowerControl'
       1000 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 108.   -- Subscribers:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =     100 Stack =  64000 LogControlThread: Receives and realizes log controls
   Prio =     100 Stack =  64000 LogInputThread: Collects and publishes log messages
   Prio =       0 Stack =  64000 IdleThread: yields all the time
   Prio =    1002 Stack =  64000 gateway:
   Prio =     100 Stack =  64000 PowerControlThread:
   Prio =     100 Stack =  64000 BatteryInterfaceThread:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
```

Figure 16.10: Console Output of the `PowerControl`-application running on top of RODOS on the Embedded System

16.4.2 *Use Case 2: PRINTF-Replacement (Advanced)*

The next use case for testing the functionality of the Monitoring F/W is supposed to be a more realistic one. The first use case described in Section 16.4.1 is very useful in order to demonstrate the overall functionality of the first prototype and run firsts test on it (cf. to Section 18.2. But it is not a very realistic scenario in the sense, that the `Logger` component as the main constituent of the F/W is most likely to run in concurrency to many other S/W applications. These will not only consume additional resources, but also put more stress on the Monitoring F/W by sending a lot more monitoring messages than it was the case in use case 1.



```
RODOS RODOS-100.0 OS Version RODOS-linux-8
Loaded Applications:
      424299 -> 'logOutputApp'
      424298 -> 'logApp'
         10 -> 'Topics & Middleware'
       1100 -> 'PowerControl'
       1100 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 24.   -- Subscribers:
     LogOutputThread
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
        1 TimeEvent managers
Threads in System:
   Prio =      50 Stack =  32000 SimpleLogControllerThread: Provides simple console interface to publish log controls
   Prio =     100 Stack =  32000 LogOutputThread: Receives and outputs log messages
   Prio =      50 Stack =  32000 LogControlThread: Receives and realizes log controls
   Prio =     100 Stack =  32000 LogInputThread: Collects and publishes log messages
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =     100 Stack =  32000 PowerControlThread:
   Prio =     100 Stack =  32000 BatteryInterfaceThread:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
[DEBUG] Powe.Powe: Time since Boot (sec): 1
[INFO] Powe.Batt: Battery Current (mA): 914
[INFO] Powe.Batt: Battery Voltage (mV): 4542
[DEBUG] Powe.Powe: Time since Boot (sec): 2
[DEBUG] Powe.Powe: Time since Boot (sec): 3
[DEBUG] Powe.Powe: Time since Boot (sec): 4
[INFO] Powe.Batt: Battery Current (mA): 1026
[INFO] Powe.Batt: Battery Voltage (mV): 5437
[DEBUG] Powe.Powe: Time since Boot (sec): 5
[DEBUG] Powe.Powe: Time since Boot (sec): 6
[DEBUG] Powe.Powe: Time since Boot (sec): 7
[INFO] Powe.Batt: Battery Current (mA): 970
[INFO] Powe.Batt: Battery Voltage (mV): 4775
[DEBUG] Powe.Powe: Time since Boot (sec): 8
```

Figure 16.11: Console output of the Monitoring F/W running on the Host Computer of the S/W Developer

The configuration of the Monitoring F/W itself stays the same as in use case 1, reading in the values from the `param.h` file (cf. to Listing 16.1). For the newly added applications `Main`, `AOCS`, `Thermal-Control` and `PowerControl` the priorities are set to the same value (100) as for the application threads of use case 1, `PowerControlThread` and `BatteryInterfaceThread` (these priorities can be seen e.g. in Listing 18.3). However – as separate tests have shown – the main threads of the Monitoring F/W `LogOutputThread` and `LogInputThread` have to be set to a higher priority value (1000) in order to work correctly (cf. to Paragraph Setting Thread Priorities on Page 158 about further details of priority settings in use case 2).

16.4.2.1 *Use Case with normal PRINTF*

In order to create such a realistic scenario we will transform the example of use case 1 depicted in Figure 15.3 into a more demanding one by adding additional applications, also letting them produce additional load on the processor and the Monitoring F/W. This is mainly done by generation of random numbers[54] within a defined range ($\rightarrow$ increasing processor load) or by the sending additional messages in the case of threshold-violations ($\rightarrow$ increasing network load).

Like drafted in Figure 16.12, the following applications and topics were added to this use case:

MAIN Can be understood as an equivalent of the startup routine of a satellite's boot image, in this use case responsible for providing the time since boot.

TEMPINTERFACE A simulated interface to five temperature sensors.

THERMALCONTROL An application responsible for the management of the satellite's thermal control system. In order to simulate some load, it reports a severe warning if one of the temperature values rises over a certain threshold.

AOCS A simple application serving as an AOCS. Simulates the starting or stopping of three thrusters and produces additional processor load by using the RODOS-internal random number generator.

In addition, the `PowerControl`-application already used in the simple example was modified for producing additional processor load: the consumed power is calculated by using the voltage and current values from the `BatteryInterface` which were received by the application over the RODOS middleware.

As before, the RODOS middleware is utilized in order to exchange data between the applications running on the embedded system. Therefore, additional topics for all five temperature sensors were added to the scenario. Again – like described in Section 12.1.1 – all applications make use of the debug-methods provided by `debug.h`.

16.4.2.2 *Use Case with Monitoring Framework*

Like it was done with the simple scenario (cf. to Section 16.4.1), we will take the `PRINTF`-based version of this use case from Section 16.4.2.1 and exchange the `PRINTF`-debugging technique by using the prototype of the Monitoring F/W. The result can be seen in Figure 16.13. While the setup on the host stays the same, the embedded side was complemented by the already known topics `log` and `logControl` and the central `Logger` component. All satellite applications are providing logging information to the F/W by publishing on the `log`-topic.

54 Generation of random numbers is a fairly common method in order to set up a simulation for the system under test (cf. to [37, pp. 437]). For the prototype use cases we will use the RODOS-internal random number generator which uses an algorithm described in [51].

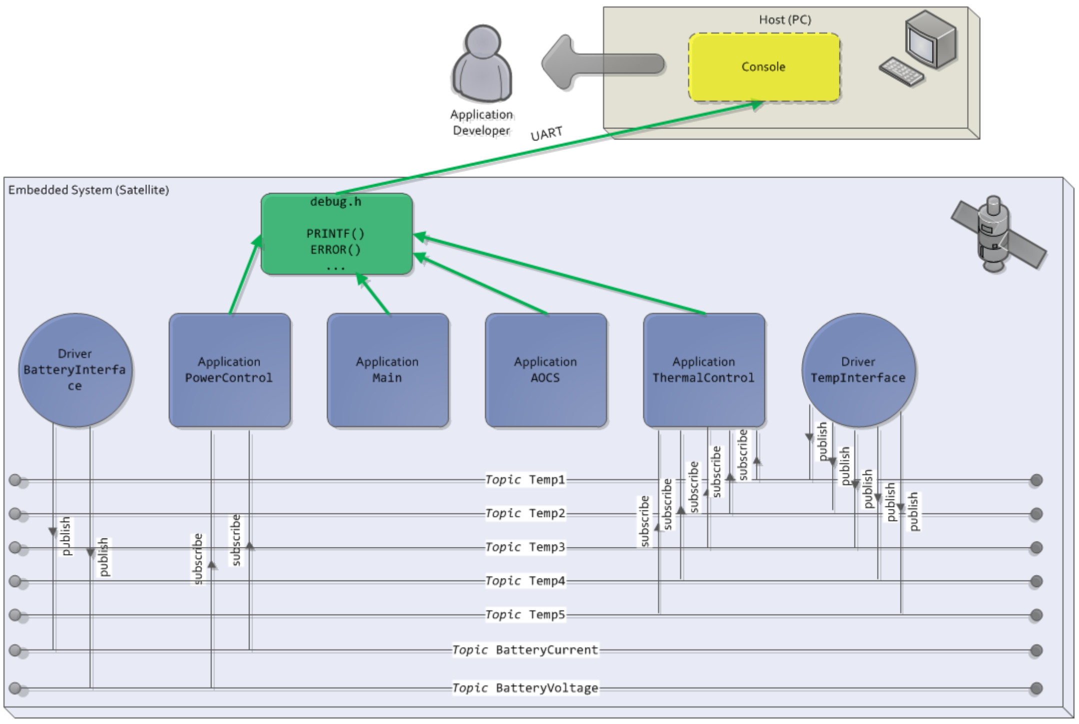


Figure 16.12: RODOS debugging using PRINTF statements (Advanced Scenario)

16.4.3 *Use Case 3: FloatingSat (Real-World Scenario)*

As the use cases 1 and 2 can be understood as breadboard tests of the Monitoring F/W prototype, one of the goals was to supplement these tests with a *real-world* scenario. Unfortunately there is currently no suitable mission with significant involvement by DLR which – in addition – is driven by a RODOS-only S/W environment. Besides the mission support from groundstations (cf. to the *Evaluation* Section 20.2) also a test environment with an EGSE using RODOS on the non-embedded side *and* onboard the satellite is currently not feasible[55].

Therefore the laboratory experiment *FloatingSat* was chosen as an experiment on earth which comes as close to a real space mission as possible. The FloatingSat is one of many experiments which were run by the *DLR_School_Lab* facility at the DLR site in Bremen[56]. Its main purpose is to demonstrate – on a small scale – which techniques can

55 The most promising candidates have been DLR's MASCOT [38] and GOSSAMER [81] missions: within the MASCOT mission RODOS was replaced in a late project phase by another RTOS, whereas the GOSSAMER mission is designed as such that RODOS is controlling only some peripheral parts of the satellite.

56 Cf. to the corresponding homepage for further details: `http://www.dlr.de/schoollab/desktopdefault.aspx/tabid-7612/`

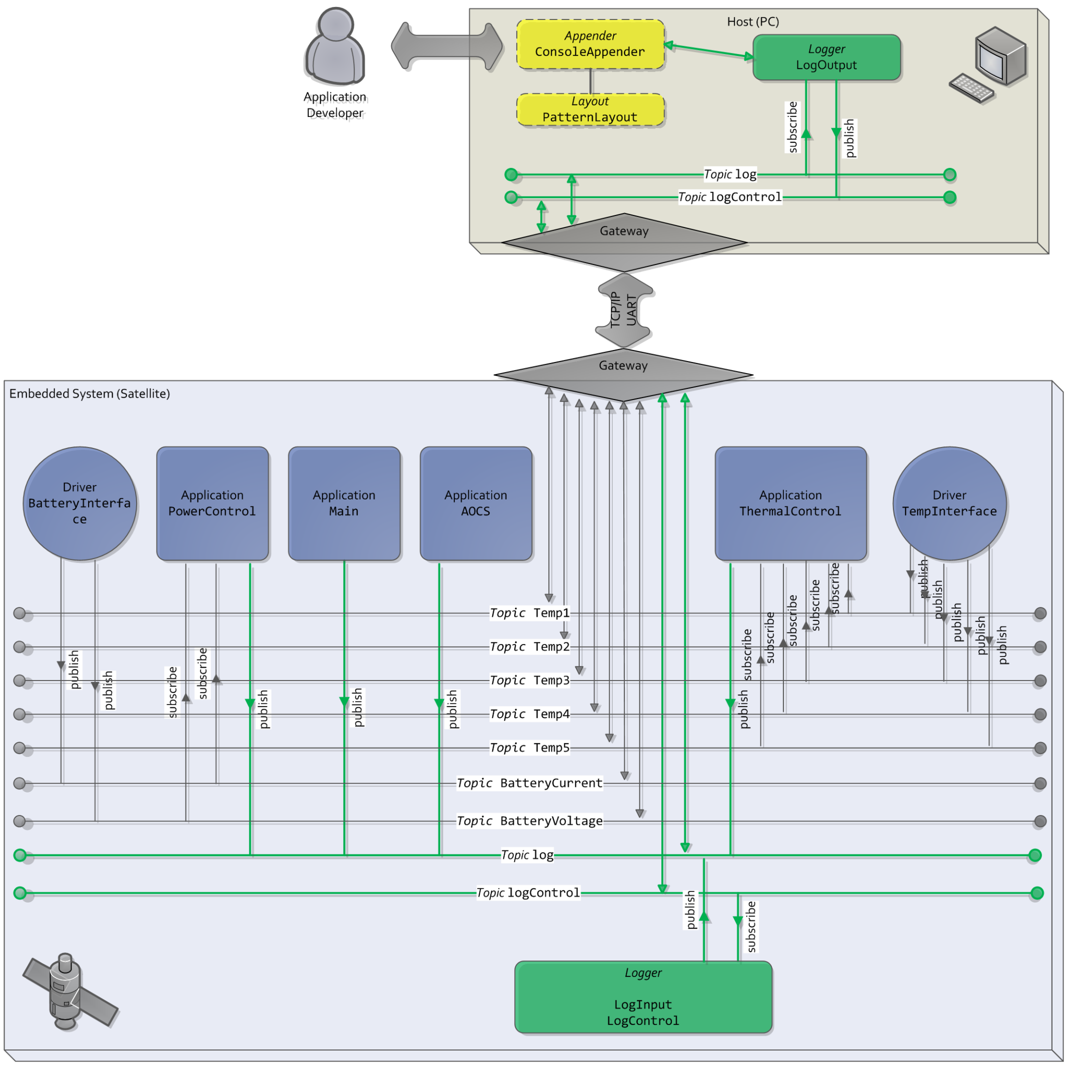


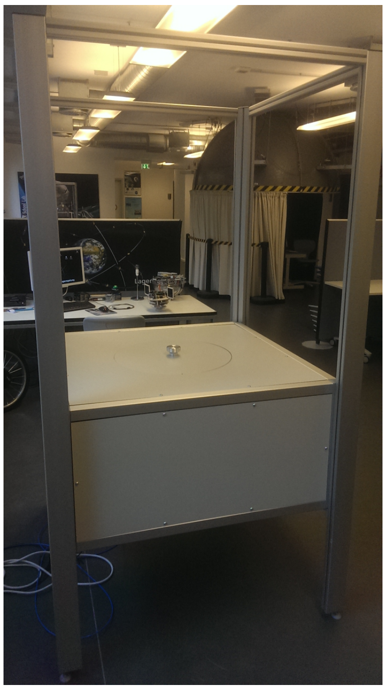
Figure 16.13: RODOS Debugging using the Monitoring F/W (Advanced Scenario)

be used to control the attitude of a satellite in space[57]. The pupils are able to get hands-on experience with the underlying physical effects and how a mini-satellite can be maneuvered in a remote operational scenario.

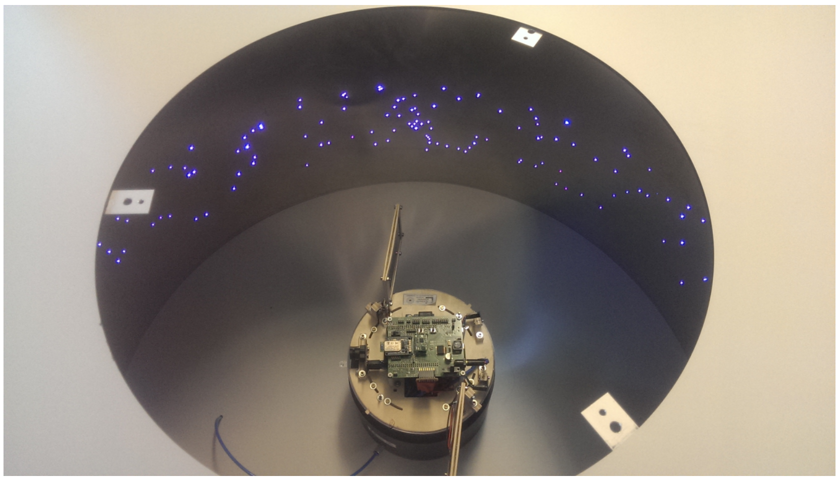
The overall FloatingSat assembly can be seen in Figures 16.14a, 16.14b and 16.15a. During an upgrade of a first FloatingSat prototype to a full-fledged satellite experiment the following enhancement in terms of the overall C&DH subsystem have been undertaken:

ADDITIONAL SENSORS The FloatingSat CDH core consists of a STM32F4-processor based board with an integrated Inertial Measurement Unit (IMU) (see Figure 16.15b). The IMU can measure rotation

57 Actually, the only technique used within the FloatingSat is to control the rotation- and acceleration rates of a brass disc. This disc is used as a simplified version of a reaction wheel.



(a) FloatingSat Overall Assembly (closed Chamber)



(b) FloatingSat with artificial Star Background (open chamber)

Figure 16.14: The FloatingSat School_Lab Experiment

and acceleration rates as well as the magnetic field. In addition, two light sensors have been added to the satellite with the purpose of automatically orientate the satellite towards a simulated sun and doing an automatic deployment of the solar panels afterwards. As in the FloatingSat prototype, information about the battery and solar cells (currents and voltages) can be accessed.

RODOS UPGRADE ON FLOATINGSAT-OBC RODOS version 113 with the compiled-in Monitoring F/W prototype is used on board the

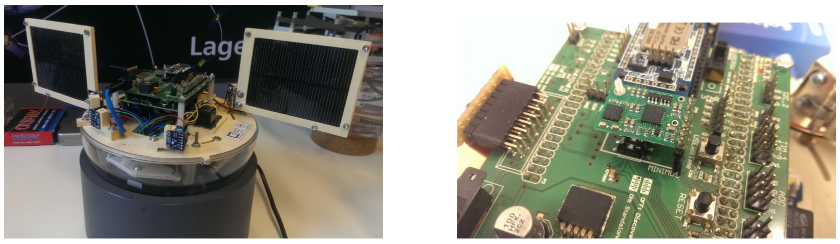

(a) FloatingSat on Air Bearing Table (b) The STM32F4 Board with IMU

Figure 16.15: FloatingSat Avionics System

FloatingSat. Newer RODOS versions do not support the Monitoring F/W on the STM32F4 port, older versions cannot process incoming middleware messages (like TC) which were sent via Bluetooth.

RODOS USAGE AS SIMULATED GROUNDSTATION: RODOS will be also used on the School_Lab desktop computer (running Linux) and act as an EGSE and/or simulated groundstation: after establishing a middleware communication to the FloatingSat via Bluetooth this RODOS instance receives middleware messages containing TM values and is also able to steer the FloatingSat by sending TCs. For convenience a GUI was developed which is able to display telemetry values and allows the manual control of the satellite be sending TCs. This GUI is shown in Figure 16.16. Here the Monitoring F/W prototype had to be linked to RODOS version 116: the Linux-port of older RODOS versions is not compatible with the STM32F4 processor, whereas older versions had problems with sending TCs to the FloatingSat.

EXCHANGE OF MIDDLEWARE MESSAGES The upgraded FloatingSat experiment will use the RODOS middleware to establish a communication between groundstation and satellite over a Bluetooth connection. This also differs from the use cases 1 and 2 which were based on a UART connection. Like all other middleware messages, also the data of the Monitoring F/W contained in the `LogItems` and `LogControlItems` have to be sent over this radio link.

In addition, many upgrades regarding other subsystems have been made. Just to name the most important, a professional air bearing table and a star background – together with a star tracker camera on the embedded side – were added to the system in order to make the DLR_School_Lab experiment most realistic in comparison to a satellite scenario in space. However, within this thesis we will concentrate on the enhancements related to the experiment's avionics.

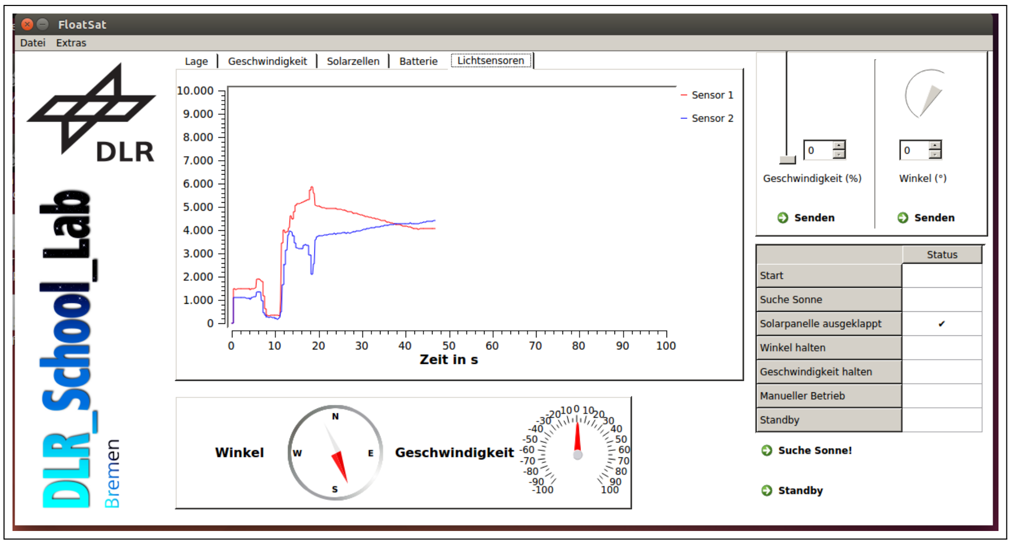


Figure 16.16: Main GUI of the FloatingSat DLR School_Lab Experiment

16.4.3.1 *Experiment Setup*

In contrast to the use cases 1 and 2 in this FloatingSat use case we have refrained from preparing a test scenario using the RODOS PRINTF method. On the one hand, the testing of use case 1 and 2 in the following Sections 18.2 and 18.3 already contain thorough comparisons of the usage of the F/W in comparison to PRINTF. On the other hand, the usage of PRINTF normally requires a wired connection (e.g. using a UART) to the FloatingSat in order to tap off the console output of the PRINTF statements. This in turn would strongly interfere with the FloatingSat's air bearing, making the overall scenario less realistic in comparison to a real satellite mission.

An overview of the experiment setup can be seen in Figure 16.17. In addition to the middleware communication already used in the use cases 1 and 2, the RODOS instance running on the non-embedded side is communicating via UDP with a GUI (for further details cf. to the next Section 16.4.3.2).

Like for the previous use case, the schematic overview in Figure 16.18 shows the communication structure for the FloatingSat use case[58]. A detailed explanation of all involved applications, threads and topics can be found in [74], at this point we will highlight some particularities of this setup:

REALISTIC OPERATIONAL CONDITIONS
For several reasons this scenario offers the possibility to the Monitoring F/W to be used in a realistic operational scenario. First – and this differs most from the other use cases of the prototype – it is the design decision of the programmer to use the

58 Due to space constraints the layouts (e.g. PatternLayout) belonging to the appenders are not shown in the diagram.

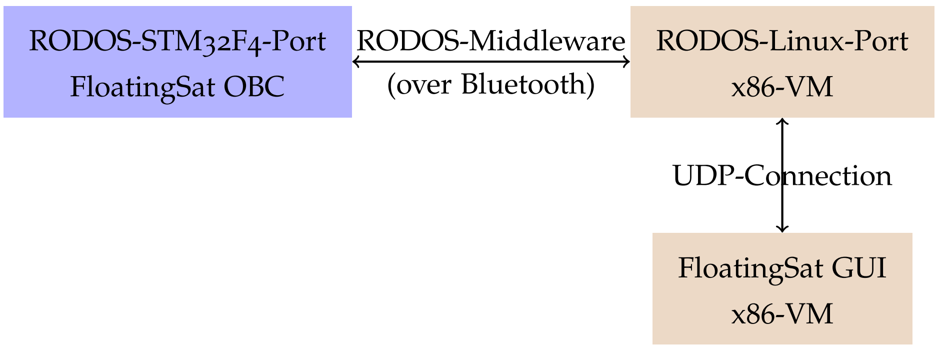


Figure 16.17: Setup of Use Case 3

Monitoring F/W for transportation of debugging information while in parallel still using normal TM for transferring status information to the groundstation. Second, it is also a design decision of the programmer to encapsulate every thread by an application (which can group together multiple threads) or let the thread operate on its own (without encapsulation by an application).

INTERNAL AND EXTERNAL TOPICS As it can be seen, some topics on the embedded side are used only internally, while the topics with the prefix `TM_` or `TC_` are going to or coming from the gateway to the simulated groundstation.

TOPICS TRANSPORTING MULTIPLE VALUES In contrast to the previous use cases, some topics do not transport single values, but whole structs. For example, the `Power` topic sends the `TM_power` struct over the middleware which contains voltages and currents from the solar cells as well as the batteries, and the struct of the `Light` topic contains the values from both light sensors.

REALISTIC TEST SCENARIO OF USE CASE 2 Finally we can state that the test scenario constructed for the prototypes use case 2 was most realistic in terms of the number of applications, threads and topics on the embedded H/W platform. As a result, the performance tests undertaken for the use case 2 can be assumed to be also valid for the FloatingSat use case.

16.4.3.2 *Framework Appenders*

As it has been practiced in the use case 1 and 2, a `ConsoleAppender` was used in order to print out monitoring information to the console of the application developer. In addition – as depicted in Figure 16.18 – a new `GUIAppender` was added to the experiment setup. This appender communicates via UDP with the main GUI of the FloatingSat (see Figure 16.16) which was extended with a `LoggingWindow` for displaying monitoring information and sending configuration commands (here: changing the global log level) to the FloatingSat in order to configure the Monitoring F/W's embedded part at runtime. Both appenders may be run in parallel at the same time, presenting the same monitoring information in a different way to the users, as it is depicted in Figure 16.19.

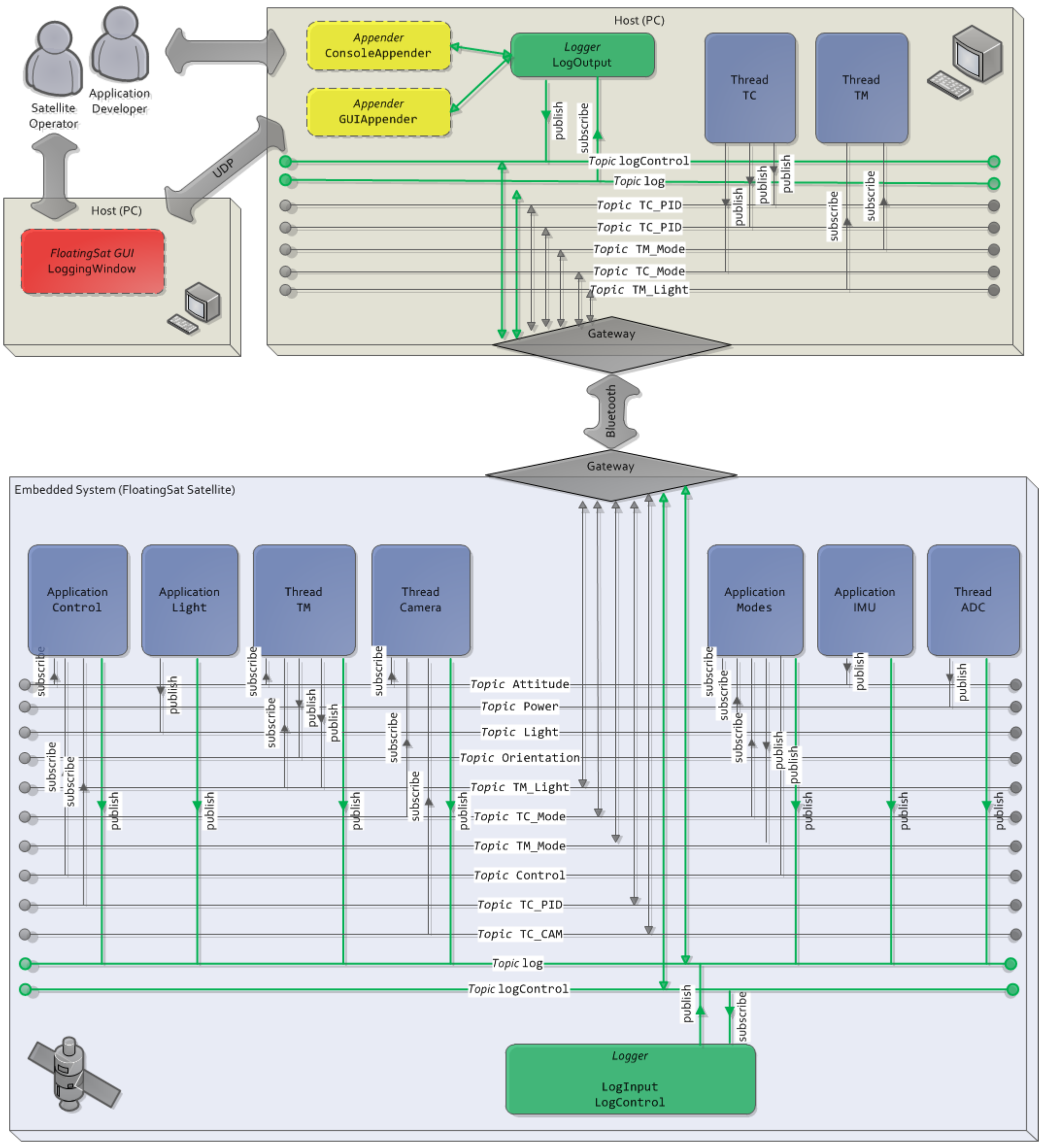
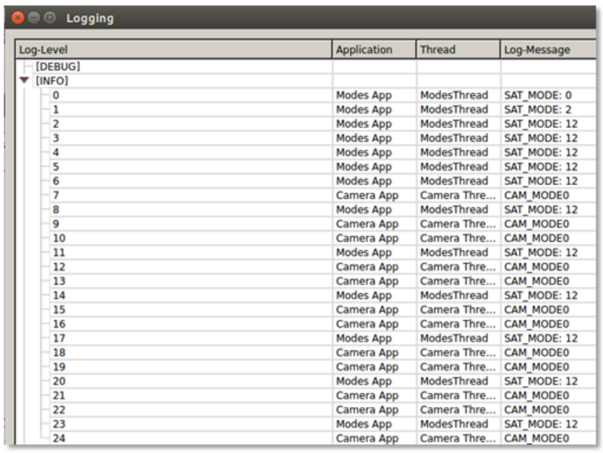


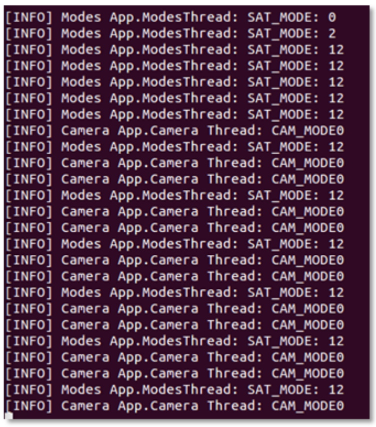
Figure 16.18: RODOS debugging using the Monitoring F/W (FloatingSat Scenario)



Logging

Log-Level	Application	Thread	Log-Message
[DEBUG]			
[INFO]			
0	Modes App	ModesThread	SAT_MODE: 0
1	Modes App	ModesThread	SAT_MODE: 2
2	Modes App	ModesThread	SAT_MODE: 12
3	Modes App	ModesThread	SAT_MODE: 12
4	Modes App	ModesThread	SAT_MODE: 12
5	Modes App	ModesThread	SAT_MODE: 12
6	Modes App	ModesThread	SAT_MODE: 12
7	Camera App	Camera Thre...	CAM_MODE0
8	Modes App	ModesThread	SAT_MODE: 12
9	Camera App	Camera Thre...	CAM_MODE0
10	Camera App	Camera Thre...	CAM_MODE0
11	Modes App	ModesThread	SAT_MODE: 12
12	Camera App	Camera Thre...	CAM_MODE0
13	Camera App	Camera Thre...	CAM_MODE0
14	Modes App	ModesThread	SAT_MODE: 12
15	Camera App	Camera Thre...	CAM_MODE0
16	Camera App	Camera Thre...	CAM_MODE0
17	Modes App	ModesThread	SAT_MODE: 12
18	Camera App	Camera Thre...	CAM_MODE0
19	Camera App	Camera Thre...	CAM_MODE0
20	Modes App	ModesThread	SAT_MODE: 12
21	Camera App	Camera Thre...	CAM_MODE0
22	Camera App	Camera Thre...	CAM_MODE0
23	Modes App	ModesThread	SAT_MODE: 12
24	Camera App	Camera Thre...	CAM_MODE0

(a) `FloatingSat GUIAppender`



```
[INFO] Modes App.ModesThread: SAT_MODE: 0
[INFO] Modes App.ModesThread: SAT_MODE: 2
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Camera App.Camera Thread: CAM_MODE0
[INFO] Modes App.ModesThread: SAT_MODE: 12
[INFO] Camera App.Camera Thread: CAM_MODE0
```

(b) `FloatingSat ConsoleAppender`

Figure 16.19: The two FloatingSat Appenders of the Monitoring F/W running in parallel

17 MONITORING FRAMEWORK FLIGHT VERSION

The content of this chapter driven by the objective to let the new monitoring infrastructure and methodology provided by the Unified Monitoring demonstrate its potential within a current space mission. Leaving behind the stages of breadboarding and demonstrators, out of the experiences and results gained from the F/W's prototype a flight version of the F/W will be developed and integrated within the development process of the Eu:CROPIS mission in order to provide a unified access to S/C status information from the earliest phases on.

17.1 EU:CROPIS COMPACT SATELLITE MISSION

The DLR has decided to develop a compact satellite[59] to be able to promote a freely accessible, space based platform in order to carry out own research and development activities [62].

The first mission of the DLR compact satellite is named *Eu:CROPIS*[60] and hosts a biological payload in the context of *research under space conditions* and *exploration*. It shall provide experiments for research on long term stability and restart capabilities of closed loop life support systems based on regenerative basis, including harvesting of food under variation of g-levels. The mission duration will be between 18 and 21 months, flying in a low earth, sun-synchronous orbit at an altitude of 500 to 650 km. The launch is planned in 2017 [34].

As it can be seen in Figure 17.1, Eu:COPRIS is the second project within the compact satellite program. It was a major design goal for most of the subsystems of the Eu:CROPIS predecessor AsteroidFinder to make as much re-use of already developed concepts and components as possible [21] [31] [58]. Nevertheless, the onboard avionics system has undergone a substantial re-design which led to a completely new H/W platform.

As a result, the challenge of dealing with new and mostly unknown H/W components was put on the S/W development for Eu:CROPIS. This and the fact that most of the satellite S/W to be developed shall be re-usable without major modifications in future compact satellite projects were the major design drivers for the layered S/W architecture chosen.

The flight version of the Monitoring F/W will be integrated into this architecture as one of the services which can be used by the flight S/W

59 The term *compact satellite* refers to a satellite limited to an envelope of approximately 1 cubic meter and a mass maximum of approximately 200 kg which means that the compact satellite is ranked at the lower end of the mini satellite class [31]

60 Euglena Combined Regenerative Organic Food Production In Space

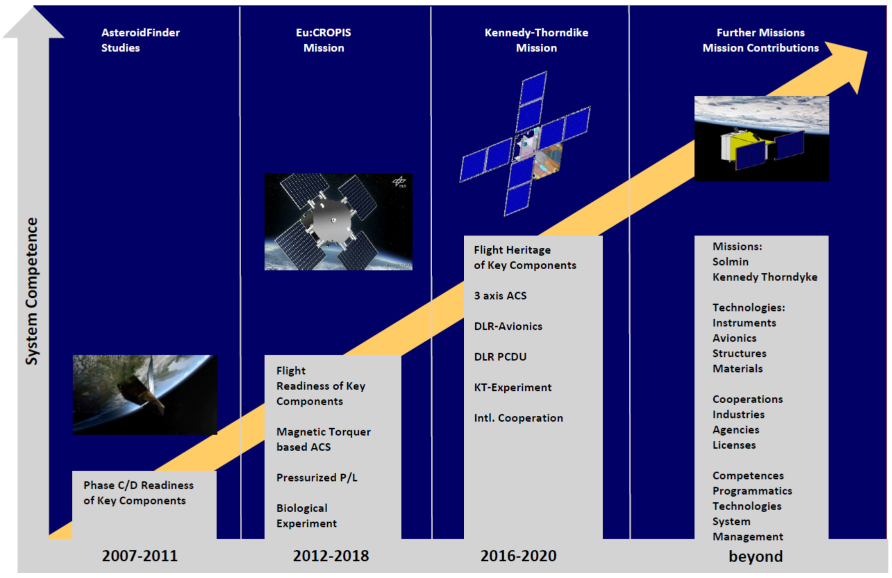

AsteroidFinder Studies
Eu:CROPIS Mission
Kennedy-Thorndike Mission
Further Missions Mission Contributions
System Competence
Phase C/D Readiness of Key Components
Flight Readiness of Key Components
Magnetic Torquer based ACS
Pressurized P/L
Biological Experiment
Flight Heritage of Key Components
3 axis ACS
DLR-Avionics
DLR PCDU
KT-Experiment
Intl. Cooperation
Missions: Solmin Kennedy Thorndyke
Technologies: Instruments Avionics Structures Materials
Cooperations Industries Agencies Licenses
Competences Programmatics Technologies System Management
2007-2011
2012-2018
2016-2020
beyond

Figure 17.1: DLR Compact Satellite Mission Timeline [62]

applications. In the following section a quick overview about the already implemented constituents of this architecture will be provided, mainly taken from the internal design documentation contained in the *Eu:CROPIS C&DH SW Design Definition File* [29].

17.1.1 *The Compact Satellite S/W Architecture*

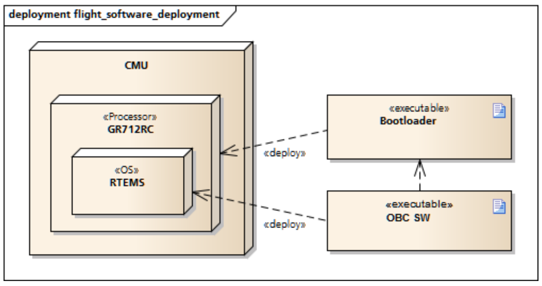


Figure 17.2: Eu:CROPIS Flight S/W Deployment on OBC H/W

The C&DH subsystem is one of the core components of the Eu:CROPIS satellite mission and is part of the compact satellite bus. It is the central instance for processing TCs and generating TM, and for supervision and control of the satellite system. The C&DH H/W will consist of a central, redundant OBC which provides interfaces to sensors, actuators, communication equipment, the power control and distribution unit, and the payloads. In addition, the C&DH flight S/W consists of two main applications: the bootloader and the OBC S/W (see Figure 17.2). While the OBC S/W uses a RTOS for multiprocessor systems (RTEMS) as OS, the bootloader runs natively on the processor to keep the executable small to fit in the limited space of the boot memory. The bootloader loads the OBC S/W from the attached non-volatile memory into the main memory, thereby checking and correcting errors in the image. After a valid image is loaded, the bootloader jumps to the start of the flight S/W, effectively transferring control to it. To promote reuse, the OBC S/W is split into two major components:

THE FLIGHT S/W APPLICATIONS
: The flight S/W applications are dedicated to steer and control the compact satellite bus. To name the most important, there are:

- the ACS,
- the ONS,
- the Thermal Control System, and
- the Power Control System.

THE LIBCOBC S/W LIBRARY
: The libCOBC S/W library is a mission and platform independent library providing OS and H/W abstraction, together with a small communication middleware and the means of communicating using CCSDS/PUS.

As it is depicted in Figures 17.3 and 17.4 the S/W applications are running on top of and making use of the services provided by libCOBC library.

17.1.2 *libCOBC Software Library*

The libCOBC-library provides a flexible, robust and re-usable S/W platform, its implementation is based on the CCSDS/ECSS recommendations. As shown in Figure 17.3, the library is located between the underlying OS and the BSP on the one hand and the applications on the other hand.

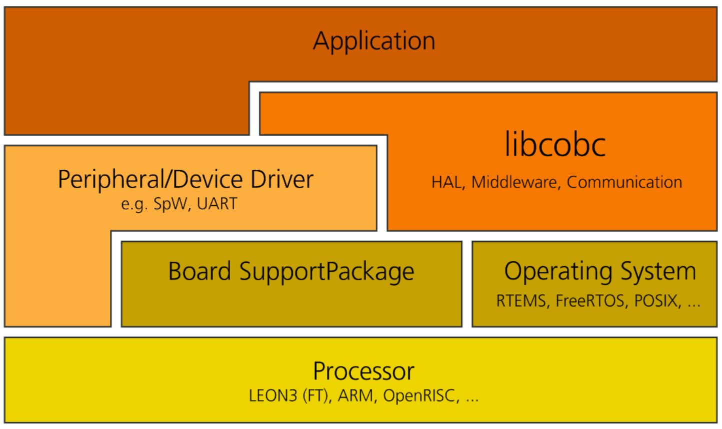


Figure 17.3: Layered S/W Architecture of the Compact Satellite

The core elements of this platform are composed of a Hardware Abstraction Layer (HAL), an OS layer, a middleware layer, and essential services like a CCSDS/PUS S/W stack. Figure 17.4 shows these included modules and layers and their dependencies among each other. A more detailed description can be found in [13].

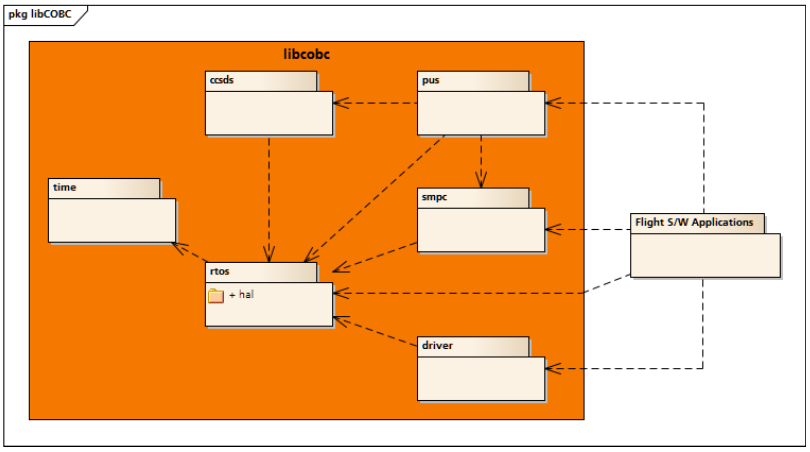


Figure 17.4: libCOBC Modules and their Dependencies

17.1.3 *Housekeeping and Integration of Monitoring Framework*

Because of the layered S/W architecture of the libCOBC-library described in the above Section 17.1.2, the implemented flight S/W applications will be independent from the embedded RTOS and the driver provided for a specific hardware platform. This facilitates their portability and re-use for future missions.

These characteristics make the libCOBC-library an ideal place for integration of the flight version of the Monitoring F/W as an additional module. However – as the Monitoring F/W is yet only available in its prototype version (cf. to Chapter 16) – a complete replacement of the traditional housekeeping application by the Monitoring F/W, meaning that all applications are providing their housekeeping information to the F/W only, would be too risky at current state of the F/W development. Therefore it was decided within the requirements phase of this project to complete the Eu:CROPIS avionics platform by adding the Monitoring F/W flight version as an additional application running on top of libCOBC-library and in parallel to the satellite's housekeeping application (see also next Section 17.2: Design). This is depicted Figure 17.5.

Standard TM will be used in order to transfer the monitoring data to ground. Figure 17.6 provides an overview of information sources for the TM link within Eu:CROPIS.

In this way the integration of the Monitoring F/W is enhancing the traditional housekeeping capability by offering extensive filtering and debugging techniques for monitoring and FDIR needs. On the other hand, for the Monitoring F/W this mission serves as an on-orbit test

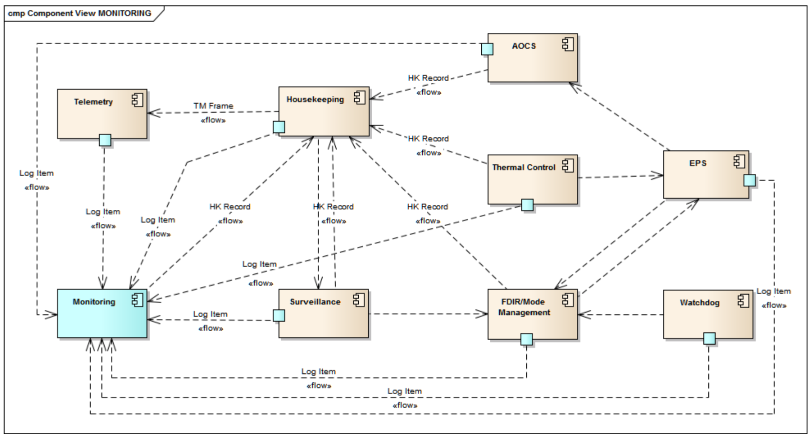


Figure 17.5: Integration of Monitoring F/W into Eu:CROPIS Flight S/W (Example; simplified View))

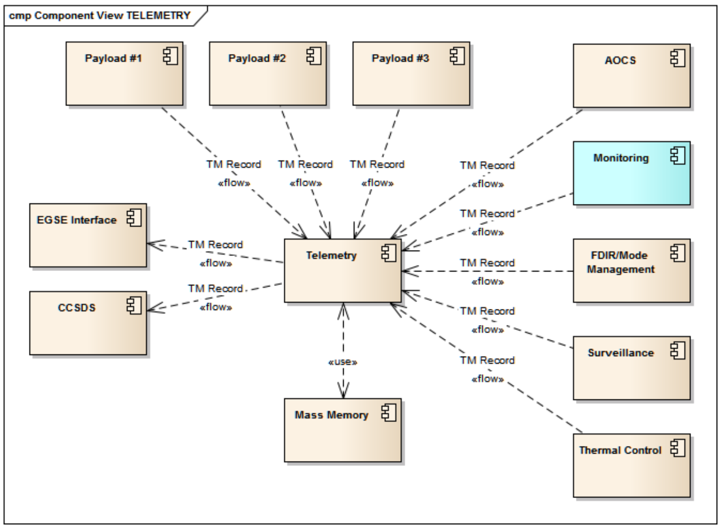


Figure 17.6: Eu:CROPIS Monitoring Framework using TM link (example; simplified view)

environment for technology demonstration. The design of flight version of the Monitoring F/W is described in the following Section 17.2.

17.2 DESIGN

The design and implementation of the Monitoring F/W flight version is based on the development of the prototype of the Monitoring F/W which was described in detail in the previous chapter (cf. to Chapter 16 ff.). It started when the Eu:CROPIS S/W development was already ongoing. As the Monitoring F/W prototype had to be ported to a completely new S/W environment using also new communication mechanisms (see above Section 17.1.3) it was decided to implement the flight version from scratch. This was primarily done within the scope of a master thesis [28]. Nevertheless, the experience and results from the tests of the prototype which are described in Chapter 18 had to be taken into account for the design of the flight version (see also the corresp. Section 2.4 in [28]).

Therefore this section starts with a short description of the traditional debug approach used in the Eu:CROPIS source code which is based on the well known printf-statements together with an associated pre-processing macro.

Afterwards the design of the Monitoring F/W flight version is described. Special emphasis is being put on addressing the final findings from the testing of the prototype which are presented in Section 18.5. Therefore, this section can be understood as a *Lessons Learned* from the Monitoring F/W prototype: during usage and testing of the prototype it turned out that a lot of the initial requirements presented in the corresponding Section 15.1 and – in more detail – in the appendix Section A.4 are fulfilled with great success and could be taken over also for designing the flight version of the F/W. On the other hand, some requirements turned out as not being very useful or practicable. They had to be adapted or skipped completely. Finally, new requirements from integration of the F/W into the flight software of the Eu:CROPIS mission (cf. to Section 17.1) arose.

Because of this, every section which describes a certain design area of the Monitoring F/W flight version also contains a look at the corresponding (high level) prototype requirements and/or test experiences and explains, how they influenced the design of the flight version[61].

Afterwards we will show how the flight version design also influenced the design of the *libCOBC* library (see 17.1.2) it is running upon.

This section ends with a look on some safety aspects. The main focus lies on describing the mechanisms which guarantee that even in

61 This *design-centric* approach differs from the *top-down* approach used during implementation of the Monitoring F/W prototype (cf. to the diploma thesis of Müller, Sven [63]) and flight version (cf. to the master thesis of Göksu, Murat [28]): herein the requirements were divided into functional, resource/performance and technical/implementation sections, whereas in this section we will look at the different design areas (general, user interface, communication & testing).

the extreme cases like FDIR events (from inside the system) or misconfiguration (from outside the system) the behavior of the F/W does not influence the overall behavior of OBSW in a significant way.

17.2.1 *Replacement of Eu:CROPIS PRINTF*

Development started with PRINTF...

The Eu:CROPIS development of the OBSW started with the traditional approach of using printf-statements for debugging the execution flow and internal state of the S/W. In order to get rid of these statements for testing the resource consumption directly on the H/W or creating the final flight version of the S/W the usage of pre-processor statements was chosen (cf. to Section 12.1.1 for further details about this technique) [62].

As an example we will have a look at a code fragment from a part of the OBSW, that is responsible for inserting commands from the groundstation into the schedule of commands to be executed next by the OBC. Listing 17.1 shows the corresponding source code from the file header. In case the `DEBUG` flag is set the C++ pre-processor is instructed to replace all statements within the code surrounded by `LOG(x)` with the statement inside the brackets. If the `DEBUG` flag is not defined (left away or set to `false`) the line containing the `LOG` statement will be deleted from the source code before compilation. Listing 17.2 displays the affected printf-statement section of the source[63].

Listing 17.1: libCOBC Pre-Processor Macro for Replacement of printf-Statements

```
#ifdef DEBUG
# define LOG(x) x
#else
# define LOG(x)
#endif
```

Listing 17.2: Affected Code Fragment from libCOBC

```
% if (!validateCommand(command)) {
%   LOG(printf("%s:%i: invalid command\n", __func__, __LINE__);)
%   return false;
% }
```

As for the Monitoring F/W prototype, the replacement of the printf-statements inside the Eu:CROPIS source code is the main use case of the flight version. Further explanations regarding the reasons for

62 In addition, this encapsulation of printf-statements avoids the breaking of the JSF rule no. 22 (see [11]) which forbids the usage of the C++ `stdio.h` library and hence the usage of printf-statements.

63 `__func__` and `__LINE__` are standard predefined macros used by the pre-processor to include the current function and input line number.

integration of the F/W's log-statements into this abstraction level were already given in Section 15.2.

17.2.2 *General Design Driver*

Like described at the beginning of this section, some of the design aspects from the F/W's prototype turned out to be extremely useful and are supposed to be taken over also for the flight version. Therefore, the general design driver to be identified for the flight version are driven by the originally laid down prototype requirements.

17.2.2.1 *Design by Separation of Concerns Principle*

The *Separation of Concerns* principle as described in Section 14.2 was already successfully implemented by the prototype. It is also a major design driver for the flight version, as most of the power and flexibility of the F/W results from this principle (cf. e.g. to Sections 17.3.5 and 17.3.5 below).

17.2.2.2 *Usage for Debugging in the Development Phase and during Operation*

Especially during testing of use case 3 (see Sections 16.4.3 and 18.4) already the prototype revealed its strength in terms of enabling debugging possibilities also in scenarios where normally no debugging would be possible. Therefore, the requirement was taken over for the flight version, even though it is already clear that a real-time debugging like in use case 3 is not yet possible (for reasons cf. to Sections 17.1.3 and 20.2).

17.2.2.3 *Store and forward Debug Information coming from other Applications*

While in principle the Monitoring F/W prototype as well as the flight version are capable of recording all kinds of information, this initial requirement was skipped again: we will concentrate on the printf replacement as a first step (for reasons cf. to Section 17.1.3) and do a complete replacement later on (cf. to Section 20.2). The printf replacement was already successfully demonstrated in the three uses cases of the prototype. The flight version will go even further by adding a better log granularities through the introduction of so called *Application Logger*s, and by adding more data types which can be logged. Please refer to the flight version file `dataTraits.h` for all currently supported data types (cf. also to the corresp. Paragraph C++ Traits on Page145).

17.2.2.4 *Configurable at Runtime by Telecommand in Terms of Monitoring Target and Level of Detail*

Both – prototype as well as flight version – can be configured by TC. For the prototype the configuration is limited to the global log level (= global level of detail), a monitoring target cannot explicitly be chosen. Switching over to the flight version, the F/W will be equipped with application loggers which bring the demanded configuration regarding the logging granularity also to the application level (= gobal & application level of detail). The various configuration possibilities for the flight version – at runtime as well as at compile time – are described in the Configuration Section 17.4.

17.2.2.5 *Support for different Log Frequencies*

By integration the Monitoring F/W in the application layer (cf. to Section 17.1.3) log frequencies can be controlled indirectly by changing the period in which an application (e.g. housekeeper) is invoked by the RTOS scheduler[64]. In addition, the monitoring for each application can be turned off completely, and the log level can be configured by TC (see Section 17.2.2.4) which can also be understood as means of log frequency change.

17.2.2.6 *Integration in RODOS Middleware*

This initial requirement which was fulfilled by the prototype had obviously be adapted for the flight version: RODOS is not used as RTOS for the Eu:CROPIS mission, therefore the flight version has to be designed for utilizing the libCOBC S/W platform instead (cf. to Section 17.1.1). Nevertheless, as libCOBC is also implemented in C++ the programming language will stay the same. The requirement has been replaced by the following new high-level requirement to be addressed by the flight version design:

THE LOGGING COMPONENT SHALL BE INDEPENDENT FROM THE UNDERLYING RTOS As already justified in Section 17.1.3, the Monitoring F/W flight version will run as an additional application on top of the libCOBC-library. With this design we implicitly achieve the independence from the underlying RTOS which is postulated by this new requirement on the flight version, as the libCOBC offers – among others – also a RTOS abstraction layer (cf. to Section 17.1.2). Moreover, running as an application on top of the libCOBC library (see Figure 17.3) results in the following additional requirements which have to be fulfilled by all other applications as well:

- The Monitoring F/W flight version has to be equipped with a unique APID.

64 This can normally be done only at compile time.

- The Monitoring F/W flight version has to deliver information to the classical housekeeping application of the satellite (like e.g. error counters for lost monitoring messages) (see Figure 17.5).
- The monitoring data will be transmitted as normal historical TM to the groundstation (see Figure 17.6).
- The monitoring configuration will take place using normal TC from the groundstation.

17.2.2.7 *Very low Resource Consumption*

The major goal of the design and development of the F/W's prototype was the demonstration of the new idea of Unified Monitoring, as a consequence the focus was on demonstrating the new functionality. Yet, as the final goal is to integrate the F/W into a satellite's boot image to be run on the OBC, some basic testing in terms of performance and resource consumption has been undertaken in Chapter 18. As summarized in Section 18.5, the prototype performed quite well in general, but revealed weaknesses in some areas:
The load on the processor by increased scheduler activity by integrating additional log threads (cf. to Section 18.5.4.1), the load by the log threads themselves (cf. to Section 18.5.4.2) and the load from message transport (cf. to Section 18.5.5) are acceptable, but optimization potential regarding the increase of Lines of Code (LoC) (see Section 18.5.3.1) and the increase of the boot image size (see Section 18.5.3.2). is given.

As the minimal resource consumption of both – memory & processor – is even more important for the flight version, most of the tests were repeated in Section 19 to resolve these findings. Furthermore, special emphasis has to be put on testing also the communication to ground (see below in Section 17.2.4) like it was done in the testing of use case 3 in Section 18.4.

17.2.2.8 *Runtime Behavior of the overall System shall be not affected*

The *performance* tests of the F/W's flight version (see above Section 17.2.2.7) have to show that the runtime behavior of the overall system by adding the F/W is not affected – or only affected within a tolerable extent. The *functional* behavior of the F/W's prototype was tested only very basically within the use cases 1 and 2 by implicitly showing that the usage scenario is still working when switching over from the traditional printlining debugging to the new F/W. However, the continuation of the functional behavior of the rest of the Eu:CROPIS flight S/W after adding the F/W will be guaranteed by appropriate unit- and integration test.

17.2.2.9 *Based on existing Frameworks*

The design of the F/W's flight version will be guided by the F/W's prototype which in turn was designed taking existing frameworks into account. For details please confer to [63, sec. 2.6, chap. 3].

17.2.2.10 *Intuitive User Interface / Easy to understand API*

As depicted in Section 16.1.2, the user interface of the prototype was already intuitive, the same will hold for the flight version (cf. to Section 17.2.3 below).

17.2.2.11 *Good Expandability*

New requirements on the F/W's prototype and flight version will arise from future use cases, some of them are already mentioned in the OUTLOOK-Chapter 22. Therefore expandability has be guaranteed by design.
Implicitly this is done by the usage of the C++ programming language and the object-oriented design of both versions. Even though it is hard to foresee which changes will be required in the future, we can at least try to address the most likely ones:

- Essential `LogLevels` are already predefined in both F/Ws, they can easily be extended/adapted by changing the F/W's configuration (see the corresp. Sections 16.3 and 17.4).
- For sure new appenders and layouts will be added, as the output of the Monitoring F/W is the place where most of the adaptions needed for future mission scenarios will take place[65]:
  - Adding new appenders for the F/W's prototype is currently optimized for console output. Other appenders can be added, but as the implementation of the `GUIAppender` in use case 3 showed they are difficult to implement due to the current design of the prototype (see Section 18.4.4).
  - Adding new appenders for the flight version is also possible (see Section 17.3.5).
- For all future changes and extensions to be made it is most important to guarantee the correct functionality of the F/W after integrating them. Therefore the F/W's flight version was equipped with unit test (see Section 19.1) which are invoked during the build process.

17.2.3 *Design of the User Interface*

The basic requirements regarding the user interface – like a familiar syntax for debugging statements, support for all possible data

65 This of course depends on whether the initial design of the F/W proves itself suitable to fulfill the monitoring needs of a space mission during its operation.

types and additional logging-specific information like the log levels – were already fulfilled by the F/W's prototype. Like described in Section 16.1.2, the advantages of operator overloading were already used by the user interface of the prototype.

For the flight version this interface was not blindly taken over but newly designed in close cooperation with the Eu:CROPIS S/W developers. These discussions led to the final decision of going back to the familiar printf-like notation: it offers more freedom for the programmer and is furthermore most common within the embedded world, whereas printing out debug messages with `cerr` or `cout` in combination with the « and » operators is mostly done on non-embedded systems which are programmed in C++.

The drawback of not profiting from the advantages of the prototype's user interface design[66] will be compensated in the implementation of the flight version by a sophisticated pre-processing approach. It is described in detail in Section 17.3.3.

The following source code line contains an example log statement of the flight version:

```
appLogger->log(DEBUG, "32 bit unsigned integer: %u32\n", uint32_-
t(123));
```

The name `log` was chosen in order to (1) avoid conflicts with already existing printf statements in the code and (2) to allow the programmer the usage of printf methods if he still wants to use them. As it can be seen, the format string specifier contained in the `log`-methods are implemented similar – and even more precise in terms of the used data types – to the ones known from printf. For all available specifiers please refer to [28, tab.5.2].

17.2.4 *Communication Design*

17.2.4.1 *Distributed Monitoring*

Distributed monitoring means the seamless exchange of monitoring information across node boundaries. Therefor the Monitoring F/W has to rely on the communication mechanisms and tooling offered by the underlying S/W infrastructure.

The exchange of monitoring information between two nodes by utilizing e.g. the RODOS middleware was best demonstrated in use case 3 of the prototype (see Section 16.4.3). This scenario could be easily extended to exchange monitoring information e.g. from ground to space or between two (or more) satellites within an inter-satellite network (cf. to Section 9.1).

66 For example that the log messages are not so nicely created like it was described in Section 16.1.2.

Among other reasons, due to the missing end-to-end support for the exchange of monitoring information in the Eu:CROPIS mission (described in Section 17.1.3) this design requirement was skipped for the flight version, but will be addressed in future missions (cf. to evaluation Section 20.2).

17.2.4.2 *Efficient Transport of Messages*

Efficiency in terms of transporting monitoring information from space to ground means (1) that the amount of data shall be limited (to avoid e.g. an overflow of debug messages) and (2) that no unnecessary data shall be transmitted.

The first requirement is implicitly fulfilled by limiting the quota of the number of bytes being sent to ground through the deterministic buffer handling of the F/W's flight version (see corresp. paragraph in Section 17.2.6). The latter one will be implemented by replacing all constant data containing strings within a log message during a pre-processing step before compilation. For details please cf. to Section 17.3.3 in the next chapter.

17.2.4.3 *Type-Safety*

For our communication design we will consider type safety to be an important property of the Monitoring F/W itself rather than only relying on the type-checking possibilities the C++ programming language offers. Therefore, we will try to enforce to usage of the correct types within as many places of the design as possible by using a combination of static (= catching potential errors at compile time) and dynamic (= associating type information with values at run-time) type safety. The final goal is that all types of monitoring information are used correctly during implementation and will be preserved at runtime throughout the whole communication chain.

For the prototype static type safety was already part of the user interface which provides dedicated operator functions of the «-operator for each data type: using (currently) not supported data types in connection with this operator will result in compilation errors. During the transport of monitoring messages by the prototype data types are preserved by sending `LogItems` over the network: these `LogItems` consist of a header and a data part - the latter one holding the different data types within a C++ `Union` construct.

Like the prototype, the F/W's flight version will also use an adequate protocol for handling all kinds of data (cf. to Section 17.3.2.1). Additionally the pre-processing approach explained in Section 17.3.3 will recognize wrong data types already at compile time.

17.2.4.4 *Robust Message Transport*

This design requirement clearly arose from the problems of the prototype with lost sub-packets (esp. lost package ends). In the final analysis Sections 18.5.6.1 and 18.5.6.2 these problems were described and the following solutions were suggested in order to make the message transport more robust:

- monitoring messages shall not be split/fragmented over several middleware messages,
- the correct reception of the monitoring messages must be guaranteed, and
- the introduction of a unique message ID for every monitoring message is necessary.

All of the above points were addressed by the F/W's flight version:

- The monitoring messages will not be split any more: as the main reason for splitting long messages in the prototype was the transport of long strings, this reason will vanish with the preprocessing approach of the flight version (see Sections 17.2.4.2) and 17.3.3).
- The correct reception of messages will be guaranteed by using the normal telemetry and the robustness features offered by the CCSDS encoding/decoding (cf. to Section 17.1.2).
- A unique message ID will be part of the protocol of the flight version (cf. to Section 17.3.2.1).

17.2.5 *Changes to libCOBC*

While designing the Monitoring F/W flight version some changes had to be made to the design and implementation of the underlying S/W infrastructure of the libCOBC library it will be running upon. Most important was the unique tracking of the source of every monitoring messages, as this is most crucial to determine its current execution context. In order to achieve this, a thread-ID was introduced in the libCOBC library for unique identification of threads (cf. to Section 17.1.2 for further details about libCOBC).

Because every native OS-thread is encapsulated by a libCOBC wrapper thread, the ID of the native OS-thread is

needed on ground for the unique assignment of the name of a thread to its corresponding thread-ID[67]. The assignment will not be done on board of the satellite, but during the post-processing of the monitoring messages on ground: after the startup of the boot image the available thread-IDs will be stored within a separate list and sent to ground via extended TM. With this list, an assignment of thread-ID and thread-name can be done.

67 While the thread-names always stay the same, the IDs are newly created by the OS after every (re-)boot.

17.2.6 *Safety Aspects*

The dynamic configuration of the logging component at compile time as well as at runtime allows a scalability of the monitoring framework in terms of adaption of log levels or changing the granularity and frequency of collecting monitoring messages within the system. But this freedom also includes some danger. For example, a (maybe accidental) misconfiguration of the system could lead to complete freeze of the system. In order to avoid this potential danger, several security mechanisms were introduced to the flight version of the Monitoring F/W.

Preventing System Freeze

DETERMINISTIC BUFFER HANDLING The storage of monitoring data on board of the satellite will take place in dedicated buffers. The F/W's prototype as well as the flight version implementation have on central logging component using one central buffer. In addition, in the flight version each application logger will have its own local buffer. The design of these buffers of the flight version is a critical point: on the one hand they have to be scalable in order to fulfill the missions needs in terms of collecting and storing monitoring data, on the other hand they have to be controlled carefully in order to prevent the freeze of the whole system in case that two many log messages are flooding the system.

Various mechanism have been introduced to achieve this buffer determinism:
At compile time a *global* upper limit and an upper limit *per application* regarding the generated monitoring data were defined as safety requirements. At runtime it is also possible to scale the amount of monitoring data further down by limiting the size which can be used of each application buffer. In addition, the periodicity of copying data out of the application buffers into the global buffer can also be adjusted at runtime. For further details please confer to the implementation Section 17.3.1.1.

THREAD SAFETY As for all multi-threaded embedded S/W also for the Monitoring F/W's prototype as well as the flight version thread safety is an important design requirement: critical sections have to be protected in order to synchronize threads in situations where an interruption from other threads would lead to an unwanted behavior of the S/W, an inconsistent state of e.g. a shared memory area or – in the worst case – a blocking of the whole S/W.

For this, both F/W implementations described in the Sections 16.2 and 17.3 are making use of the technical possibilities (lock, mutexes, semaphores, ...) offered by the underlying S/W infrastructure (RODOS & libCOBC).

FDIR LOG LEVEL In the case of a severe system failure – where even no TC can be received from the groundstation – the satellite has to react autonomously in order to survive. For these extreme cases typically predefined FDIR activities are triggered (like e.g. going into safe mode). Therefore, the Monitoring F/W flight version is equipped with a pre-defined FDIR log level for each application. It is automatically set in the case of an onboard FDIR event, overwriting the log level which is currently valid for the application (but only if the current application-specific log level is not higher[68] than the FDIR log level. All historical monitoring data – even the data which is normally not stored because of an insufficient log level – which was accumulated up to this point will be handed over to the central buffer. The application specific FDIR log level can be changed at run-time per TC. For further details please cf. to Section 17.4.

CODING FOR TESTABILITY The code of the flight version was implemented in a way that it passes the static code analysis as well as the checks regarding the Joint Strike Fighter Air Vehicle (JSF AV) C++ coding standards beforehand (cf. to Sections 19.4 and 19.4.1).

68 Cf. to Section 17.4.2 for the meaning of "higher" in terms of log levels.

17.3 IMPLEMENTATION

As already described above within the Design–Section 17.2, the implementation of F/W's flight version is strongly influenced by the experiences gained by using and testing the F/W's prototype in different scenarios. Therefore, within this section dedicated to the implementation details we will concentrate on how these *Lessons Learned* were taken over into the actual programming of the flight version. For a comprehensive description of all implementation details please refer to [28].

Programming Language

Like the prototype, also F/W's flight version is using C++ as its programming language, mainly because of easier interfacing with the underlying libCOBC S/W library. Large parts of the flight version are implemented in header files. This is due to the extensive use of C++ constructs like templates and traits (see Paragraph C++ Traits below) for which the compiler allows a usage in header files only.

17.3.1 *Static Design*

The initial design for the F/W started in Section 16.1 with the communication Diagram 16.1. For the F/W's flight version diagram, this diagram is further detailed and – in addition – combined with structural elements such as the implemented classes and the data packets to be transmitted. It can be seen in Figure 17.7 and will serve as the central implementation reference for the sections to come. As the reader will notice, many similarities to the implementation details of the prototype can be found (cf. to corresp. Section 16.2).

The F/W's composition out of embedded and non-embedded parts – already implemented within the F/W's prototype – has proven its efficiency and can therefore also be found in Diagram 16.1: the embedded *Transmitter* part is responsible for collecting monitoring information from the applications, storing them at a central point and sending them to the non-embedded *Receiver* part (e.g. development host, EGSE or groundstation). The `(Raw)DataPackets` contain the actual monitoring information, thereby corresponding to the `LogItems` which are already known from the prototype. The I/O interface is responsible for delivering the monitoring information to the end user by utilizing the F/W's appenders which are also well-known from the prototype. The same interface offers also possibilities to configure the F/W: *ConfigPacket*s are substituting the prototypes `LogControlItems` and transport configuration information regarding the adjustments of global- and application log levels and other settings (see Section 17.4 below). An obvious difference to the initial design from Figure 16.1 are the extensions to be seen on the right-hand side within Figure 17.7: due to the design goal of saving bandwidth by extracting a maximum amount of redundant and static information out of

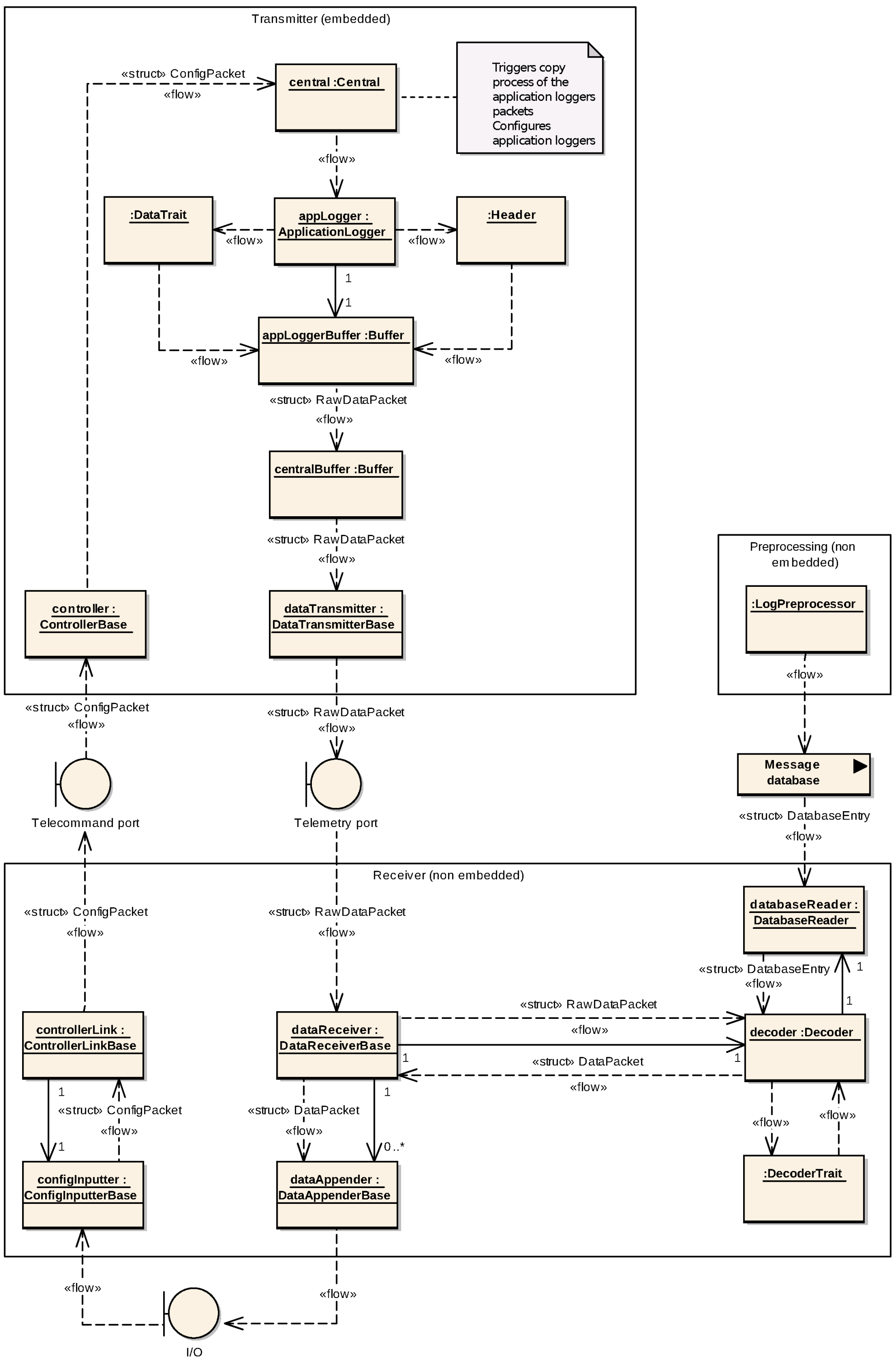


Figure 17.7: Communication and Structure Diagram of the Monitoring F/W Flight Version [28, pic. 4.1]

the monitoring information before sending them to ground (see Section 17.2.4.2), a pre-processing process was introduced (described below in Section 17.3.3). As a result, the `RawDataPackets` received need a further decoding step to restore to original monitoring information before handing them over to connected appenders.

C++ TRAITS Within the F/W's prototype implementation for each type of monitoring data to be handled by the F/W a dedicated operator function overloading of the «-operator had to be implemented (see Section 16.2). As for the flight version a different kind of user interface was selected (see Section 17.2.3) this technique was replaced by a special template class, called *trait*. Within the flight version traits are used at the points where data has to be entered in and to get out of the F/W (see again in Figure 17.7): `DataTraits` realize the correct saving of data within a packet structure (for the `Header` structure see Section 17.3.2.1 below), and `DecoderTraits` are responsible for the correct decoding of the data types out of the `RawDataPackets`.

One of benefits of using traits – especially for the application programmer – is, that they prevent mis-usage of the F/W already at compile-time: the compiler checks the data types and outputs an error message in the case of non-supported types. This ensures that every data type is handled correctly by the Monitoring F/W flight version (see Section 17.2.4.3). For the currently supported data types please refer to [28, sec. 5.2.4].

Traits ensure Type-Safety at Compile-Time

17.3.1.1 *LogBuffer Implementation*

A safe and deterministic buffer handling is most important for an embedded system (see Section 17.2.6). Essentially for the F/W's flight version this was a key point of the development. As Figure 17.7 reveals, the introduction of `ApplicationLoggers` also results in additional application buffers. And as for the prototype, there is still a central buffer. Both are inheriting from the same `Buffer`-class.

MUTUAL EXCLUSION Within each application several threads can be contained. If they try to write monitoring data *in parallel* into the buffers serious errors can occur. To prevent this, the buffer is protected by a mutex (offered by the libCOBC library).

INTERNAL STRUCTURE The buffer of the Monitoring F/W flight version is implemented as a stand-alone and independent FiFo-buffer using a ring buffer structure [69]. As on the embedded side dynamic memory allocation is not allowed, the buffer uses a byte array (instead of a list) to store the monitoring data.

69 In contrast, the corresponding `LogInputBuffer` of the F/W's prototype was making use of the FiFo-buffer from the RODOS RTOS (cf. to Section 16.2).

EMPTYING Periodically the `centralBuffer` reads the content from the `appLoggerBuffers`. This periodicity can be adjusted at compile- and run-time. Only monitoring data with the same or a smaller log-level than the global one is copied until the `appLoggerBuffer` is empty or its transfer limit for this period is reached.

COMPRESSION Before monitoring data is stored into the application buffers it is compressed using two encoding techniques for protocol buffers: Varints (7-bit-Encoding) and ZigZag-Encoding [69]. Details about this compression can be found in [28, page 28].

17.3.2 *Dynamic Behavior*

Like for the prototype (see Figures 16.3 and 16.4) the dynamic behavior of the F/W can be best described by corresponding sequence diagrams. Here we concentrate on one of the main differences between the prototype and the flight version: the introduction of the `ApplicationLoggers`. Figure 17.8 depicts the sequence of storing log messages from the applications into these buffers by invocation of the `logPreprocessed()`-method (cf. to the Pre-Processing Process-Section 17.3.3) for details about the pre-processing approach).

17.3.2.1 *Message Protocol*

Like mentioned in the design Section 17.2.4, the F/W's prototype had several shortcomings regarding the robust and efficient transportation of monitoring data. This is also affecting the design of the protocol used by the flight version. As the saving of monitoring data itself within a packet is realized using traits (see C++ Traits Paragraph above), this section is about the structure of the packet header which encapsulates the data (often also referred to as *meta data*.

Non-splitting of Messages

First of all a serious issue regarding the robust message transport described in Section 17.2.4.4 is addressed by the non-splitting of messages: one monitoring data item has to fit into one TM packet to be sent to the groundstation. This of course results in the fact that the size of monitoring data is limited by the size of the TM packets[70]. As lengthy monitoring data like e.g. long strings are eliminated during the pre-processing step described below in Section 17.3.3, this restriction in terms of size will not be a problem.

The data contained in the header is (partially) specified in the requirements of the flight version (see Appendix A.5):

- Message ID
- Application ID

70 The TM packet size is defined within the Eu:CROPIS project between the space segment and ground segment when designing the ratio link and communication chain.

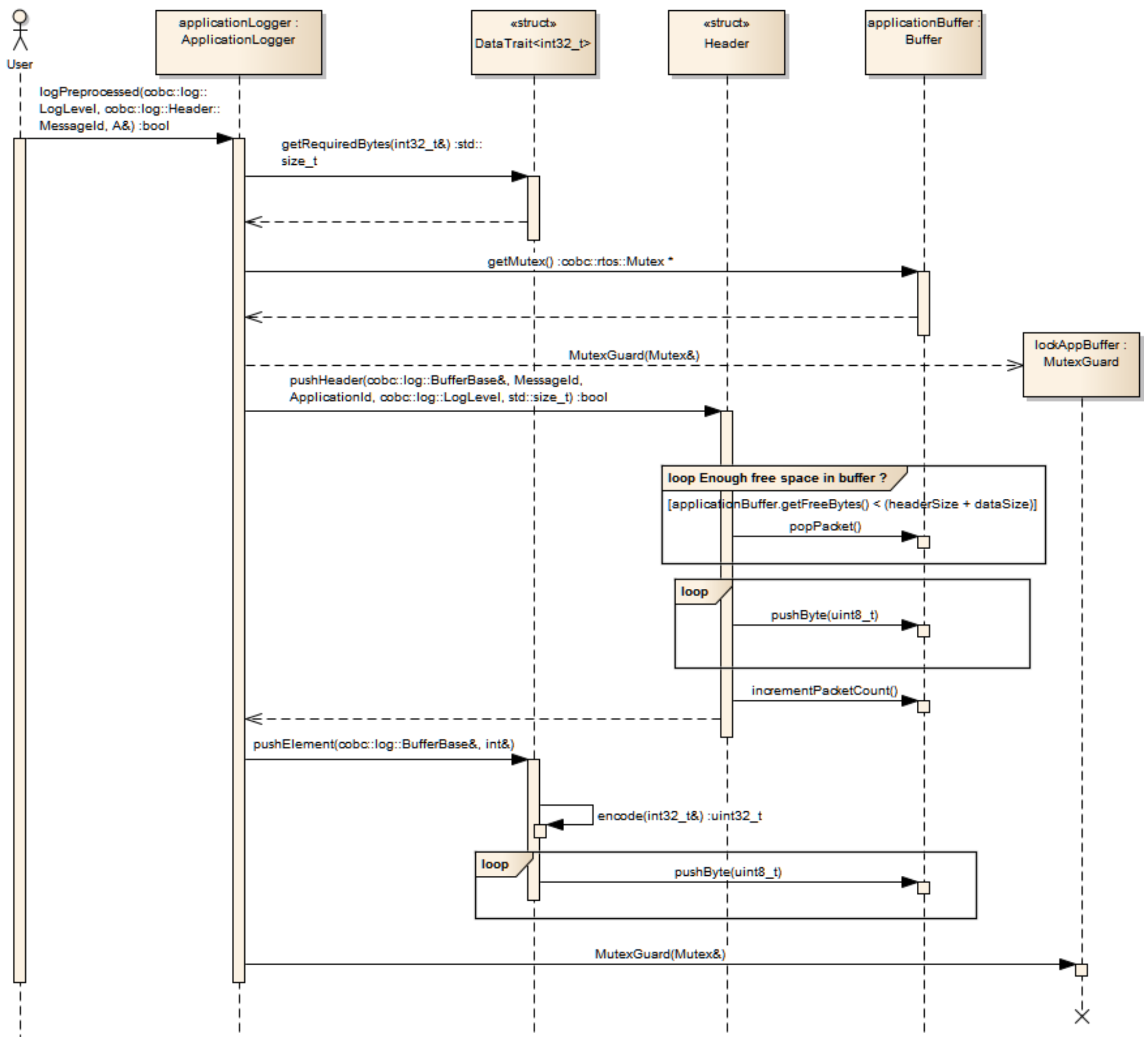

User
applicationLogger : ApplicationLogger
«struct» DataTrait<int32_t>
«struct» Header
applicationBuffer : Buffer
logPreprocessed(cobc::log::LogLevel, cobc::log::Header::MessageId, A&) :bool
getRequiredBytes(int32_t&) :std::size_t
getMutex() :cobc::rtos::Mutex *
lockAppBuffer : MutexGuard
MutexGuard(Mutex&)
pushHeader(cobc::log::BufferBase&, MessageId, ApplicationId, cobc::log::LogLevel, std::size_t) :bool
loop Enough free space in buffer ?
[applicationBuffer.getFreeBytes() < (headerSize + dataSize)]
popPacket()
loop
pushByte(uint8_t)
incrementPacketCount()
pushElement(cobc::log::BufferBase&, int&)
encode(int32_t&) :uint32_t
loop
pushByte(uint8_t)
MutexGuard(Mutex&)

Figure 17.8: Sequence Diagram showing the Collection of Monitoring Messages by the `Application Loggers`

- Thread ID
- Log level
- Message counter (with respect to the application)
- Time stamp
- Header size
- Data size

Here the responsible `Header`-class (see Figure 17.7) is implemented as such that only those parameters have to be provided by the application logger which cannot be determined at run-time: message ID, application ID, log level and the header size. The remaining data is provided by other classes of the flight version and/or by using functionalities of the libCOBC library, e.g. for determining the time stamp or the thread ID (see Section 17.2.5 above.

17.3.3 *Pre-Processing Process*

The introduction of the pre-processing of all source files containing log statements shall not only reduce the size of the final boot image (cf. to Section 18.5.3.2), but – even more important – is supposed to reduce the bandwidth utilization as well (see Section 18.5.5.2). The pre-processing approach tries to avoid the transmission of as many constant and/or redundant data as possible which is especially useful for monitoring messages containing long strings. The details of the pre-processing are explained in [28, sec. 5.1], in this section an overview of the key steps involved in the process is given.

The pre-processing is already integrated within the automatic compilation process of the flight version: herein all source files will be scanned in order to find log statements of the F/W. If the preprocessing is *not* done beforehand the compiler will complain.

All contained format strings within a log statement will be replaced by a unique ID which will be stored for each processed monitoring string within a database – together with the following additional information:

- Message ID
- Format string
- Log level
- Filename
- Line number
- Number of start column

Except for the log level – which can be changed by TC – all of the above listed information is static and can therefore safely be extracted during pre-processing in order to avoid the inclusion into the data to be transmitted via the radio link. This is already a major advantage over the prototype: herein this kind of information was not only not available, but – if additionally added – would have had to be integrated completely into the middleware messages which would have made the bandwidth utilization by the F/W even worse.

Listing 17.3: `log`-Statement of the F/W's Flight Version before and after the Pre-Prossing Step

```
// Before Pre-Processing
applicationLogger.log(DEBUG, "Current counter value: %u32", cnt);
// After Pre-Processing
applicationLogger.logPreprocessed<uint32_t>(DEBUG, 26, cnt);
```

As it can be seen in Listing 17.3, depending on the format string specifier a corresponding template parameter (here: `<uint32_t>`) is added to the `log`-method. This enables the the recognition of wrong specifiers already at compile-time. In addition to the type checking by using data traits (see Paragraph C++ Traits above) this adds even more type safety to the F/W's flight version.

At the very end the pre-processing finalizes by renaming the processed `log`-methods (new name: `logPreprocessed`) and the processed `cpp`-files (new prefix: `pre.cpp`) which are then ready for the normal compilation by the C++ compiler.

Of course, once the monitoring data is received on ground the post-processing also has to include a decoding step in order to restore the original format string. Therefore, all decoded data has to be inserted at the right place into the format string.

17.3.4 *Application Logger*

Besides the pre-processing capability explained in Section 17.3.3, the introduction of application loggers is the most significant improvement of the F/W's flight version in comparison to its prototype. Their main tasks are to provide a better configuration on application level (see Section 17.2.2.4) and to manage the buffering of the application monitoring data in a deterministic way (see Section 17.2.6). For doing this every application which wants to use the F/W's flight version has to use the `ApplicationLogger` shown in Figure 17.7 in order to create its own logging instance. As in the prototype, where every application had to inherit from the central logger in order to use the logging functionality, the usage of the Monitoring F/W is an optional feature for the application programmer.

17.3.5 *Possibilities for Adaptions and Extensions*

This section gives a short overview about possible further developments of the Monitoring F/W flight version.

HISTORICAL DATA FROM CENTRAL BUFFER The switching to the pre-defined application's FDIR log level (see Paragraph FDIR Log Level above) also offers some insight into historical monitoring data. Nevertheless, during normal operation the central buffer discards all monitoring messages with insufficient log levels. To give an example, if the global log level is set to ERROR, every monitoring data corresponding to the WARN, INFO, or DEBUG is discarded (for the available log levels confer to the next Section 17.4). If the onboard storage capacity permits, these kind of monitoring data could be flagged and stored for enhanced historical debugging purposes.

ADDING NEW APPENDERS Like the F/W's prototype, flexibility in terms of sending the monitoring data to connected data sinks and formatting them accordingly (see Section 17.2.2.1) is also an important issue for the flight version. Appenders can be easily added/implemented by inheriting from the class `DataAppendeBase` (see Figure 17.7 and also [28, sec. 5.3.2]). As it was implemented for the prototype (see e.g. the Framework Appenders Section 16.4.3.2 of use case 3), multiple appenders receiving the monitoring data can be connected at the same time to the `DataReceiver`. Currently the only appender implemented is the `DataAppenderConsole` which prints out monitoring data to the console of e.g. the application developer using standard printf. An example of the resulting output can be seen in Listing 17.4.

Listing 17.4: Example Output of the F/W's Flight Version using a console-based Appender

```
[DEBUG] [MsgId:14] [MsgCnt:34] [AppId:1] [ThreadId:2237]
[TimeStamp:1171846833] [DataSize:8] [FileName:application1.cpp]
[LineNumber:56] [Position:19] Value of d is 123456.789
```

This implementation is very similar to the use cases 1 and 2 (see Sections 16.4.1 and 16.4.2) of the F/W's prototype where the behavior of printf-debugging was simulated as an end-to-end scenario. As it was done for the prototype in use case 3 (see Section 16.4.3), a graphical (GUI) appender for the flight version is currently in preparation.

CHANGING THE COMMUNICATION TECHNIQUE Following the *Separation of Concerns*-principle (cf. to Section 17.2.2.1) also means to be able to easily exchange to communication technique used for transporting monitoring data. For doing this the implementation of the F/W's flight version is well prepared: as it is shown in Figure 17.7 only the base classes `ControllerBase`, `ControllerLinkBase`, `DataTransmit-`

`terBase` and `DataReceiverBase` which are interfacing with the boundaries *Telecommand port* and *Telemetry port* have to be used and inherited in order to realize new communication ways. Currently only UDP is supported here (cf. to the test Section 19 later on), but an integration of the monitoring packages into corresponding CCSDS/PUS services (see Section 20.2) is currently being carried out and can be done most comfortable with this programming interface.

17.4 CONFIGURATION

One of the valuable features of the Monitoring F/W is its configuration at compile- as well as at runtime. The multiple options available for the prototype have been explained in Section 16.3. The scalability in terms of performance and functionality also holds for the flight version. As a mission critical piece of S/W special emphasis was put on considering several security mechanisms to prevent failures – either by extensive testing (cf. to the next Chapter 19) or by prevention against unintended misconfiguration (cf. e.g. to Paragraph Deterministic Buffer Handling on Page 141).

17.4.1 *Configuration at Compile-Time*

Like for the prototype, also the flight version can be easily configured at compile time using the corresponding header file `parameters.h`. Herein most of the static constant attributes are affecting the behavior of the embedded part of the flight version which is shown in the upper half of Figure 17.7. To give some examples, the priority, stack size & period for the central thread can be adapted, the TM/TC classes responsible for communication can be fine-tuned as well as the monitoring data can be changed in terms of its structure (header-specific settings) and size (e.g. by setting the maximum packet size). Especially the last two options have a great impact on the saving of bandwidth. For details about all available configuration options available at compile-time see [28, sec. 5.2.10]) (esp. Table 5.4) and also the listing of the complete parameter file seen in Listing A.1 in Appendix A.6.

17.4.2 *Configuration at Run-Time*

One of the most important requirements on the flight version is to design its architecture as such that the granularity as well as the category of monitoring data can be configured at run-time – at global and on subsystem (=application) level (see Appendix A.5 and also refer to Figure 3.2).

This is possible by sending TM packets containing the F/W's new configuration settings[71] [see 28, sec. 5.3.8 and fig. 5.9]. Changeable configuration types are

- the global log level,
- the global FDIR log level,
- a specific application's log level,
- a specific application's FDIR log level,

71 Currently only UDP is supported is a carrier protocol. Cf. to Section 19 for more details.

- the period (in milliseconds) of the central thread,
- the quota of monitoring data (=transfer limit) for a specific application logger, and
- the resetting of the central buffer, a specific application's buffer or all buffers.

The currently supported log levels are the same as in the prototype (see Section 16.3.2 – namely DISABLED, ERROR, WARN, INFO and DEBUG. Each of them corresponds to an integer value, here ranging from 0 (DISABLED) to 5 (DEBUG). Therefrom the terms *high* or *low* in combination with the log level are derived: a higher log levels correspond to higher integer values. The decision on which log level to choose for a specific monitoring message is on behalf of the S/W developer. Typically higher log levels correspond to more in-depth and fine-granular debug information, whereas lower log levels are used only in case of e.g. a severe failure of the component.

Part VII

CASE STUDIES & EVALUATION

This part concentrates on the evaluation of the Monitoring Framework through several use cases and real-world scenarios.

18 TESTING THE MONITORING FRAMEWORK PROTOTYPE

The following tests are based on different example scenarios for the usage of the Monitoring F/W prototype in order to investigate, how they perform in comparison to e.g. the usage of printf statements. These scenarios were introduced and prepared as use cases in chapter 16.4. We will start with the testing of use case 1 which demonstrates that the functionality of the Monitoring F/W can be used in order to replace the behavior of the normally used RODOS `PRINTF`-method. Afterwards the more demanding use case 2 is set up in order to test how the prototype behaves under an increasing load of the OBC. The tests are completed with the use case 3, the real-world scenario.

STRUCTURE OF TESTS In the following chapters the use cases will be tested in different scenarios – once with and once without the usage of the Monitoring F/W. The tests are performed in three different areas where changes are to be expected[72]:

TESTING THE STATIC CHANGES: measuring the impact of using the prototype by counting the additional of lines of code to be compiled, and by looking at the size of the resulting boot image.

TESTING THE DYNAMIC CHANGES: measuring the impact of the Monitoring F/W prototype on the runtime behavior of the relevant system threads by using different profiling tools. Relevant threads are those associated to the RODOS scheduler, the Monitoring F/W and the transport of messages (e.g. over UART).

TESTING THE COMMUNICATION CHANGES: measuring the increased load on the RODOS middleware by the transfer of additional messages coming from the Monitoring F/W.

In addition to the results presented at the end of each test section, at the end of each use case section there are also dedicated sections which compare the results from the different use case tests with each other. Finally, at the end of this prototype test section the results from all three use cases are summarized and compared with each other.

FOCUS AND QUALITY OF TESTS The tests performed on the Monitoring F/W prototype are focused on verifying the following two core statements:

72 These different test areas are quite typical for testing a F/W which is added to an existing S/W. For example, a similar testing approach is also described in the paper of Sreenuch et al. [78].

1. The functional behavior of the three use cases defined in Section 16.4 is correct.

2. Using the Monitoring F/W prototype does not significantly increase the resource consumption (load on memory, processor & communication) in the areas identified in Paragraph Structure of Tests on Page 156.

After the final analysis of all test results at the end of this chapter, we will address these core statements again in Section 18.6 in order to show, if they can be fulfilled by the Monitoring F/W prototype[73].

We will concentrate on validating the general functional behavior and – at the same time – having a rough look at the resource consumption. Therefore the test results were analyzed on a phenomenological basis. The main emphasis was put on comparing the results of the test runs to each other and analyzing the relative changes, rather than going into a fine-granular in-depth analysis of the results. We will not do a comprehensive analysis of e.g. which formula should be used for determining the sampling size or the stopping criteria of the tests as this would be beyond the scope of this test chapter[74].

The general goal to find the smallest sample size and shortest sampling time to provide the desired confidence in the test results was fulfilled on an empirical basis: it showed up that after a test run of 10 minutes duration the profiling results do not alter significantly. Nevertheless, we followed the general test strategy given in [37, p. 26] during evaluation of the prototype.

PROFILING TECHNIQUE It has to be mentioned that the two performance analysis tools used – GRMON and GPROF – slightly differ in the way they perform their measurements within the target environment.

Difference of profiling with GRMON and GPROF

GRMON communicates with the LEON Debug Support Unit (DSU) and allows a non-intrusive debugging of the complete LEON-based

73 A simple functional test of the Monitoring F/W prototype was also performed in [63]. In that test, a single-threaded application was prepared to prove the basic functionality of the F/W in a purely virtual environment. The main differences to the undertaken tests within this thesis are that:

- our test cases are far more elaborated and realistic,
- our test cases are designed for comparing the results with and without the usage of the F/W,
- our tests are also performed on embedded H/W, and
- our tests are not only concentrated on the functional behavior, but do take into account the resource consumption as well.

74 For a detailed overview of the field of computer systems performance analysis please refer to the comprehensive book of Raj Jain ([37]). For example, Section 25.5. is about determining the appropriate sampling size, whereas Section 25.5. answers the question how to find the sufficient stopping time of the tests.

target system. The profiling feature in GRMON is a statistical profiling. It will collect samples of which code is currently executed as often as possible, whereas the debug link is the bottleneck. Therefore it is recommended to use the fastest of the debug links that are supported by the system under test in order to achieve to highest possible sampling rate.
On the other hand, the usage of GPROF for profiling Unix/Linux applications is bound to a hybrid approach of instrumentation and sampling: to gather caller-function data, the instrumentation code is automatically inserted into the program code during compilation by using the GCC compiler option ′-pg′. Please cf. to Section 12.2.1 for details about these monitoring techniques.

Both techniques have drawbacks and advantages and it cannot be said which one is better: the non-intrusive measurement of GRMON allows a good measurement of the sample ratio for each function without any influence of the runtime behavior of the function. The total time spent within each function can be derived from this ratio. On the other hand, it does not allow "measuring from the inside": in the worst case it may happen, that the sampling rate of GRMON and the calling rate of a specific function are slightly shifted. This may result in a scenario, that a frequently used function will never appear in the outcome of GRMON′s profiling. Or – being not so extreme – that the measured sampling rate is just within the wrong order of magnitude. This takes us directly to the advantage and – at the same time – the drawback of GPROF: as instrumentation of the source code always means influencing the runtime-behavior of the system under test (which should be avoided), this will nevertheless allow an "inside"" measurement. The results are precise measurements of the sampling rates of the called functions and – in contrast to GRMON – the exact time spent within each function.

GRMON: non-intrusive but maybe not so exact

GPROF: exact measurement but influence on runtime behavior

SETTING THREAD PRIORITIES For the performance tests of the Monitoring F/W prototype described in following sections the priorities of the main threads `LogOutputThread` and `LogInputThread` were set to the same value as the rest of the applications running in the system. The priorities and all other values used for configuration of the F/W are contained in the `param.h` file, to be seen in Listing 16.1.

This test setup is the result of some experiments with different priorities which were undertaken before doing the in-depth performance measurements contained in the following sections. The experiments were undertaken in order to make a decision, how to set the priorities of the threads involved in the tests of the different use cases of the prototype. They exposed, that the overall functional behavior of the F/W strongly depends on the application load:

For use case 1 (cf. to Section 16.4.1), where the system load caused by the application is very low, the priorities of the Monitoring F/W threads can be the same as the rest the threads within the embedded system.

This changes drastically when the load produced by the applications increases in a significant manner: if the priorities of Monitoring F/W threads are lower or equal to the priority of the load-producing threads, the transmission rate of monitoring messages first decreases and – after some seconds – goes down to zero (no output of the Monitoring F/W). This was tested by using use case 2 (cf. to Section 16.4.2) where the application load consumes most of the processor time and the idle-tread completely disappears within the profiling results. Only when the F/W threads were prioritized higher than the applications, monitoring messages were transmitted: the higher the difference between the priorities, the higher is the processor's idle time and the output of the Monitoring F/W.

18.1 LINES OF CODE

While most of the tests to be performed on the different target platforms have to be described in the corresponding sections within this thesis, the counting of the LoC can be done beforehand. This purely static test only depends on the source code *before* compile time and therefore stays the same for all test scenarios in the following Sections 18.2, 18.3 and 18.4.

The counting of the LoC was done with the tool *Universal Code Lines Counter* (version 1.1.6)[75] , using the RODOS version 111[76]. Within this comparison we concentrate on the *netto* LoC, meaning that blank lines or lines containing only comments were not counted.

18.1.1 *LoC of RODOS Core Library*

Only the LoC contained within the RODOS core layer were taken into account. All other RODOS layers were not included – neither the H/W layer (containing the RODOS ports to the various H/W platforms and operating systems) nor the middleware and management layers (cf. to Section 5.2.1.2 and Figure 5.5):

	Netto LoC	File size (in bytes)
RODOS API	807	96823
RODOS SRC	989	66203

75 `http://www.ab-tools.com/en/software/universalcodelinescounter`

76 Version 111 was chosen because the file system structure ("api"-, "src"- and "support"-directories) of the Monitoring F/W in this version is similar to the structure of RODOS itself (cf. to Section 18.5.1 for further remarks on the different RODOS versions used during testing).

18.1.2 *LoC of LOG Library*

Regarding the Monitoring F/W prototype, only the embedded part was tested. The F/W parts which are normally running on the non-embedded side (e.g. the development host) and stored under the "support"-directory are not taken into account (they mainly contain the output-parts of the F/W, but also the implemented appenders and layouts, cf. to Section 16.1):

	Netto LoC	File size (in bytes)
LOG API	145	4250
LOG SRC	728	24813

18.1.3 *LoC of Application Code*

It was already explained in Section 16.1.2 that the user interface of the Monitoring F/W prototype is designed to replace the PRINTF debug statements of the RODOS OS by a similar, one-line statement to which the application developer is used to. As it can be seen at first glance by comparing the two code snippets from the `PowerControl` application of use case 2 contained in the Listings 18.1 and 18.2 no additional lines were added to the application code. Looking at Line 6 in both listings, the PRINTF-statement is simply replaced by the `log`-method of the Monitoring F/W. Therefore we can conclude that there are no changes in the LoC within the *Application Layer* of RODOS which is depicted at the top of Figure 5.5.

Listing 18.1: Code Snippet from the `PowerControl` Application of Use Case 2, using the `PRINTF`-method of the RODOS Debug Interface

```
1 class BatteryCurrentSubscriber : public Subscriber {
2 public:
3     BatteryCurrentSubscriber() : Subscriber(battery_current, "
          PowerControl") { }
4     long put(const long topicId, const long len, const void* data
          ,  const NetMsgInfo& netMsgInfo) {
5         batteryCurrent = *(long*)data;
6         PRINTF("POWERCONTROL: Battery Current (mA): %ld\n",
              batteryCurrent);
7         return 1;
8     }
9 } batteryCurrentSubscriber;
```

Listing 18.2: Code Snippet from the `PowerControl` Application of Use Case 2, using the `log`-method of the Monitoring F/W

```
class BatteryCurrentSubscriber : public Subscriber, private LOG::
    Logger {
public:
  BatteryCurrentSubscriber() : Subscriber(battery_current, "
      PowerControl"), Logger(&powerApp) { }
        long put(const long topicId, const long len, const void*
            data,  const NetMsgInfo& netMsgInfo) {
            batteryCurrent = *(long*)data ;
        this->log(info) << "Battery Current (mA): " <<
            batteryCurrent << endl;
        return 1;
  }
} batteryCurrentSubscriber;
```

18.2 TEST RESULTS FROM USE CASE 1

Within this section the use case presented in Section 16.4.1 will be tested regarding its performance and resource consumption. It is a single node scenario, meaning that there is one embedded computing node, simulating the satellite OBC.

A short overview of the H/W test environments for this use case can be seen in Figure 18.1. The following sections contain tables with the corresponding test fixtures, describing each of the test setups in detail.

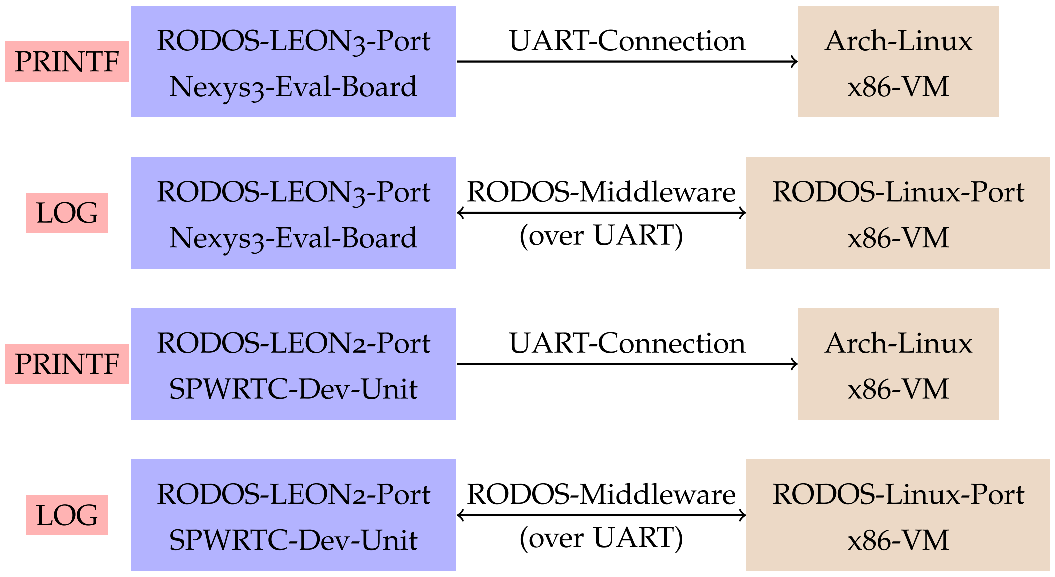


Figure 18.1: H/W Test Environments for Use Case 1

For each test environment the example using normal PRINTF is tested as it is presented in Section 16.4.1.1, afterwards the functionally equivalent example – using the Monitoring F/W prototype in order to simulate the PRINTF behavior – is tested in the same way (see Section 16.4.1.2). The results of both tests are then compared.

18.2.1 *RODOS-LEON3-Port, Nexys3-Eval-Board*

For the first test we are using the Nexys3 embedded development board and its Spartan 6 FPGA. The FPGA is configured with a LEON3 space processor. A RODOS ELF file is loaded onto this target system and executed. As it is depicted in Figure 18.2, the UART DBG debug port is used by GRMON Professional in order to upload, start and profile the RODOS *tst-sat* boot image. Communication to the desktop PC running Linux is established through the UART COM-port[77].

77 The desktop PC is actually a VM, which provides a complete HAL to the OS running inside the machine.

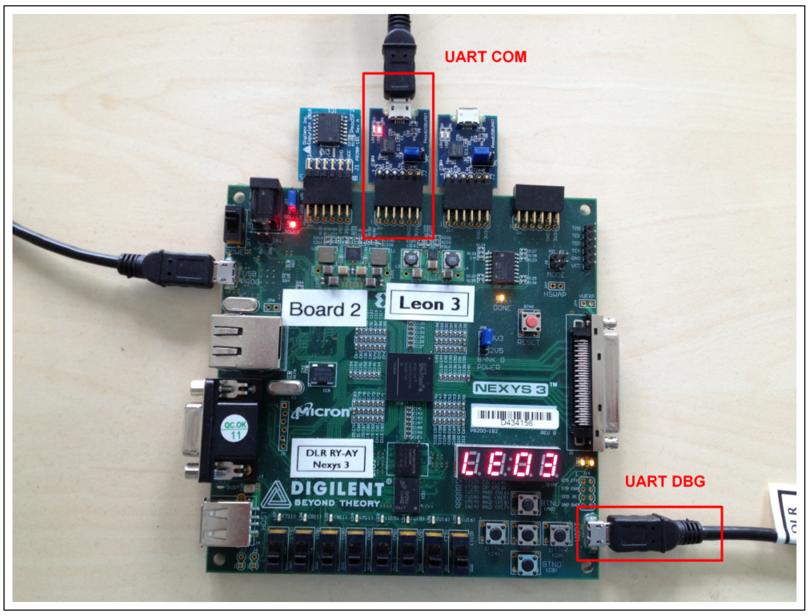


Figure 18.2: The Nexys 3 Development board.

18.2.1.1 *Test Fixture*

Table 18.1 shows the setup (fixture) of the use case tests: the involved H/W and S/W as well as infos about the test tool and the used boot image.

18.2.1.2 *Test Results - PRINTF*

After loading the RODOS boot image *tst-sat* to the target H/W and starting it with GRMON profiling enabled, the PRINTF output is grabbed via the UART COM port of the Nexys3 board using a terminal program[78] under Linux running on the desktop PC. Listing 18.3 shows this output of the *tst-sat* boot image.

After ten minutes the `tst-sat` boot image was interrupted on the target H/W with `CTRL-C` and GRMON was prompted – by using the `prof`-command – for printing out the profiling results of the test run. The resulting output contains an overview of the called functions of the boot image and their samples in absolute number and percentage. The bar chart in Figure 18.3 gives an overview of the results.

78 Most of the test results were grabbed using the *picocom* terminal emulation program (`https://code.google.com/p/picocom`).

Test Name Test Date Test Duration	Use Case 1 13.7.2013 10 minutes
Computing Node 1 (Satellite-Simulator)	
Hardware	Digilent Nexys 3 Board, Xilinx Spartan-6 LX16 FPGA @ 60Mhz, LEON3 SPARC v8 Processor
Software	RODOS RODOS-110.1 OS Version RODOS-Leon3-v1
Computing Node 2 (Groundstation-Simulator)	
Hardware	Orcale VirtualBox Machine (Emulated Hardware): 128KiB BIOS memory, 2019MiB System memory, processor Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz, 17GB VBOX hard-disk
Software	Linux devel 3.10.6-2-ARCH #1 SMP PREEMPT Tue Aug 13 10:20:52 CEST 2013 i686 GNU/Linux
Test-Tool	
GRMON-Version	GRMON LEON debug monitor v1.1.50 professional version
GRLIB-Version	GRLIB build version: 4113
Test-ELF File on HW (tst-sat PRINTF): section: .boot at 0x40000000, size 5404 bytes section: .text at 0x4000151c, size 37180 bytes section: .data at 0x4000a658, size 3880 bytes total size: 46464 bytes read 335 symbols	
Test-ELF File on HW (tst-sat Monitoring Framework): section: .boot at 0x40000000, size 5436 bytes section: .text at 0x4000153c, size 43556 bytes section: .data at 0x4000bf60, size 4488 bytes total size: 53480 bytes read 367 symbols	

Table 18.1: Test Fixture for the Test between LEON3-based H/W & Linux

18.2.1.3 *Test Results - Monitoring Framework*

After starting the simulation, RODOS shows some initial startup output [79]:

79 For this initial startup output PRINTF – and not the Monitoring F/W – is still being used. If the F/W will become an integrated part of RODOS this might change in the future. Until now the Monitoring F/W is deployed as a support library for the purpose of providing monitoring capabilities to the applications running on-top of RODOS.

Listing 18.3: Use Case 1: Output tst-sat w/o Monitoring F/W on LEON3-based H/W (only PRINTF)

```
RODOS RODOS-110.1 OS Version RODOS-Leon2-v1
Loaded Applications:
         10 -> ’Topics & Middleware’
         20 -> ’Gateway’
       1100 -> ’PowerControl’
       1100 -> ’BatteryInterface’
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =    1002 Stack =  10000 gateway:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
BigEndianity = 1, cpu-Arc = leon3, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 208

Node Number: HEX: FFFFFFFF Dec: -1
-----------------------------------------------------
Default internal MAIN
-------------- application running ------------

PWR: Time since Boot (sec): 1
PWR: Battery Current (mA): 914                        ⋮
```

- Listing 18.4 shows the output of the *tst-sat* boot image which was loaded and started by GRMON. A terminal program on Linux running on a desktop PC was used in order to grab the input from the UART COM port of the Nexys3 board.
- Listing 18.5 shows the output of the *tst-ground* boot image which is running under Linux and communicating with the embedded *tst-sat* image on the Nexys3 board using the RODOS middleware for transportation of monitoring messages.

After startup, the monitoring information is sent via the RODOS middleware to a connected desktop computer which is acting as the simulated ground station or EGSE environment.

Like practiced while testing the PRINTF-version (cf. to Section 18.2.1.2), the simulation was stopped after 10 minutes and the internal profiling mechanism of GRMON Professional outputs the samples of the called functions within the RODOS boot image. The bar chart in Table 18.4 presents this output in a re-worked format, showing the sample ration for all called functions.

18.2.1.4 *Test Analysis & Comparison*

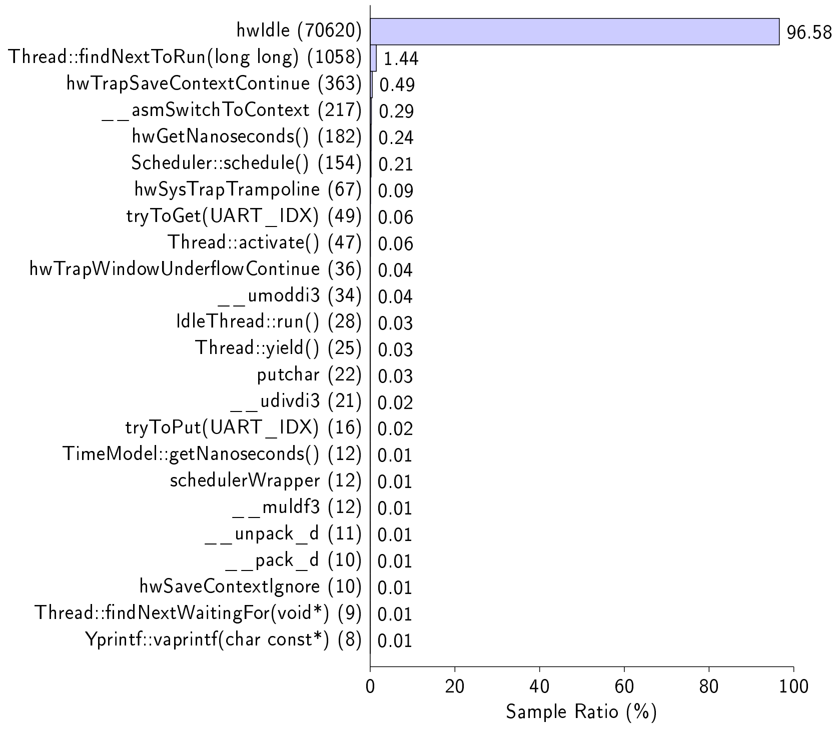


Figure 18.3: Sample Ratio of Use Case 1 on LEON3-based H/W with PRINTF (GRMON Profiler Results)

BOOT IMAGE SIZE What can be easily recognized at a first glance from the information given about the boot images in Table 18.1 is, that the two boot images differ in size as well as in the number of symbols. This is mainly the result of the Monitoring F/W being linked into the final boot image. This results in a 15% higher increase of the boot image size and a 9% increase of symbols.

PROCESSOR LOAD The increase of the processor load due to the usage of the Monitoring F/W can be measured by having look at the idle thread: this thread is doing nothing and yields all the time, meaning that it is willing to "give up the CPU" as soon this is requested by the thread scheduler (e.g. when a different thread is selected to be run). Therefore the usage percentage of this thread is directly related to the time the processor is used by other tasks, meaning that a 100% usage of the idle thread is equivalent to a processor which usage percentage by other threads is near zero.

By using the Monitoring F/W two additional threads were started on the embedded target platform: the `LogInputThread` which is responsible for collecting all monitoring messages being published over the RODOS middleware, and the `LogControlThread` which is responsible for receiving control messages for the Monitoring F/W and the runtime configuring of it. In Chapter 16 and esp. Section 16.4.1 herein a detailed overview of the architecture of the initial prototype of Monitoring F/W and the additional threads is given.

Listing 18.4: Use Case 1: Output tst-sat with Monitoring F/W on LEON3-based H/W

```
RODOS RODOS-110.1 OS Version RODOS-Leon2-v1 Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
       1100 -> 'PowerControl'
       1100 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 64.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =      50 Stack =  10000 LogControlThread: Receives and realizes log controls
   Prio =     100 Stack =  10000 LogInputThread: Collects and publishes log messages
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =    1002 Stack =  10000 gateway:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
BigEndianity = 1, cpu-Arc = leon3, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 208

Node Number: HEX: FFFFFFFF Dec: -1
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
```

Increased Scheduler Load due to additional Threads

Load from Scheduler The inclusion of these two additional threads in the boot image results in an increase of the processor usage: the sample ratio of the `hwIdle`-thread decreases from 96% in the PRINTF-version of use case 1 down to 78% when using the Monitoring F/W prototype in order to simulate the behavior of the usage of PRINTF-statements. As a comparison of the Diagrams 18.3 and 18.4 shows, a major part of the processing time is being consumed by the scheduler and its triggered context switches. This is most likely caused by the additional Monitoring F/W threads which have to be invoked by the scheduler: the functional behavior of the included applications used here is kept at a minimum, and they are mainly busy with publishing monitoring data which does not consume a lot of processing time (see next paragraph). Therefore the scheduler is invoked quite frequently. The following list shows the five most significant, scheduler-related threads and their increases in sample rate when using the Monitoring F/W:

- `Thread::findNextToRun(long long)`:
  1.44% PRINTF $\Rightarrow$ 7.66% Monitoring FW (increase of $\approx$ 430%)
- `hwTrapSaveContextContinue`:
  0.49% PRINTF $\Rightarrow$ 3.05% Monitoring FW (increase of $\approx$ 510%)

Listing 18.5: Use Case 1: Output tst-ground (getting Monitoring Messages from LEON3-based H/W)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424299 -> 'logOutputApp'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
 log  Id = 42 len = 60.   -- Subscribers:
     LogOutputThread
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:

Event servers:
  1 TimeEvent managers
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 SimpleLogControllerThread: Provides simple console interface to publish log
         controls
   Prio =     100 Stack =  32000 LogOutputThread: Receives and outputs log messages
   Prio =    1002 Stack =  32000 gateway:

BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
Node Number: HEX: 65D Dec: 1629
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
[DEBUG] PowerControl.PowerControlThre: Time since Boot (sec): 1
[INFO] PowerControl.BatteryInterface: Battery Current (mA): 914
Message dropped because expected size(90) != real size(177)
[INFO] PowerControl.BatteryInterface: (mV): 4542
[DEBUG] PowerControl.PowerControlThre: Time since Boot (sec): 2
[DEBUG] PowerControl.PowerControlThre: Time since Boot (sec): 3
[DEBUG] PowerControl.PowerControlThre: Time since Boot (sec): 4
[INFO] PowerControl.BatteryInterface: Battery Current (mA): 1026
                                                          ⋮
```

- `__asmSwitchToContext`:
  0.29% PRINTF ⇒ 1.87% Monitoring FW (increase of ≈ 530%)
- `Scheduler::schedule()`:
  0.21% PRINTF ⇒ 1.18% Monitoring FW (increase of ≈ 460%)
- `hwSysTrapTrampoline`:
  0.09% PRINTF ⇒ 1.01% Monitoring FW (increase of ≈ 1010%)

Additional Load from Log Threads is negligible

Load from Log Threads The additional threads from the Monitoring F/W do not have a big influence on the processor load. Within the Bar Chart 18.4, the `LOG::LogInputThread::run()` method was sampled only 14 times within the 10 minutes of test run, thus contributing only 0,01% to the overall load of the system. The `LogControlThread` does not appear at all in the test output from GRMON. The simple reason behind is, that during the tests no configuration commands have been sent to the embedded system.

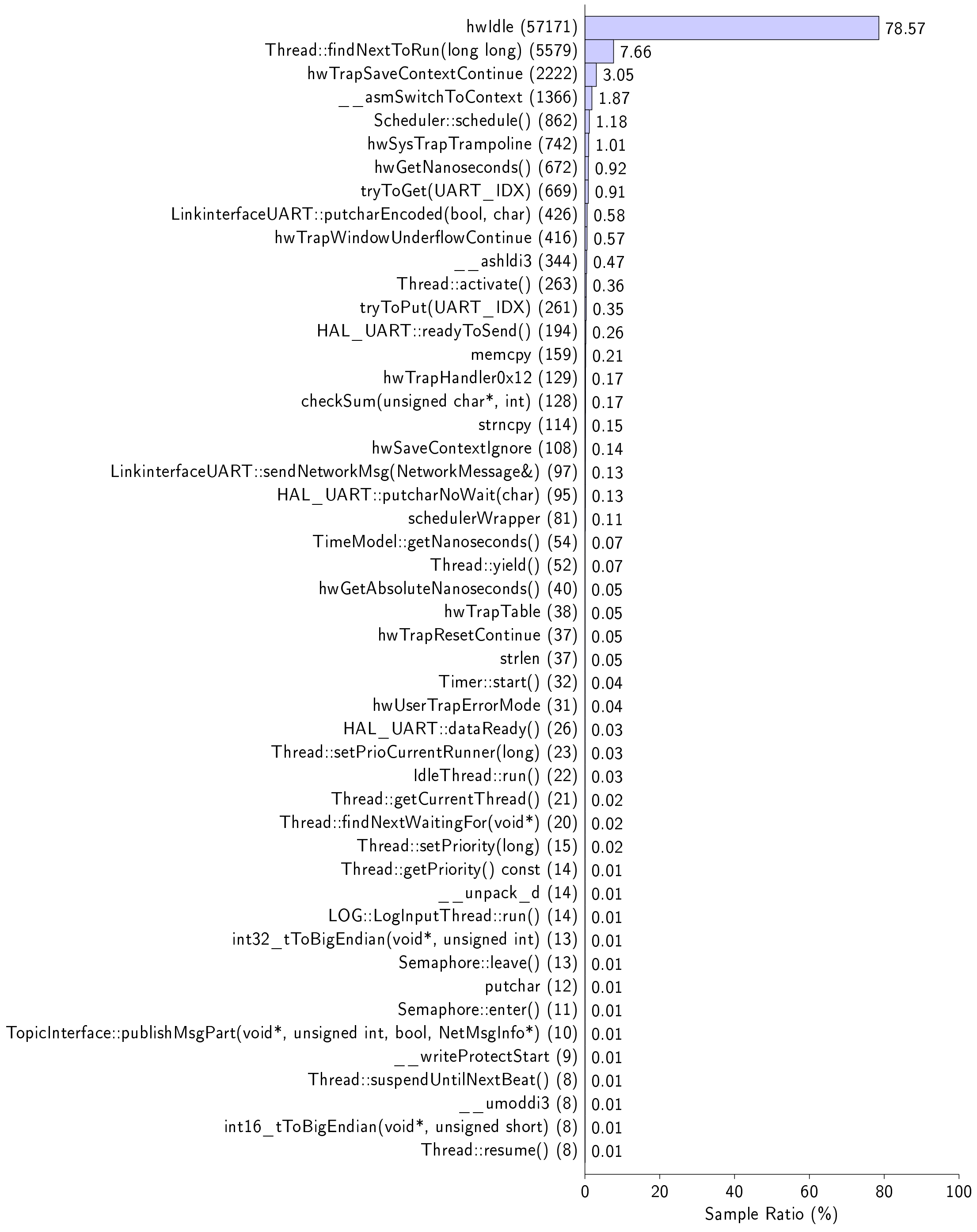


Figure 18.4: Sample Ratio of Use Case 1 on LEON3-based H/W with the Monitoring F/W Prototype (GRMON Profiler Results)

MESSAGE TRANSPORT

Load from Middleware Threads The additional load by the sending of monitoring message is also negligible on the embedded side. This can be seen when looking at the position of UART-related methods in the Bar Chart 18.4 relative to the sampling rates of the other meth-

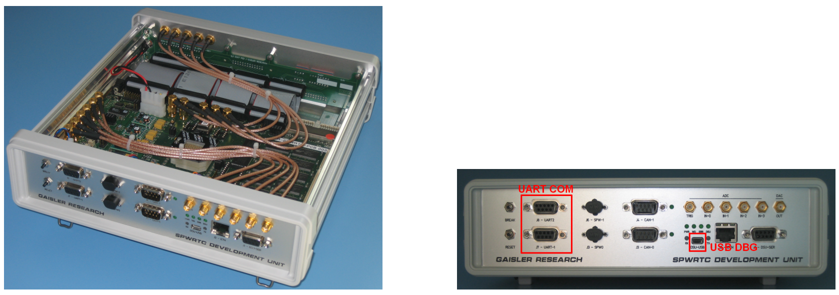
Additional Load from Sending of Network Messages is negligible ods. This also applies to the new methods appearing in the profiling results: `LinkinterfaceUART::putcharEncoded(bool, char)` (sampled 426 times) and `LinkinterfaceUART::sendNetworkMsg(NetworkMessage&)` (sampled 97 times). These methods are implemented in the RODOS interface class `LinkinterfaceUART`, which provides methods to connect to UART-based network- or H/W-interfaces and to enable data transfer. `Linkinterface` is the corresponding base class, from which all H/W dependent implementations must inherit.

Nonetheless, there is a clear increase in the sampling rates of the UART-related methods which also appear in the PRINTF-based scenario on LEON3:

- `tryToGet(UART_IDX)`:
  0.06% PRINTF ⇒ 0.91% Monitoring FW (increase of ≈ 1270%)
- `tryToPut(UART_IDX)`:
  0.02% PRINTF ⇒ 0.35% Monitoring FW (increase of ≈ 1530%)

18.2.2 *LEON2 Processor, SPWRTC Development Kit*

In order to evaluate the Monitoring F/W prototype on another H/W platform relevant for S/C avionics, for the next tests we are using the SpaceWire Remote Terminal Controller (SPWRTC) development unit, as it is depicted in Figure 18.5a. It comes with a LEON2-based custom ASIC and is therefore an ideal testbed for the OBSW and the monitoring F/W prototype. The SPWRTC front panel which can be seen in Figure 18.5b) provides a DSU port, which can be accessed via an Universal Serial Bus (USB) connection. This DSU-USB is used within this test by GRMON Professional in order to upload, start and profile the RODOS boot image. The communication with a desktop PC running Linux is established through one of the panel's UART COM-ports.



(a) SPWRTC Development Unit (b) SPWRTC Front Panel

Figure 18.5: SpaceWire Remote Terminal Controller Development Unit

The procedure for the tests performed on the SPWRTC development unit is the same like before on the Nexys3 board which are described in detail in the preceding Section 18.2.1. Therefore – in order to avoid repetitions – the describing text passages are kept to a minimum.

LEON2 UART HANDSHAKE PROBLEM During this test it turned out that a full end-to-end test consisting of sending monitoring messages from the RODOS instance running on the SPWRTC to the instance running on top of Linux on a standard (emulated) PC was not possible (see output of `tst-ground` in code Snippet 18.8). The reason lies within the non-functional communication chain between these two platforms which in turn is originated in a problem with UART-handshaking of the LEON2.

This problem is already described in detail in [80, sec. 3.12]. To give a short summary of the findings herein, the problem is originated in the fact that the LEON2 UART H/W handshaking is incompatible with many other UART implementations because it is not industry standard compliant and is therefore not able to communicate with the H/W of a standard PC. As the LEON2 UART flow control is completely implemented in H/W, there is also no way to fix this problem in S/W.

18.2.2.1 *Test Fixture*

The test fixture can be seen in Table 18.2.

18.2.2.2 *Test Results - PRINTF*

Listing 18.6 shows the output of the *tst-sat* boot image.

The bar chart in Figure 18.6 gives an overview of the results.

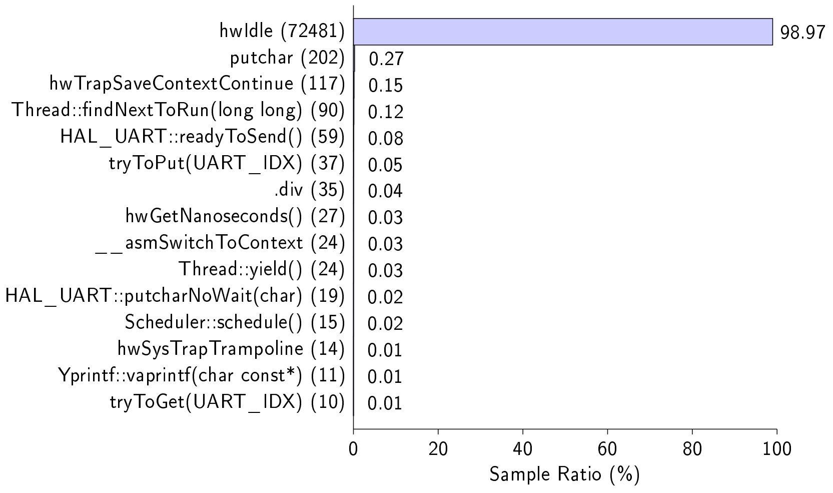


Figure 18.6: Sample Ratio of Use Case 1 on LEON2-based H/W with PRINTF (GRMON Profiler Results)

18.2.2.3 *Test Results - Monitoring Framework*

Listing 18.7 shows the output of the *tst-sat* boot image.

Listing 18.8 shows the output of the *tst-ground* boot image.

Test Name	Use Case 1
Test Date	20.9.2013
Test Duration	10 minutes
Computing Node 1 (Satellite-Simulator)	
Hardware	GR-SPWRTC-DEV Unit, LEON2-FT ASIC @ 30Mhz
Software	RODOS RODOS-111.1 OS Version RODOS-Leon2-v1
Computing Node 2 (Groundstation-Simulator)	
Hardware	Orcale VirtualBox Machine (Emulated Hardware): 128KiB BIOS memory, 2019MiB System memory, processor Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz, 17GB VBOX harddisk
Software	Linux devel 3.10.6-2-ARCH #1 SMP PREEMPT Tue Aug 13 10:20:52 CEST 2013 i686 GNU/Linux
Test-Tool	
GRMON-Version	GRMON LEON debug monitor v1.1.50 professional version
GRLIB-Version	GRLIB build version: 4113
Test-ELF File on HW (tst-sat PRINTF):	
section: .boot at 0x40000000, size 5644 bytes	
section: .text at 0x4000160c, size 31100 bytes	
section: .data at 0x40008f88, size 3848 bytes	
total size: 40592 bytes	
read 316 symbols	
Test-ELF File on HW (tst-sat Monitoring Framework):	
section: .boot at 0x40000000, size 5676 bytes	
section: .text at 0x4000162c, size 37476 bytes	
section: .data at 0x4000a890, size 4456 bytes	
total size: 47608 bytes	
read 348 symbols	

Table 18.2: Test Fixture for the Test between LEON2-based H/W & Linux

The bar chart in Figure 18.7 gives an overview of the test results performed with the Monitoring F/W.

18.2.2.4 *Test Analysis & Comparison*

The analysis and comparison of the LEON2-based tests strongly orient on the results from the LEON3-based system in the previous Section 18.2.1. Therefore the textual description within this section is reduced to a minimum

Listing 18.6: Use Case 1: Output tst-sat w/o Monitoring F/W on LEON2-based H/W (only PRINTF

```
RODOS RODOS-111.1 OS Version RODOS-Leon2-v1
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
       1100 -> 'PowerControl'
       1100 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =    1002 Stack =  10000 gateway:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
BigEndianity = 1, cpu-Arc = leon2, Basis-Os = baremetal, Cpu-
    Speed (K-Loops/sec) = 1563
Node Number: HEX: FFFFFFFF Dec:
    -1-------------------------------------------------------
Default internal MAIN
--------------- application running ------------
PWR: Time since Boot (sec): 1
PWR: Battery Current (mA): 914
                                 ⋮
```

- in order to not repeat longish text passages which would overstrain the reader, and
- in order to focus on the important changes and findings in the test results.

BOOT IMAGE SIZE The usage of the Monitoring F/W results again in an increase of the file size (≈ 17%) and the number of symbols used (≈ 10%).

Listing 18.7: Use Case 1: Output tst-sat with Monitoring F/W on LEON2-based H/W

```
RODOS RODOS-111.1 OS Version RODOS-Leon2-v1
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
       1100 -> 'PowerControl'
       1100 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 64.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =      50 Stack =  10000 LogControlThread: Receives and
       realizes log controls
   Prio =     100 Stack =  10000 LogInputThread: Collects and
       publishes log messages
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =    1002 Stack =  10000 gateway:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
BigEndianity = 1, cpu-Arc = leon2, Basis-Os = baremetal, Cpu-
    Speed (K-Loops/sec) = 1563
Node Number: HEX: FFFFFFFF Dec: -1
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
```

PROCESSOR LOAD

Load from Scheduler The additional number of threads in the system result again (like in Section 18.2.1.4) in an increased scheduler activity due to context switches:

Listing 18.8: Use Case 1: Output tst-ground (getting Monitoring Messages from LEON2-based H/W)

```
RODOS RODOS-111.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424299 -> 'logOutputApp'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
 log  Id = 42 len = 60.   -- Subscribers:
     LogOutputThread
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:

Event servers:
  1 TimeEvent managers
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 SimpleLogControllerThread:
       Provides simple console interface to publish log controls
   Prio =     100 Stack =  32000 LogOutputThread: Receives and
       outputs log messages
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed
    (K-Loops/sec) = 350000
Node Number: HEX: C8B Dec: 3211
--------------------------------------------------
Default internal MAIN
--------------- application running ------------
Message dropped because expected size(872) != real size(100)
Message dropped because expected size(10781) != real size(101)
Message dropped because expected size(872) != real size(101)
                                  ⋮
```

- `Thread::findNextToRun(long long)`:
  0.12% PRINTF ⇒ 0.87% Monitoring F/W (increase of ≈ 615%)
- `hwTrapSaveContextContinue`:
  0.15% PRINTF ⇒ 1.78% Monitoring F/W (increase of ≈ 1010%)
- `__asmSwitchToContext`:
  0.03% PRINTF ⇒ 0.55% Monitoring F/W (increase of ≈ 1600%)

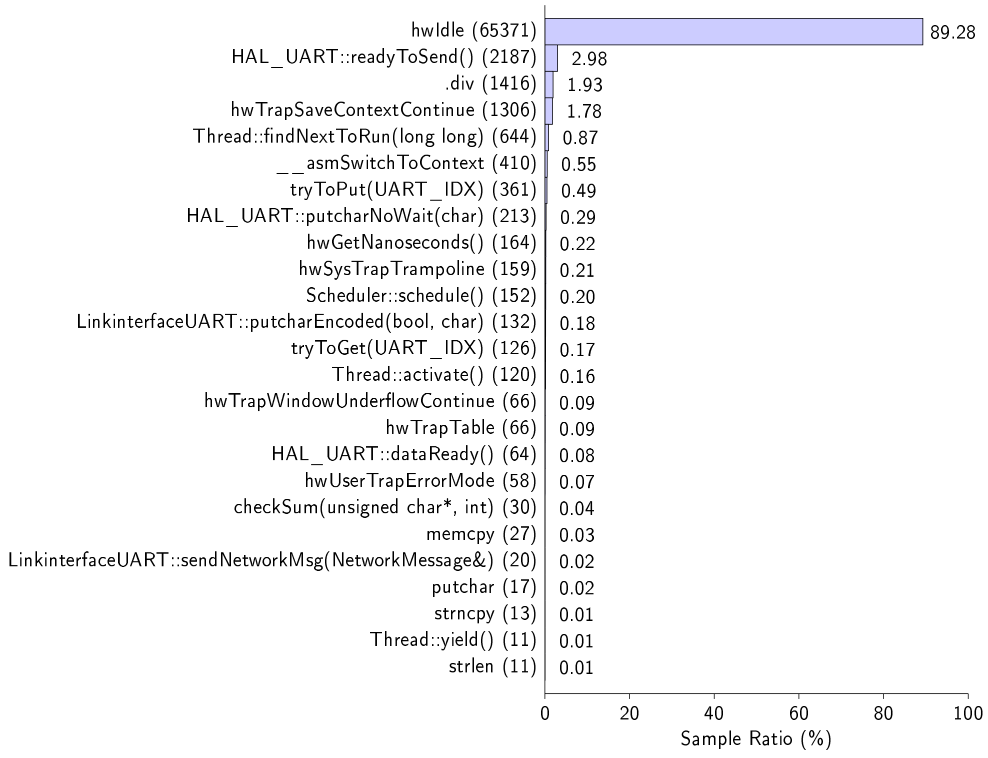


Figure 18.7: Sample Ratio of Use Case 1 on LEON2-based H/W with the Monitoring F/W prototype (GRMON Profiler Results)

- `Scheduler::schedule()`:
  0.02% PRINTF ⇒ 0.20% Monitoring F/W (increase of ≈ 900%)
- `hwSysTrapTrampoline`:
  0.01% PRINTF ⇒ 0.21% Monitoring F/W (increase of ≈ 1130%)

Load from Log Threads The load from the threads of the Monitoring F/W (like `LOG::LogInputThread::run()` cannot be determined, because they do not appear in the Bar Chart 18.7.

MESSAGE TRANSPORT

Load from Middleware Threads Despite the missing messages due to the UART-problem described in the Paragraph LEON2 UART Handshake Problem on Page 171, we will have a look at the influence of the sending of monitoring messages over the RODOS middleware on the runtime behavior of the embedded system. Looking at the Bar Cart 18.7, a comparison of the dominant sampling rate of the `hwIdle`-method against the sampling rates of the methods involved in sending monitoring message over UART shows, that the additional load is negligible on the embedded side. This also implies on the middleware-specific RODOS methods `LinkinterfaceUART::sendNet-`

`workMsg(NetworkMessage&)` and `LinkinterfaceUART::putcharEncoded-(bool, char)`.

Nevertheless, the usage of the Monitoring F/W results in a significant increase in the sampling rates of all methods which are invoked when sending data over the UART interface to the host (development) PC:

- `HAL_UART::readyToSend()`:
  0.08% PRINTF ⇒ 2.98% Monitoring F/W (increase of ≈ 3600%)
- `tryToPut(UART_IDX)`:
  0.05% PRINTF ⇒ 0.49% Monitoring F/W (increase of ≈ 880%)
- `HAL_UART::putcharNoWait(char)`:
  0.02% PRINTF ⇒ 0.29% Monitoring F/W (increase of ≈ 530%)
- `tryToGet(UART_IDX)`:
  0.01% PRINTF ⇒ 0.17% Monitoring F/W (increase of ≈ 1160%)

18.2.3 *Overall Test Analysis & Comparison*

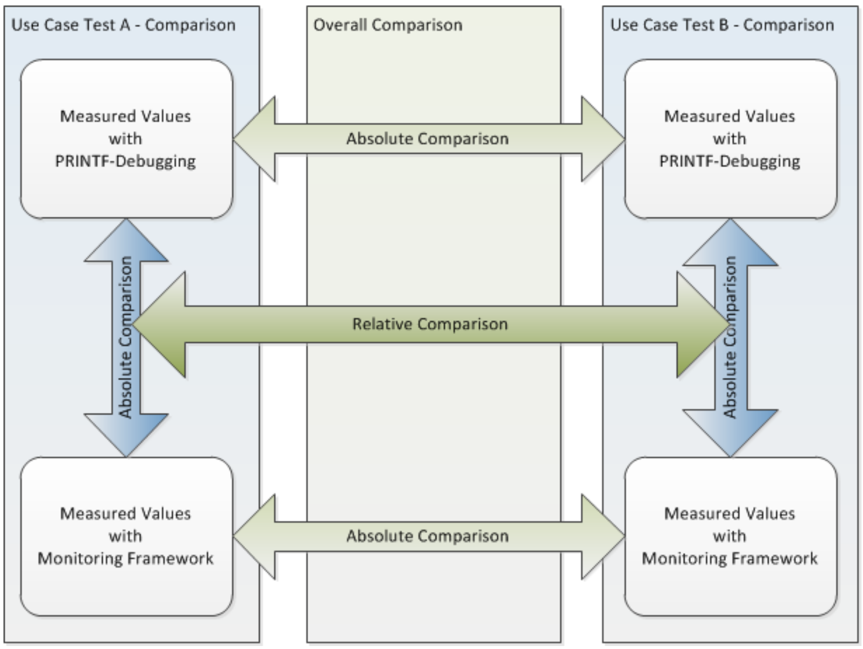

Use Case Test A - Comparison
Overall Comparison
Use Case Test B - Comparison
Measured Values with PRINTF-Debugging
Absolute Comparison
Measured Values with PRINTF-Debugging
Absolute Comparison
Relative Comparison
Absolute Comparison
Measured Values with Monitoring Framework
Absolute Comparison
Measured Values with Monitoring Framework

Figure 18.8: Different Kinds of Comparison of Test results (Generalized Representation)

In this section we will briefly compare the outcome of the performed tests of use case 1 on LEON3- and on LEON2-based H/W with each other. For this, we will have a look at the results in the corresponding Sections 18.2.1.4 and 18.2.2.4, and compare them using two kinds of techniques:

ABSOLUTE COMPARISON For this comparison we will have a look at the *direct measured values* and compare them with each other.

RELATIVE COMPARISON Here we will inspect the *degree of increase* between the measured values with and without the monitoring F/W.

The relationship and context of each type of comparison is also depicted in Figure 18.8.

Though the comparison of the direct measured values is the basis for the corresponding analysis Sections 18.2.2.4 and 18.2.1.4, the values are strongly influenced by platform-dependent specifics. In order to illustrate some examples, the boot image sizes may differ because of the different linked-in libraries from the corresponding BSPs (cf. to Section 5.2.1). The same holds for the measured sampling rates which are also influenced by H/W specific method calls and hence may vary from platform to platform. Moreover, the CPU architecture and clock frequency will surely have a significant influence on the runtime behavior of *number-crunching* S/W applications like e.g. the random number generator of RODOS.

Focus on Relative Comparison

In this section we will therefore mostly concentrate on the relative comparison, focusing on the changes in the increase rates resulting from the usage of the Monitoring F/W on different H/W platforms.

BOOT IMAGE SIZE Comparing the *absolute* values of boot image size and number of symbols with each other, it shows that there is a decrease when switching from LEON3- to LEON2-based H/W. As shown in the following table, the rate of decrease with and without the Monitoring F/W stays almost the same:

LEON3-Test $\rightarrow$ LEON2-Test	Boot Image Size	Number of Symbols
PRINTF version	−13%	−6%
Monitoring-FW version	−11%	−6%

However, there is almost no *relative* increase of the boot image size and the number of loaded symbols when we compare the switching from PRINTF- to Monitoring F/W-based debugging on the different H/W platforms (cf. to 18.2.1.4 and 18.2.2.4). Taken the mean value for both tests, we find a 16% higher consumption of the boot image size and 9% increase of symbols when using the Monitoring F/W as a PRINTF substitute.

PROCESSOR LOAD

Load from Scheduler The absolute sampling rate values are much lower on LEON2-based H/W, which is reflected – among others – by a 10% higher load of the `hwIdle`-threat on LEON2. Nevertheless the relative increase of the scheduler load is significantly when switching from LEON3 to LEON2, as a comparison of the sample rates of the methods related to the scheduler reveals. For some of the threads the increase is more than three times as high (cf. to Paragraphs 18.2.1.4 and 18.2.2.4).

Load from Log Threads As the sampling rate of threads related to the Monitoring F/W is already quite negligible on LEON3-based H/W, these threads do not even appear on the LEON2-based H/W any more.

Message Transport

Load from Middleware Threads Though the absolute values are quite low, the increase of the sampling rates of middleware-specific RODOS methods is significant for both H/W test scenarios of use case 1 when using the Monitoring F/W prototype in order to send the monitoring messages over UART to the host PC. It is also noteworthy, that within the LEON2-based test scenario the values are lower than on the LEON3-based H/W which results in a decrease within the relative comparison of these platform-specific values:

- `tryToPut(UART_IDX)`:
  1530% increase on LEON3 $\Rightarrow$ 880% increase on LEON2
  (relative decrease of 650%)
- `tryToGet(UART_IDX)`:
  1270% increase on LEON3 $\Rightarrow$ 1160% increase on LEON2
  (relative decrease of 110%)

18.3 TEST RESULTS FROM USE CASE 2

After gathering first experiences with the testing of the simple example scenario in use case 1 in Section 18.2, the same tests will be repeated for the more advanced and realistic use case 2 presented in Section 16.4.2. In addition, the H/W-based tests will be complemented by tests under a pure virtual Linux-only environment. However, due to the communication problems which occurred during testing of use case 1 the LEON2-based H/W platform SPWRTC development unit was skipped as an appropriate test environment (see Section 18.2.2 for further details about this problem). Figure 18.9 gives an overview of the H/W test environments to be covered during the following tests.

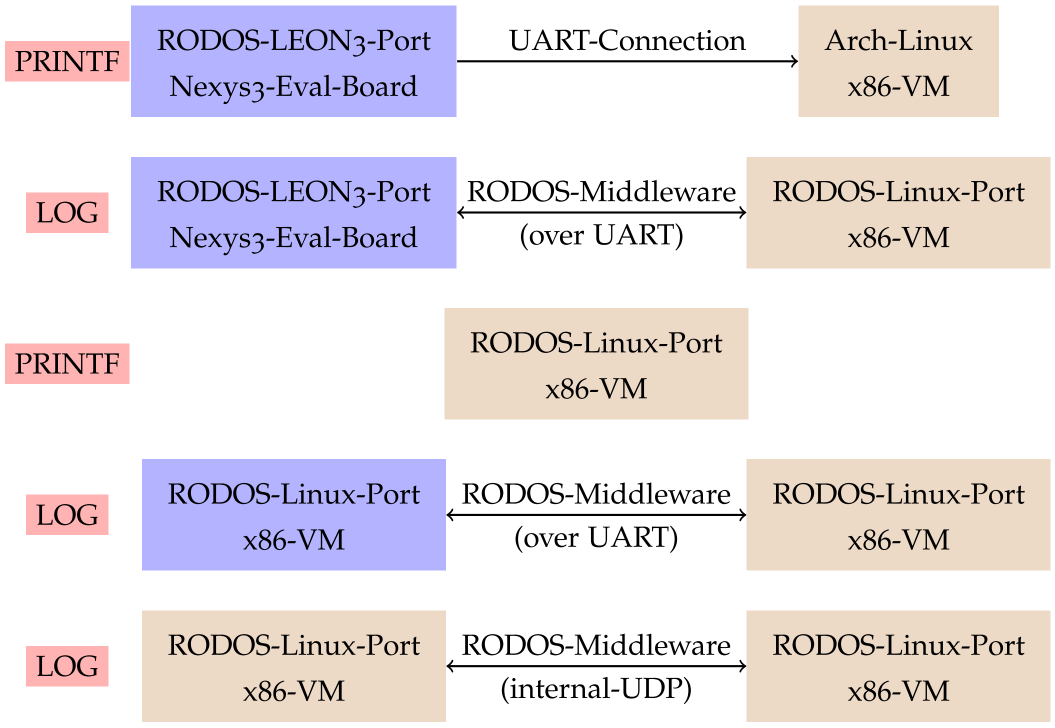


Figure 18.9: H/W Test Environments for Use Case 2

The test procedures performed on use case 2 were the same as with use case 1 (cf. to Section 18.2). All tests are performed with and without the usage of the Monitoring F/W prototype, in the latter case by using only the PRINTF method of RODOS. On the LEON-based platforms, GRMON's profiling functionality is used for profiling (see Section 12.2.1, while in the Linux environment GCC's GPROF[80] serves as the profiling-tool of choice.

In order to make GPROF work correctly, the application under test has to terminate on its own. As normally all application within the satellite's boot image are running in an endless loop (cf. e.g. to Sec-

80 https://sourceware.org/binutils/docs/gprof

tion 5.2.1.2), a slight modification of this kind of implementation is introduced here: the `Main`-application was modified with a `for`-loop, which terminates after the predefined time period used for the tests. Afterwards an `exit(0)`-method is invoked, which stops the execution of the boot image under Linux[81].

18.3.1 *RODOS-LEON3-Port, Nexys3-Eval-Board*

18.3.1.1 *Test Fixture*

The test fixture for LEON3-based test can be seen in Table 18.3.

18.3.1.2 *Test Results - PRINTF*

The PRINTF terminal output from the `tst-sat` boot image can be seen in Listing 18.9.

After expiration of the ten minutes test period the GRMON profiler yields the profiling results which can be seen in the bar chart in Figure 18.10.

18.3.1.3 *Test Results - Monitoring Framework*

The output of the two RODOS instances communicating with each other can be seen in Listing 18.10 (for the embedded part) and Listing 18.11 (for the part running on the development PC).

Table 18.11 reveals the profiling results from GRMON after ten minutes of testing on LEON3-based H/W.

18.3.1.4 *Test Analysis & Comparison*

BOOT IMAGE SIZE The values for the total bytes of the ELF-file loaded on the embedded system on and the number of symbols read are listed in Table 18.3. The usage of the monitoring results in an increase of the boot image size of 30% and in the number of symbols of 24%.

PROCESSOR LOAD What comes immediately apparent when looking at GRMON results with and without the Monitoring F/W presented within the Bar Charts 18.10 and 18.11 is, that the Idle-thread

81 One of the drawbacks of this modification can be seen in some of the outputs of GPROF: the destructors of the instantiated classes were also included in the measurements. In the bar chart and textual call graphs they can easily be identified by having the same name as the class, but with a tilde (~) in front of it. It has to be mentioned, that also RODOS prints out warning messages after invocation of the `exit(0)`-method, as this behavior is not normal for an RTOS of a satellite (in the best case the boot image is started once and runs continuously until the end of life of the S/C).

Test Name Test Date Test Duration	Use Case 2 08.1.2014 10 minutes
Computing Node 1 (Satellite-Simulator)	
Hardware	Digilent Nexys 3 Board, Xilinx Spartan-6 LX16 FPGA @ 60Mhz, LEON3 SPARC v8 Processor
Software	RODOS RODOS-110.1 OS Version RODOS-Leon3-v1
Computing Node 2 (Groundstation-Simulator)	
Hardware	Orcale VirtualBox Machine (Emulated Hardware): Intel Core2Duo E8500 @3.16 GHz; 8GB RAM
Software	Arch Linux x86 (Kernel 3.12.6-1-ARCH)
Test-Tool	
GRMON-Version	GRMON LEON debug monitor v1.1.50 professional version
GRLIB-Version	GRLIB build version: 4113
Test-ELF File on HW (tst-sat PRINTF): section: .boot at 0x40000000, size 5412 bytes section: .text at 0x40001524, size 34876 bytes section: .data at 0x40009d60, size 4144 bytes total size: 44432 bytes read 309 symbols	
Test-ELF File on HW (tst-sat Monitoring Framework): section: .boot at 0x40000000, size 5452 bytes section: .text at 0x4000154c, size 47348 bytes section: .data at 0x4000ce40, size 5112 bytes total size: 57912 bytes read 386 symbols	

Table 18.3: Test Fixture for the Test between LEON3-based H/W & Linux/X86 VM

is not dominant any more: the simulated load – especially from the AOCS application – becomes visible by the increased sampling rate of the RODOS-internal random number generator which is used for this purpose. Nevertheless, the additional load from the scheduler related threads (see next paragraph) seems to be the cause, that the sampling rate of the `randomTT800()`-method is approximately 8% lower when using the Monitoring F/W.

Load from Scheduler The additional Monitoring F/W threads invoked by the scheduler are causing a significant increase in the sam-

Listing 18.9: Use Case 2: Output tst-sat w/o Monitoring F/W on LEON3-based H/W (only PRINTF)

```
RODOS RODOS-110.1 OS Version RODOS-Leon2-v1
Loaded Applications:
         10 -> 'Topics & Middleware'
       1005 -> 'AOCS'
       1000 -> 'Main'
       1002 -> 'ThermalControl'
       1001 -> 'TempInterface'
       1004 -> 'PowerControl'
       1003 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
 Temp5  Id = 17210 len = 4.   -- Subscribers:
     ThermalControl
 Temp4  Id = 13115 len = 4.   -- Subscribers:
     ThermalControl
 Temp3  Id = 9056 len = 4.   -- Subscribers:
     ThermalControl
 Temp2  Id = 28993 len = 4.   -- Subscribers:
     ThermalControl
 Temp1  Id = 24866 len = 4.   -- Subscribers:
     ThermalControl
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =     100 Stack =  10000 AOCSThread:
   Prio =     100 Stack =  10000 MainThread:
   Prio =     100 Stack =  10000 ThermalControlThread:
   Prio =     100 Stack =  10000 TempInterfaceThread:
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
BigEndianity = 1, cpu-Arc = leon3, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 208
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
POWERCONTROL: Calculated power (mW): 0
THERMALCONTROL: Temperature 1 (°C): 31
THERMALCONTROL: Temperature 2 (°C): 40
THERMALCONTROL: Temperature 3 (°C): 79
THERMALCONTROL: Temperature 4 (°C): 5
MAIN: Time since boot (sec): 1
THERMALCONTROL: Temperature 5 (°C): 56
                                                       :
                                                       :
```

ple rate of the scheduler-related threads when using the Monitoring F/W instead of printlining with PRINTF:

- `Thread::findNextToRun(long long)`:
  5.17% PRINTF ⇒ 20.06% Monitoring F/W (increase of ≈ 300%)

- `hwTrapSaveContextContinue`:
  1.72% PRINTF ⇒ 5.21% Monitoring F/W (increase of ≈ 215%)

- `__asmSwitchToContext`:
  1.06% PRINTF ⇒ 3.32% Monitoring F/W (increase of ≈ 225%)

- `Scheduler::schedule()`:
  0.58% PRINTF ⇒ 2.06% Monitoring F/W (increase of ≈ 265%)

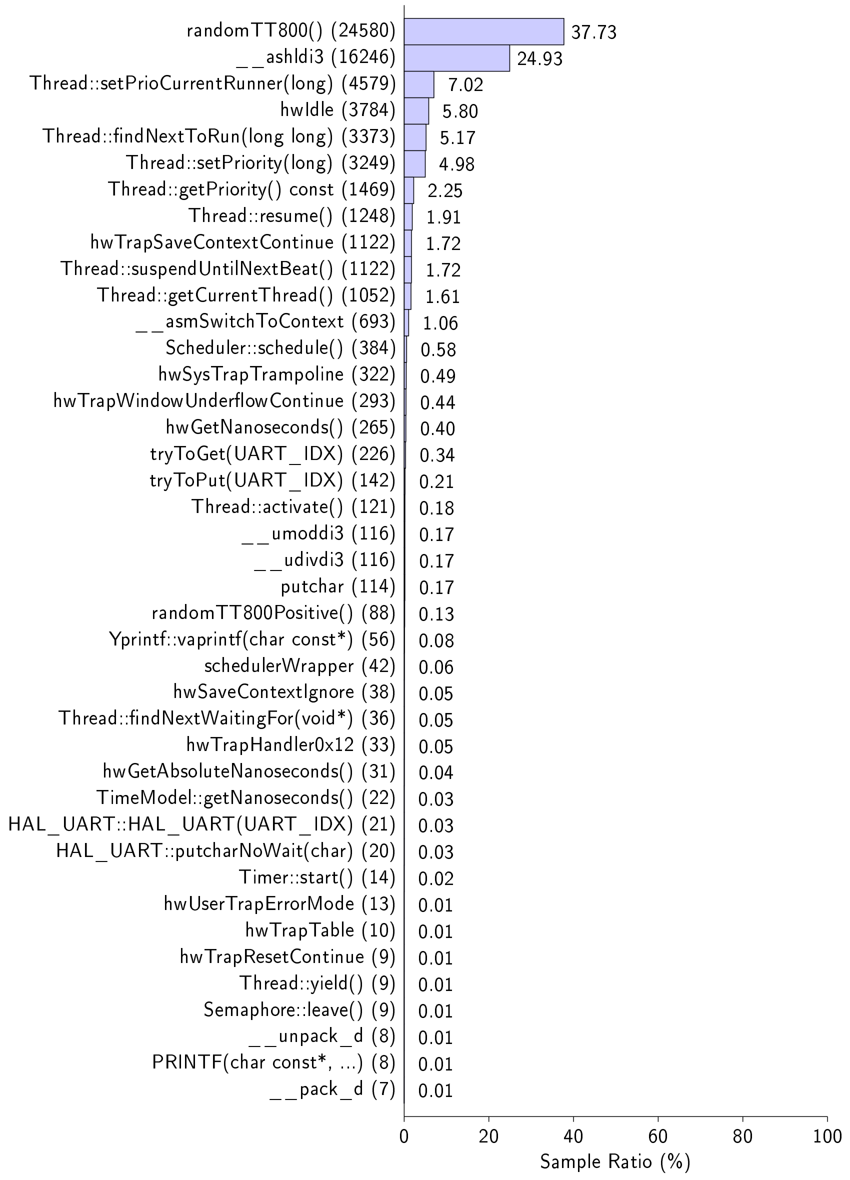


Figure 18.10: Sample Ratio of Use Case 2 on LEON3-based H/W with PRINTF (GRMON Profiler Results)

- `hwSysTrapTrampoline`:
  0.49% PRINTF ⇒ 1.73% Monitoring F/W (increase of ≈ 325%)

Load from Log Threads The increased usage of the Monitoring F/W – especially by the `ThermalControl` application which continuously logs all its information received from the temperature interface (see Figure 16.13) – does not result in an increased sampling rate of the corresponding methods: the `LOG::LogInputThread::run()`-method is sampled only 18 times, and even if the operator-method `LOG::LogInputter::operator«(void*)` does appear in these test results (contrary to the results from use case 1, cf. to Bar Chart 18.4) it is only sampled eight times.

Listing 18.10: Use Case 2: Output tst-sat with Monitoring F/W on LEON3-based H/W

```
RODOS RODOS-110.1 OS Version RODOS-Leon2-v1
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
       1005 -> 'AOCS'
       1000 -> 'Main'
       1002 -> 'ThermalControl'
       1001 -> 'TempInterface'
       1004 -> 'PowerControl'
       1003 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 64.   -- Subscribers:
 Temp5  Id = 17210 len = 4.   -- Subscribers:
     ThermalControl
 Temp4  Id = 13115 len = 4.   -- Subscribers:
     ThermalControl
 Temp3  Id = 9056 len = 4.   -- Subscribers:
     ThermalControl
 Temp2  Id = 28993 len = 4.   -- Subscribers:
     ThermalControl
 Temp1  Id = 24866 len = 4.   -- Subscribers:
     ThermalControl
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  10000 IdleThread: yields all the time
   Prio =      50 Stack =  10000 LogControlThread: Receives and realizes log controls
   Prio =    1000 Stack =  10000 LogInputThread: Collects and publishes log messages
   Prio =     100 Stack =  10000 AOCSThread:
   Prio =     100 Stack =  10000 MainThread:
   Prio =     100 Stack =  10000 ThermalControlThread:
   Prio =     100 Stack =  10000 TempInterfaceThread:
   Prio =     100 Stack =  10000 PowerControlThread:
   Prio =     100 Stack =  10000 BatteryInterfaceThread:
   Prio =    1002 Stack =  10000 gateway:
BigEndianity = 1, cpu-Arc = leon3, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 208
----------------------------------------------------
Default internal MAIN
--------------- application running ------------      .
                                                        .
                                                        .
```

MESSAGE TRANSPORT

Load from Middleware Threads The simulated increased load on the monitoring messages being sent out to the groundstation over the RODOS middleware does not result in conspicuously high sampling rates of the responsible methods: `LinkinterfaceUART::putcharEncoded(bool, char)` was sampled 494 times and `LinkinterfaceUART::sendNetworkMsg(NetworkMessage&)` and 144 times.

Like in Use Case 1 (cf. to Paragraph 18.2.1.4), there is an increase in the sampling rates of the UART-related methods which also appear in the PRINTF-based scenario on LEON3:

- `tryToGet(UART_IDX)`:
  0.34% PRINTF $\Rightarrow$ 1.34% Monitoring F/W (increase of $\approx$ 300%)

Listing 18.11: Use Case 2: Output tst-ground (getting Monitoring Message from LEON3-based H/W)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424299 -> 'logOutputApp'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
 log  Id = 42 len = 60.   -- Subscribers:
     LogOutputThread
 Temp5  Id = 17210 len = 4.   -- Subscribers:
 Temp4  Id = 13115 len = 4.   -- Subscribers:
 Temp3  Id = 9056 len = 4.   -- Subscribers:
 Temp2  Id = 28993 len = 4.   -- Subscribers:
 Temp1  Id = 24866 len = 4.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:

Event servers:
  1 TimeEvent managers
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 SimpleLogControllerThread: Provides simple console interface to publish log
         controls
   Prio =    1000 Stack =  32000 LogOutputThread: Receives and outputs log messages
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[DEBUG] Main.MainThread: Time since Boot (sec): 1
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°C): 28
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[INFO] ThermalControl.TempInterfaceThr: Temperature 2 (°C): 39
[INFO] ThermalControl.TempInterfaceThr: Temperature 3 (°C): 86
[INFO] ThermalControl.TempInterfaceThr: Temperature 4 (°C): 6
[INFO] ThermalControl.TempInterfaceThr: Temperature 5 (°C): 53
[WARN] ThermalControl.ThermalControlTh: Temperature 3 is critical
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°C): 32
                                                   ⋮
```

- `tryToPut(UART_IDX)`:
  0.21% PRINTF ⇒ 0.80% Monitoring F/W (increase of ≈ 280%)
- `HAL_UART::putcharNoWait(char)`:
  0.03% PRINTF ⇒ 0.30% Monitoring F/W (increase of ≈ 950%)

18.3.2 *RODOS-Linux-Port, x86-VM*

18.3.2.1 *Test Fixture*

The test fixture for the Linux-only tests can be found in Table 18.4. In comparison to the LEON-based tests this time the complete file sizes of the compiled ELF-files are given (for an explanation cf. to Section 18.5.2.2). For the tests with the Monitoring F/W prototype two file sizes are stated: one for the test over the simulated UART-port (connection between two VMs) and one for the UDP-based test inside

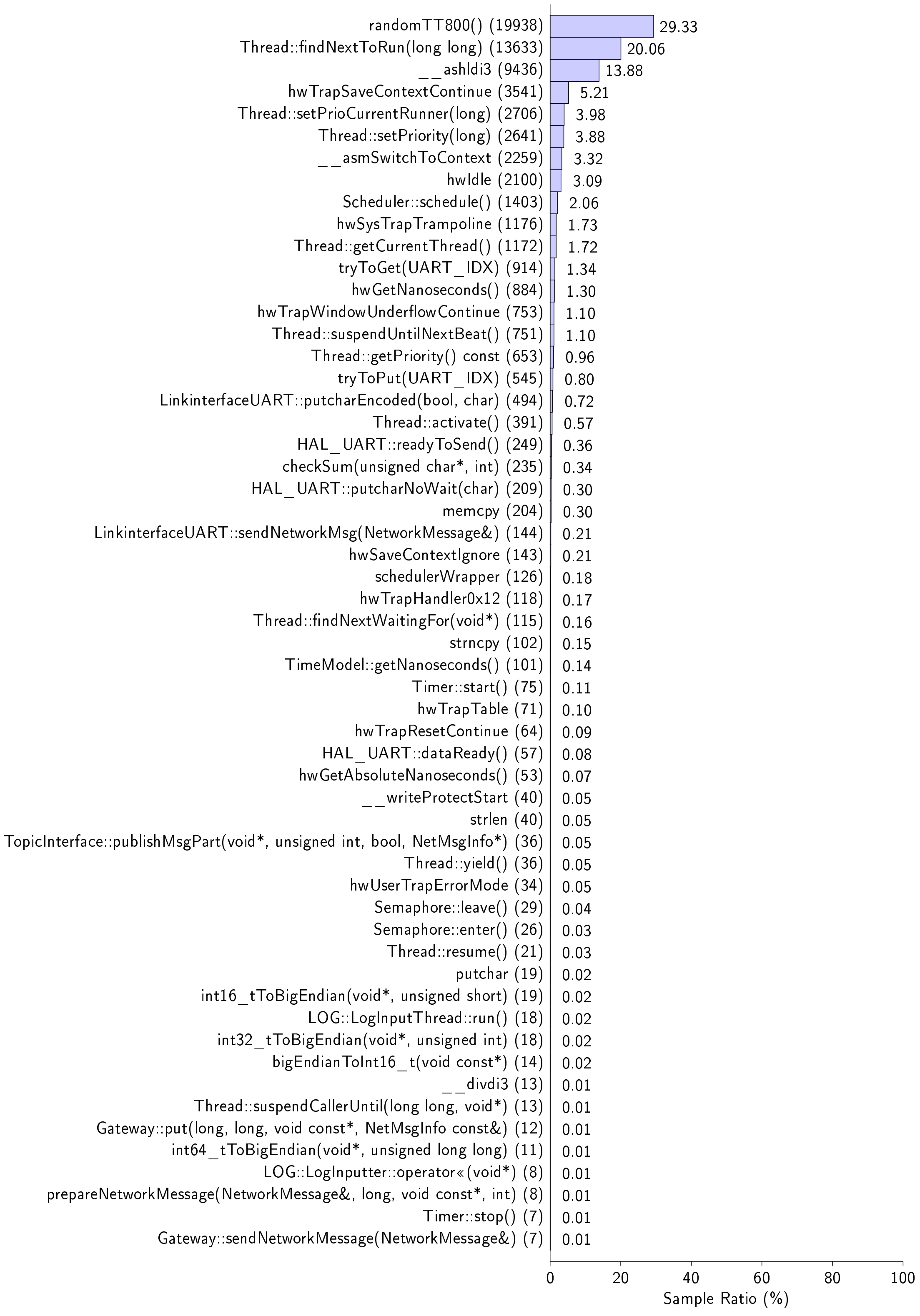


Figure 18.11: Sample Ratio of Use Case 2 on LEON3-based H/W with the Monitoring F/W prototype (GRMON Profiler Results)

one VM (for an overview of the different test setups of use case 2 cf. to Figure 18.9).

Test Name	Use Case 2
Test Date	09.1.2014
Test Duration	10 minutes
Computing Node 1 (Satellite- & Groundstation-Simulator)	
Hardware	Orcale VirtualBox Machine (Emulated Hardware): Intel Core2Duo E8500 @3.16 GHz; 8GB RAM
Software	Arch Linux x86 (Kernel 3.12.6-1-ARCH)
Test-Tool	
GCC-Version	4.8.2
Test-ELF File on Linux/X86-VM (tst-sat PRINTF):	
total size: 166178 bytes	
Test-ELF File on Linux/X86-VM (tst-sat Monitoring Framework):	
total size (over UART): 380547 bytes	
total size (internal-UDP): 399504 bytes	

Table 18.4: Test Fixture for the Tests on Linux/X86-based VM

18.3.2.2 *Test Results - PRINTF*

The `tst-sat`-printout in Listing 18.12 shows the RODOS startup sequence of the PRINTF-based version of use case 2.

After ten minutes of execution time the `tst-sat` image exits and the profiling results from GPROF can be analyzed. They consist of two parts: the flat profile and the call graph. The flat profile gives the execution time spent in each function as a percentage of the total running time. Function call counts are also reported. The output shown in Figure 18.12 is sorted by percentage, with hot spots at the top of the list.

The second part of the output is the textual call graph, which shows for each function who called it (parent) and who it called (child subroutines). The graphical form of the results can be seen in Picture 18.13. They are created by an external tool[82].

Both output results from GPROF had to be reduced (1) because of their length and (2) to concentrate on the most important aspects: the bar charts were reduced in terms of how many items are displayed,

82 *gprof2dot* was used here (see `http://code.google.com/p/jrfonseca/wiki/Gprof2Dot`). This tool is capable of converting the call graph from GPROF into a graphical form

Listing 18.12: Use Case 2: Output tst-sat w/o the Monitoring F/W (only PRINTF)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
          10 -> 'Topics & Middleware'
        1005 -> 'AOCS'
        1000 -> 'Main'
        1002 -> 'ThermalControl'
        1001 -> 'TempInterface'
        1004 -> 'PowerControl'
        1003 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
 Temp5  Id = 17210 len = 4.   -- Subscribers:
     ThermalControl
 Temp4  Id = 13115 len = 4.   -- Subscribers:
     ThermalControl
 Temp3  Id = 9056 len = 4.   -- Subscribers:
     ThermalControl
 Temp2  Id = 28993 len = 4.   -- Subscribers:
     ThermalControl
 Temp1  Id = 24866 len = 4.   -- Subscribers:
     ThermalControl
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =     100 Stack =  32000 AOCSThread:
   Prio =     100 Stack =  32000 MainThread:
   Prio =     100 Stack =  32000 ThermalControlThread:
   Prio =     100 Stack =  32000 TempInterfaceThread:
   Prio =     100 Stack =  32000 PowerControlThread:
   Prio =     100 Stack =  32000 BatteryInterfaceThread:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
-------------- application running ------------
POWERCONTROL: Calculated power (mW): 0
THERMALCONTROL: Temperature 1 (°C): 27
THERMALCONTROL: Temperature 2 (°C): 38
THERMALCONTROL: Temperature 3 (°C): 72
THERMALCONTROL: Temperature 4 (°C): 5
THERMALCONTROL: Temperature 5 (°C): 41
MAIN: Time since boot (sec): 1
POWERCONTROL: Calculated power (mW): 0
                                                        ⋮
```

and the call graphs were edited in terms of deletion of some of the displayed boxes[83].

18.3.2.3 *Test Results - Monitoring Framework*

Like it is shown in the overview picture on page 180, the tests of use case 2 with the Monitoring F/W prototype are divided into two scenarios:

TWO VIRTUAL MACHINES , communicating over a simulated UART-connection, whereas one machine is acting as the satellite (exe-

83 In addition, the appearance of some functions which are related to the *MetaLogger* part of the F/W's prototype (cf. to implementation Section 16.2, Footnote 48) were removed.

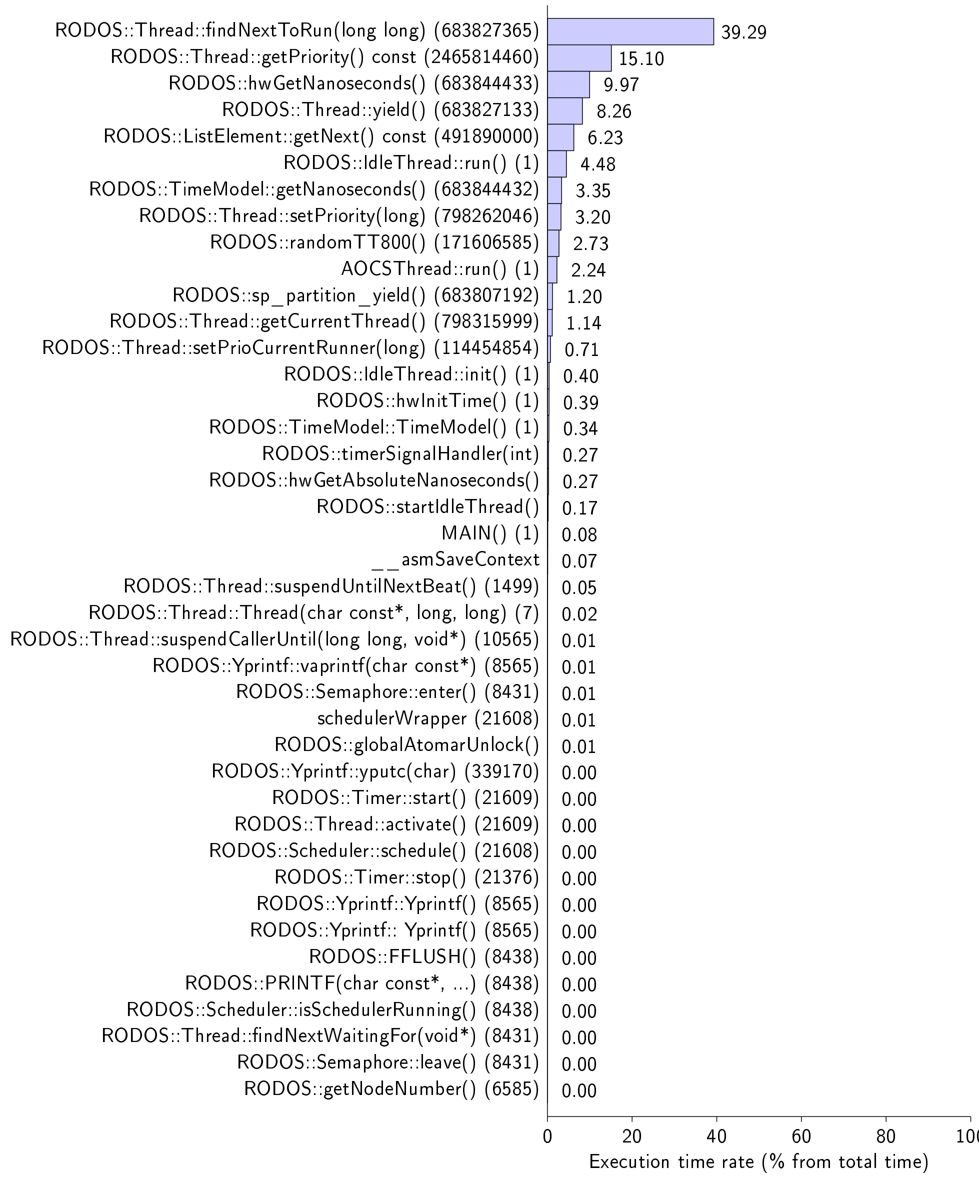


Figure 18.12: Use Case 2 on Linux/X86 with PRINTF (results from GPROF): Execution time spent in each function given as percentage of total running time. The function call counts are written in parenthesis.

cuting the `tst-sat` boot image) and the other one is acting as the groundstation (executing the `tst-ground` image).

ONE VIRTUAL MACHINE , letting the `tst-sat` and `tst-ground` images communicate over a UDP-connection.

TWO VIRTUAL MACHINES (OVER UART) The initial output sequences of the started `tst-sat` and `tst-ground` files are contained in the Listings 18.13 and 18.14.

Listing 18.13: Use Case 2: Output tst-sat with the Monitoring F/W (over UART)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424298 -> 'logApp'
       1005 -> 'AOCS'
       1000 -> 'Main'
       1002 -> 'ThermalControl'
       1001 -> 'TempInterface'
       1004 -> 'PowerControl'
       1003 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 60.   -- Subscribers:
 Temp5  Id = 17210 len = 4.   -- Subscribers:
     ThermalControl
 Temp4  Id = 13115 len = 4.   -- Subscribers:
     ThermalControl
 Temp3  Id = 9056 len = 4.   -- Subscribers:
     ThermalControl
 Temp2  Id = 28993 len = 4.   -- Subscribers:
     ThermalControl
 Temp1  Id = 24866 len = 4.   -- Subscribers:
     ThermalControl
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl

Event servers:
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 LogControlThread: Receives and realizes log controls
   Prio =    1000 Stack =  32000 LogInputThread: Collects and publishes log messages
   Prio =     100 Stack =  32000 AOCSThread:
   Prio =     100 Stack =  32000 MainThread:
   Prio =     100 Stack =  32000 ThermalControlThread:
   Prio =     100 Stack =  32000 TempInterfaceThread:
   Prio =     100 Stack =  32000 PowerControlThread:
   Prio =     100 Stack =  32000 BatteryInterfaceThread:
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
---------------------------------------------------
Default internal MAIN
--------------- application running ------------     ⋮
                                                     ⋮
```

After ten minutes the test run is stopped, the GPROF results in bar-chart format can be examined in Bar-Chart 18.14. The same results – displayed as a textual call graph – are also contained in Figure 18.15.

inside one virtual machine (over udp) The initial output sequences of the started `tst-sat` and `tst-ground` files are contained in the Listings 18.15 and 18.16.

After ten minutes the test run is stopped, the GPROF results in bar-chart format can be examined in Bar Chart 18.16. The same results – displayed as a textual call graph – are also contained in Figure 18.17.

Listing 18.14: Use Case 2: Output tst-sat with the Monitoring F/W (over UART)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424299 -> 'logOutputApp'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
 log  Id = 42 len = 60.   -- Subscribers:
     LogOutputThread
 Temp5  Id = 17210 len = 4.   -- Subscribers:
 Temp4  Id = 13115 len = 4.   -- Subscribers:
 Temp3  Id = 9056 len = 4.   -- Subscribers:
 Temp2  Id = 28993 len = 4.   -- Subscribers:
 Temp1  Id = 24866 len = 4.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:

Event servers:
  1 TimeEvent managers
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 SimpleLogControllerThread: Provides simple console interface to publish log
         controls
   Prio =    1000 Stack =  32000 LogOutputThread: Receives and outputs log messages
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
-------------- application running ------------
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°C): 27
[INFO] ThermalControl.TempInterfaceThr: Temperature 2 (°C): 38
[INFO] ThermalControl.TempInterfaceThr: Temperature 3 (°C): 72
[INFO] ThermalControl.TempInterfaceThr: Temperature 4 (°C): 5
[INFO] ThermalControl.TempInterfaceThr: Temperature 5 (°C): 41
[DEBUG] Main.MainThread: Time since Boot (sec): 1
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°C): 31
                                                      ⋮
```

18.3.2.4 *Test Analysis & Comparison*

BOOT IMAGE SIZE Like in the embedded H/W-based tests before, the values in Table 18.4 reveal that also in a pure virtual Linux-based environment there is a significant increase in the boot image size when using the Monitoring F/W prototype: both images (UART and UDP) have more than twice the size of the image which is using PRINTF for debug purposes.

PROCESSOR LOAD Within this paragraph we will compare the GPROF sampling results of use case 2 without the Monitoring F/W shown in Bar Chart 18.12 with the tests performed using the F/W with the simulated UART connection (Bar Chart 18.14) and with the UDP-based communication (Bar Chart 18.16).

Using Bar Charts for Comparison

The reason for choosing the bar charts for comparison instead of the corresponding textual call graphs presented in Figure 18.15 and Figure 18.17 is the following: The call graphs help to understand the

Listing 18.15: Use Case 2: Output tst-sat with the Monitoring F/W (over UDP)

```
ODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424298 -> 'logApp'
        1005 -> 'AOCS'
        1000 -> 'Main'
        1002 -> 'ThermalControl'
        1001 -> 'TempInterface'
        1004 -> 'PowerControl'
        1003 -> 'BatteryInterface'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
     LogControlThread
 log  Id = 42 len = 60.   -- Subscribers:
 Temp5  Id = 17210 len = 4.   -- Subscribers:
     ThermalControl
 Temp4  Id = 13115 len = 4.   -- Subscribers:
     ThermalControl
 Temp3  Id = 9056 len = 4.   -- Subscribers:
     ThermalControl
 Temp2  Id = 28993 len = 4.   -- Subscribers:
     ThermalControl
 Temp1  Id = 24866 len = 4.   -- Subscribers:
     ThermalControl
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
     PowerControl
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
     PowerControl
 udp async topic  Id = 22582 len = 12.   -- Subscribers:
     anonymThreadSubscriber

Event servers:
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 LogControlThread: Receives and realizes log controls
   Prio =    1000 Stack =  32000 LogInputThread: Collects and publishes log messages
   Prio =     100 Stack =  32000 AOCSThread:
   Prio =     100 Stack =  32000 MainThread:
   Prio =     100 Stack =  32000 ThermalControlThread:
   Prio =     100 Stack =  32000 TempInterfaceThread:
   Prio =     100 Stack =  32000 PowerControlThread:
   Prio =     100 Stack =  32000 BatteryInterfaceThread:
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
----------------------------------------------------
Default internal MAIN
--------------- application running ------------      ⋮
                                                      ⋮
```

relationships between the sampled methods, but it is somehow misleading by how the results are presented: because the time spent in each function is summed up in every parent node, the boxes marked e.g. in the color red do not contain the methods which consume most of the processor time. As an example, the topmost box in both figures is showing the summed up execution time for the `threadStartupWrapper`-method. A quick look into the source in the code snippet in Listing 18.17 reveals, that this method does nothing than yielding all the time in an endless loop (like e.g. the Idle-thread, mentioned in Paragraph 18.2.1.4) The execution time spent in this function is so little that it does not even appear in the corresponding bar charts.

Listing 18.16: Use Case 2: Output tst-sat with the Monitoring F/W (over UDP)

```
RODOS RODOS-110.1 OS Version RODOS-linux-8
Loaded Applications:
         10 -> 'Topics & Middleware'
         20 -> 'Gateway'
      424299 -> 'logOutputApp'
Calling Initiators
Distribute Subscribers to Topics
List of Middleware Topics:
 CharInput  Id = 28449 len = 12.   -- Subscribers:
 SigTermInterrupt  Id = 16716 len = 4.   -- Subscribers:
 UartInterrupt  Id = 15678 len = 4.   -- Subscribers:
 TimerInterrupt  Id = 25697 len = 4.   -- Subscribers:
 routerTopic  Id = 21324 len = 1326.   -- Subscribers:
 gatewayTopic  Id = 0 len = 12.   -- Subscribers:
     Gateway
 logControl  Id = 26706 len = 8.   -- Subscribers:
 log  Id = 42 len = 60.   -- Subscribers:
     LogOutputThread
 Temp5  Id = 17210 len = 4.   -- Subscribers:
 Temp4  Id = 13115 len = 4.   -- Subscribers:
 Temp3  Id = 9056 len = 4.   -- Subscribers:
 Temp2  Id = 28993 len = 4.   -- Subscribers:
 Temp1  Id = 24866 len = 4.   -- Subscribers:
 BatteryVoltage  Id = 17758 len = 4.   -- Subscribers:
 BatteryCurrent  Id = 32073 len = 4.   -- Subscribers:
 udp async topic  Id = 22582 len = 12.   -- Subscribers:
     anonymThreadSubscriber

Event servers:
  1 TimeEvent managers
Threads in System:
   Prio =       0 Stack =  32000 IdleThread: yields all the time
   Prio =      50 Stack =  32000 SimpleLogControllerThread: Provides simple console interface to publish log
         controls
   Prio =    1000 Stack =  32000 LogOutputThread: Receives and outputs log messages
   Prio =    1002 Stack =  32000 gateway:
BigEndianity = 0, cpu-Arc = x86, Basis-Os = baremetal, Cpu-Speed (K-Loops/sec) = 350000
-----------------------------------------------------
Default internal MAIN
--------------- application running ------------
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°CC): 27
[INFO] ThermalControl.TempInterfaceThr: Temperature 2 (°CC): 38
[INFO] ThermalControl.TempInterfaceThr: Temperature 3 (°CC): 72
[INFO] ThermalControl.TempInterfaceThr: Temperature 4 (°CC): 5
[INFO] ThermalControl.TempInterfaceThr: Temperature 5 (°CC): 41
[DEBUG] Main.MainThread: Time since Boot (sec): 1
[INFO] PowerControl.PowerControlThre: Calculated power (mW): 0
[INFO] ThermalControl.TempInterfaceThr: Temperature 1 (°CC): 31
                                                       ⋮
```

Listing 18.17: RODOS method `threadStartupWrapper`

```
void threadStartupWrapper(Thread* thread) {
    Thread::currentThread = thread;
    thread->suspendedUntil = 0;
    thread->run();
    while(1) {
        thread->suspendedUntil = END_OF_TIME;
        thread->yield();
    }
}
```

What can be seen at a first glance within the bar charts is, that the number of function calls and the execution time spent in each function are not necessary correlated: taken the `IdleThreads run()` method as an example, it is only called once, but is using 4.48% of the total execution time. Furthermore, the GPROF results under Linux for UDP and UART are more or less the same. Therefore – for the following comparisons – we will use the mean (averaged) values from both tests in order to compare them to the PRINTF-based scenario of use case 2. The remaining variations/fluctuations between UART and UDP seem most likely to result from interference of the underlying Linux scheduler with the RODOS-internal scheduler. Additional test using only the UART-based scenario have validated this observation.

Averaging the Results

Load from Scheduler The additional Monitoring F/W threads invoked by the scheduler are causing an increase in the sample rates of scheduler-related functions – in the UDP- as well as in the UART-based test scenario. The increase is moderate for the top-most scheduler methods in which most of the execution time is spent, and gets higher for the methods which are not that dominant:

- `RODOS::Thread::findNextToRun(long long)`:
  39.29% PRINTF ⇒ 43.9% Monitoring F/W (increase of ≈ 11%)
- `RODOS::Thread::getPriority()`:
  15.10% PRINTF ⇒ 17.18% Monitoring F/W (increase of ≈ 13%)
- `RODOS::Thread::setPriority()`:
  3.20% PRINTF ⇒ 11.00% Monitoring F/W (increase of ≈ 240%)
  item `__asmSaveToContext`:
  0.07% PRINTF ⇒ 0.23% Monitoring F/W (increase of ≈ 220%)

Load from Log Threads The influence of the methods of the methods contained in the Monitoring F/W source code is negligible. The methods – residing in the C++ namespace *Log* and therefore can be identified by the preceding `LOG::`-text within the bar charts – are mostly in the range of 0.01% execution time rate. The highest execution time rate is caused in the UDP-scenario by the `LOG::LogInputThread::run()`-method which rate is indicated in Bar Chart 18.16 with 0.03%.

MESSAGE TRANSPORT

Load from Middleware Threads In both scenarios there is no appreciable influence of the methods involved in the transport of the monitoring messages over the network: in the UART-based scenario the highest rated method `LinkinterfaceUART::putcharEncoded(bool,char)` is only indicated with 0.02% in Bar Chart 18.14, and in the in UDP-based scenario the `RODOS::NetworkMessage::put_len()`-method has the value 0.01%, as stated in Bar Chart 18.16.

18.3.3 *Overall Test Analysis & Comparison*

As already explained at the beginning of Section 18.2.3, in this section we will concentrate on the *relative* comparison of the results of the two tests scenarios of use case 2. For this purpose we refer to the two corresponding analysis Sections 18.3.1.4 and 18.3.2.4.

Nevertheless, there are some fundamental differences in the results of the two test scenarios which are so evident that they require a closer examination:

LOAD FROM RANDOMTT800-METHOD
: On the LEON3-based H/W the RODOS random number generator method `randomTT800` was dominating the sampling results with and without the Monitoring F/W prototype which changes in the test scenario running in pure Linux environment. The main reason for this behavior is the fact, that use case 2 was designed and implemented in a way that it runs without problems on all platforms given in Figure 18.9 without changes in the source code (among others, also for comparison reasons). As the H/W the Linux system runs upon (the VM as well as the underlying real H/W) is much more performant than the LEON3 processor on the Nexys3 board, there is a significant drop in the processor consumption by this computing-intense method.

LOAD FROM PRINTF-METHOD
: Unexpectedly, in the two test scenarios in which Monitoring F/W prototype is used the RODOS PRINTF-method has still a high proportion on the execution time, as it can be observed in the corresponding Bar Charts 18.10 and 18.11. That there is always a small number of PRINTF-methods being used was already explained in at the beginning of Section 18.2.1.3 and in the footnote on Page 164. But the reason for this increase in execution time lies not within the startup messages printed out by the boot images, but within the errors resulting from problems with the transport of the monitoring messages between the embedded system and the host (development) system:

Monitoring F/W Buffer vs. UART Transmission Speed

During both tests – after some time – missing characters, line breaks and completely lost messages could be observed on the receiver side. The cause lies in the insufficient transmission rate from one computing node to another: the embedded side is not able to send out the monitoring messages faster than they were produced and collected by the Monitoring F/W. Therefore, the buffer responsible for storing these messages before transmission overflows (cf. to the description of `LogInputBuffer` in

Section 16.2)[84]. These transmission errors are printed out by RODOS using its ERROR()-method: every time an error occurs, this method is invoked and does nothing more than increasing the corresponding error counter and printing out the error messages, using – again – PRINTF-statements.
In addition to the lost data and the increase in the sampling rates of PRINTF/ERROR-statements on the embedded side, there are also further problems caused by the buffer-handling at the output side on the target system which are described in Section 18.5.6.2 of the Final Test Analysis & Comparison-Section.

boot image size In all test scenarios of use case 2 depicted in Figure 18.9 we can observe an increase of the boot image size. Making a relative comparison, the increase in the pure Linux scenario is a little bit higher than in the case where we used the LEON3-based H/W for testing purposes. Taking a rough mean value, we find an average increase of about 40% percent when using the Monitoring F/W prototype for this use case.

processor load

Load from Scheduler Comparing the increase rate of the processor load resulting from the scheduler-related threads leads to different results on the two platforms: For the lower rated methods the increase rate is almost in the same order of magnitude ($\approx$ 250% averaged). This changes for the top-most rated method, e.g. for `Thread::findNextToRun()` the increase rate goes down from $\approx$ 300% on the LEON3-H/W to $\approx$ 11% on the Linux-based test environment.

Load from Log Threads In both H/W scenarios of use case 2 the load from the methods, which are called from threads related to the Monitoring F/W prototype are negligible: the highest value can be observed in Bar Chart 18.16 in which the execution time rate for this method is given with 0.03%.

message transport

Load from Middleware Threads In both test environments the absolute values for all functions involved in sending messages of the Monitoring F/W prototype from one computing node to another via UART or UDP are very low. Due to a lack of common methods for comparison it is not possible to make a statement about a relative

84 An additional test revealed that sending out 14 monitoring messages per second during a 10 minute test run is causing no problems. But the more monitoring messages were sent to the input buffer the more transmission errors occur. In the worst case – if the buffer overflows very fast – the transmission of monitoring messages stops completely.

increase or decrease between these two test environments (cf. also to Section 18.5.2.3).

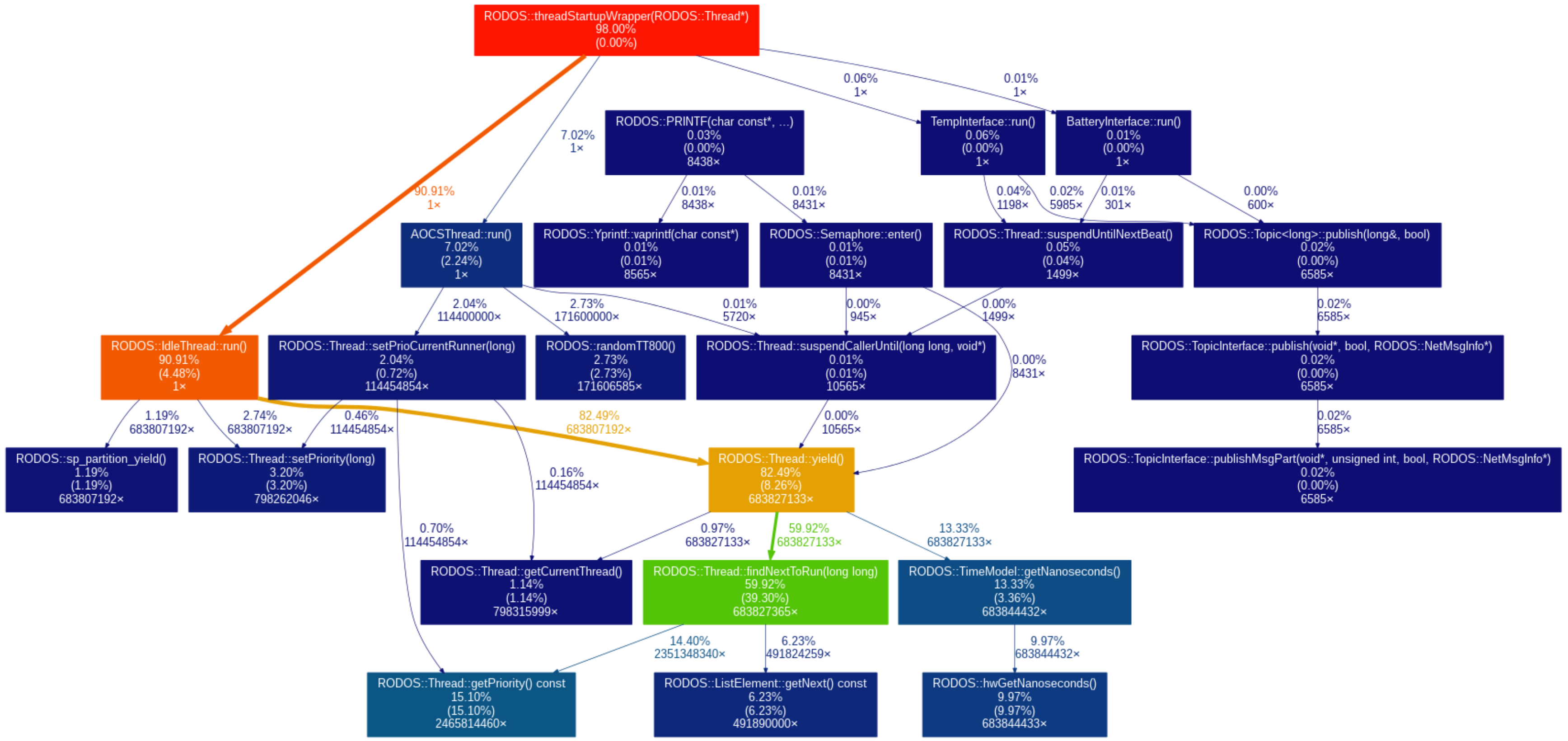

RODOS::threadStartupWrapper(RODOS::Thread*) 98.00% (0.00%)
90.91% 1×
7.02% 1×
0.06% 1×
0.01% 1×
RODOS::PRINTF(char const*, ...) 0.03% (0.00%) 8438×
TempInterface::run() 0.06% (0.00%) 1×
BatteryInterface::run() 0.01% (0.00%) 1×
0.01% 8438×
0.01% 8431×
0.04% 1198×
0.02% 5985×
0.01% 301×
0.00% 600×
AOCSThread::run() 7.02% (2.24%) 1×
RODOS::Yprintf::vaprintf(char const*) 0.01% (0.01%) 8565×
RODOS::Semaphore::enter() 0.01% (0.01%) 8431×
RODOS::Thread::suspendUntilNextBeat() 0.05% (0.04%) 1499×
RODOS::Topic<long>::publish(long&, bool) 0.02% (0.00%) 6585×
2.04% 114400000×
2.73% 171600000×
0.01% 5720×
0.00% 945×
0.00% 1499×
0.02% 6585×
RODOS::IdleThread::run() 90.91% (4.48%) 1×
RODOS::Thread::setPrioCurrentRunner(long) 2.04% (0.72%) 114454854×
RODOS::randomTT800() 2.73% (2.73%) 171606585×
RODOS::Thread::suspendCallerUntil(long long, void*) 0.01% (0.01%) 10565×
0.00% 8431×
RODOS::TopicInterface::publish(void*, bool, RODOS::NetMsgInfo*) 0.02% (0.00%) 6585×
1.19% 683807192×
2.74% 683807192×
0.46% 114454854×
82.49% 683807192×
0.00% 10565×
0.02% 6585×
RODOS::sp_partition_yield() 1.19% (1.19%) 683807192×
RODOS::Thread::setPriority(long) 3.20% (3.20%) 798262046×
0.16% 114454854×
RODOS::Thread::yield() 82.49% (8.26%) 683827133×
RODOS::TopicInterface::publishMsgPart(void*, unsigned int, bool, RODOS::NetMsgInfo*) 0.02% (0.00%) 6585×
0.70% 114454854×
0.97% 683827133×
59.92% 683827133×
13.33% 683827133×
RODOS::Thread::getCurrentThread() 1.14% (1.14%) 798315999×
RODOS::Thread::findNextToRun(long long) 59.92% (39.30%) 683827365×
RODOS::TimeModel::getNanoseconds() 13.33% (3.36%) 683844432×
14.40% 2351348340×
6.23% 491824259×
9.97% 683844432×
RODOS::Thread::getPriority() const 15.10% (15.10%) 2465814460×
RODOS::ListElement::getNext() const 6.23% (6.23%) 491890000×
RODOS::hwGetNanoseconds() 9.97% (9.97%) 683844433×

Figure 18.13: Use Case 2 on Linux/X86 with PRINTF: Textual call graph of the results from GPROF-profiling.

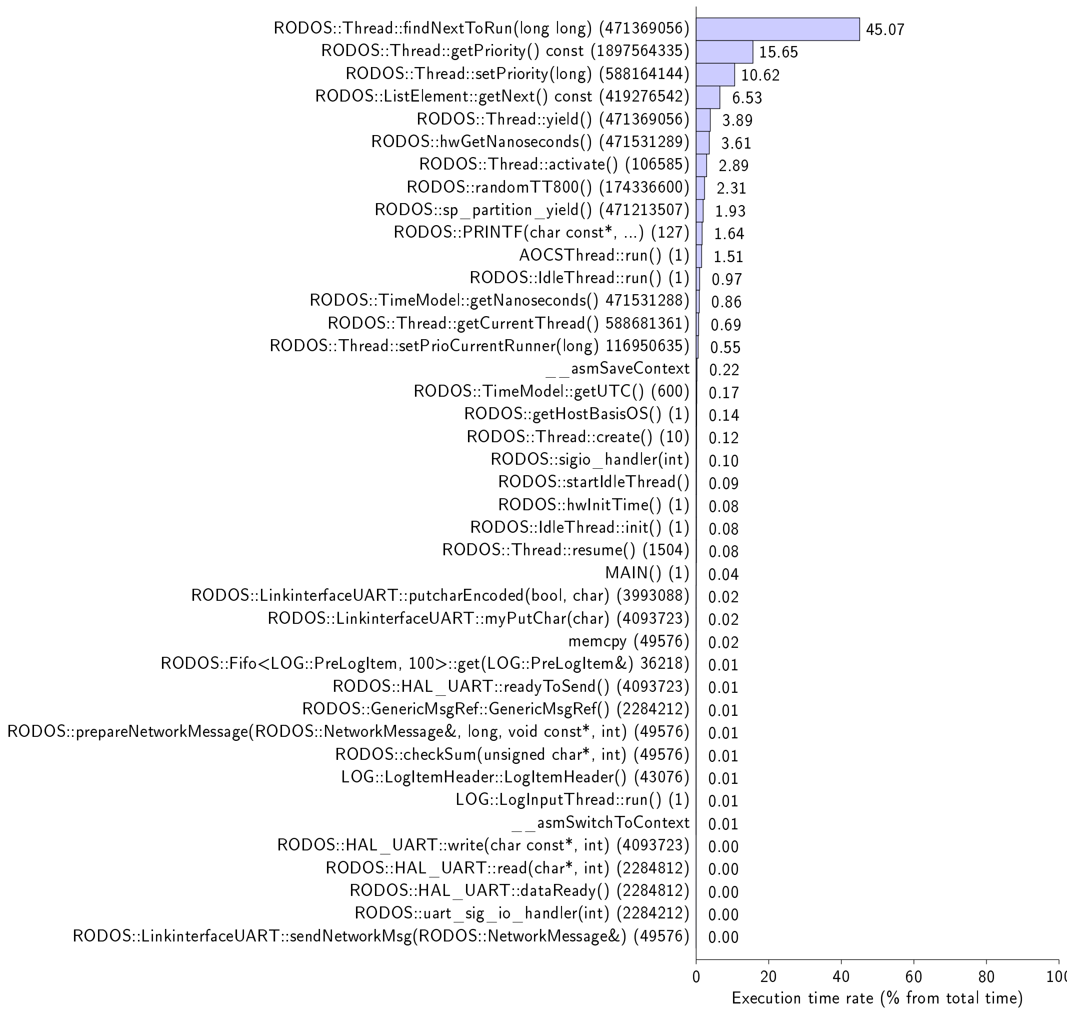


Figure 18.14: Use Case 2 with the Monitoring F/W prototype on Linux/X86 (over UART) (results from GPROF): Execution time spent in each function given as percentage of total running time. The function call counts are written in parenthesis.

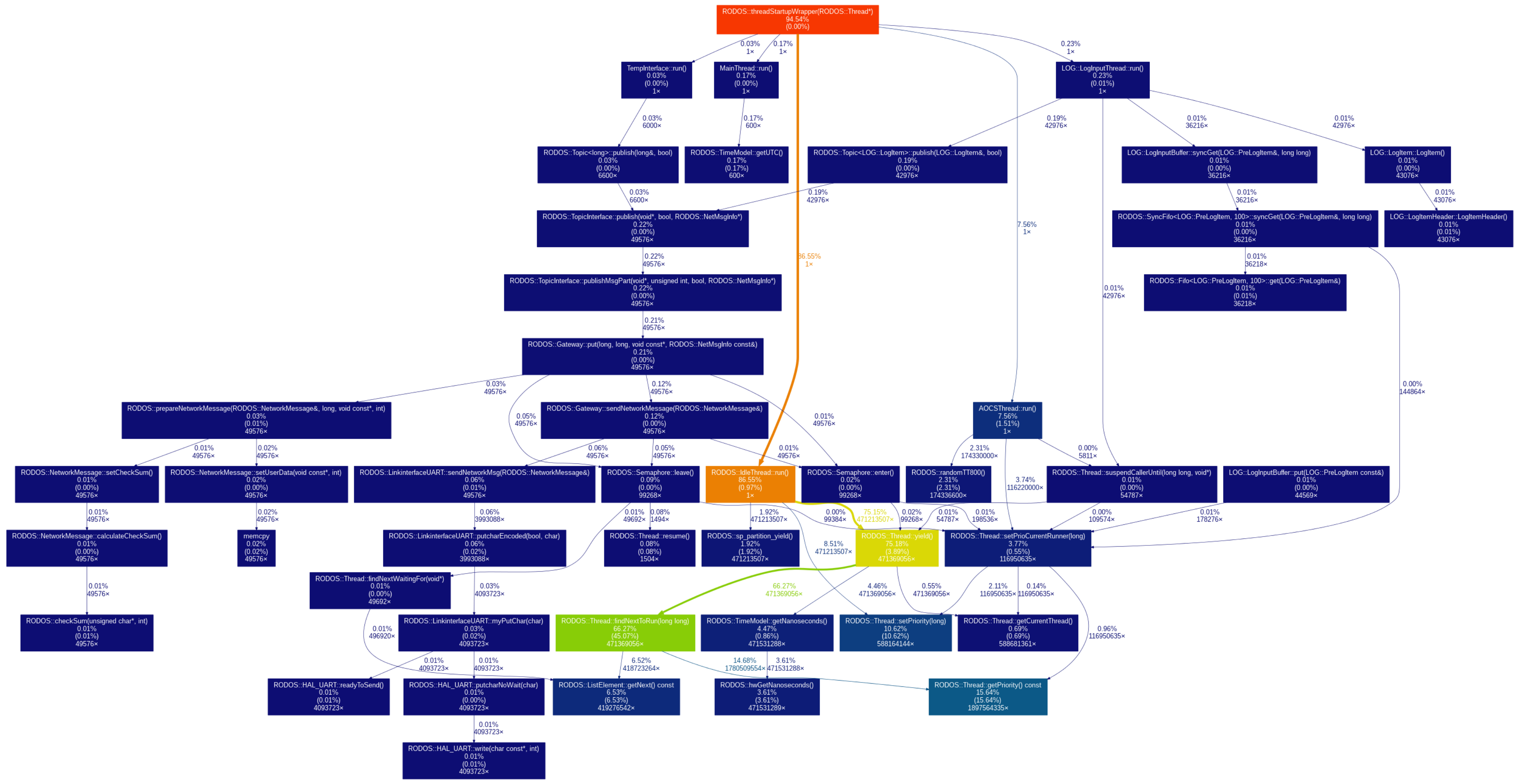

RODOS::threadStartupWrapper(RODOS::Thread*) 94.54% (0.00%)
TempInterface::run() 0.03% (0.00%) 1×
MainThread::run() 0.17% (0.00%) 1×
LOG::LogInputThread::run() 0.23% (0.01%) 1×
RODOS::Topic<long>::publish(long&, bool) 0.03% (0.00%) 6600×
RODOS::TimeModel::getUTC() 0.17% (0.17%) 600×
RODOS::Topic<LOG::LogItem>::publish(LOG::LogItem&, bool) 0.19% (0.00%) 42976×
LOG::LogInputBuffer::syncGet(LOG::PreLogItem&, long long) 0.01% (0.00%) 36216×
LOG::LogItem::LogItem() 0.01% (0.00%) 43076×
RODOS::TopicInterface::publish(void*, bool, RODOS::NetMsgInfo*) 0.22% (0.00%) 49576×
RODOS::SyncFifo<LOG::PreLogItem, 100>::syncGet(LOG::PreLogItem&, long long) 0.01% (0.00%) 36216×
LOG::LogItemHeader::LogItemHeader() 0.01% (0.01%) 43076×
RODOS::TopicInterface::publishMsgPart(void*, unsigned int, bool, RODOS::NetMsgInfo*) 0.22% (0.00%) 49576×
RODOS::Fifo<LOG::PreLogItem, 100>::get(LOG::PreLogItem&) 0.01% (0.01%) 36218×
RODOS::Gateway::put(long, long, void const*, RODOS::NetMsgInfo const&) 0.21% (0.00%) 49576×
RODOS::prepareNetworkMessage(RODOS::NetworkMessage&, long, void const*, int) 0.03% (0.01%) 49576×
RODOS::Gateway::sendNetworkMessage(RODOS::NetworkMessage&) 0.12% (0.00%) 49576×
AOCSThread::run() 7.56% (1.51%) 1×
RODOS::NetworkMessage::setCheckSum() 0.01% (0.00%) 49576×
RODOS::NetworkMessage::setUserData(void const*, int) 0.02% (0.00%) 49576×
RODOS::LinkinterfaceUART::sendNetworkMsg(RODOS::NetworkMessage&) 0.06% (0.01%) 49576×
RODOS::Semaphore::leave() 0.09% (0.00%) 99268×
RODOS::IdleThread::run() 86.55% (0.97%) 1×
RODOS::Semaphore::enter() 0.02% (0.00%) 99268×
RODOS::randomTT800() 2.31% (2.31%) 174336600×
RODOS::Thread::suspendCallerUntil(long long, void*) 0.01% (0.00%) 54787×
LOG::LogInputBuffer::put(LOG::PreLogItem const&) 0.01% (0.00%) 44569×
RODOS::NetworkMessage::calculateCheckSum() 0.01% (0.00%) 49576×
memcpy 0.02% (0.02%) 49576×
RODOS::LinkinterfaceUART::putcharEncoded(bool, char) 0.06% (0.02%) 3993088×
RODOS::Thread::resume() 0.08% (0.08%) 1504×
RODOS::sp_partition_yield() 1.92% (1.92%) 471213507×
RODOS::Thread::yield() 75.18% (3.89%) 471369056×
RODOS::Thread::setPrioCurrentRunner(long) 3.77% (0.55%) 116950635×
RODOS::Thread::findNextWaitingFor(void*) 0.01% (0.00%) 49692×
RODOS::checkSum(unsigned char*, int) 0.01% (0.01%) 49576×
RODOS::LinkinterfaceUART::myPutChar(char) 0.03% (0.02%) 4093723×
RODOS::Thread::findNextToRun(long long) 66.27% (45.07%) 471369056×
RODOS::TimeModel::getNanoseconds() 4.47% (0.86%) 471531288×
RODOS::Thread::setPriority(long) 10.62% (10.62%) 588164144×
RODOS::Thread::getCurrentThread() 0.69% (0.69%) 588681361×
RODOS::HAL_UART::readyToSend() 0.01% (0.01%) 4093723×
RODOS::HAL_UART::putcharNoWait(char) 0.01% (0.00%) 4093723×
RODOS::ListElement::getNext() const 6.53% (6.53%) 419276542×
RODOS::hwGetNanoseconds() 3.61% (3.61%) 471531289×
RODOS::Thread::getPriority() const 15.64% (15.64%) 1897564335×
RODOS::HAL_UART::write(char const*, int) 0.01% (0.01%) 4093723×

Figure 18.15: Use Case 2 with Monitoring F/W (over UART): textual call graph of results from GPROF-profiling.

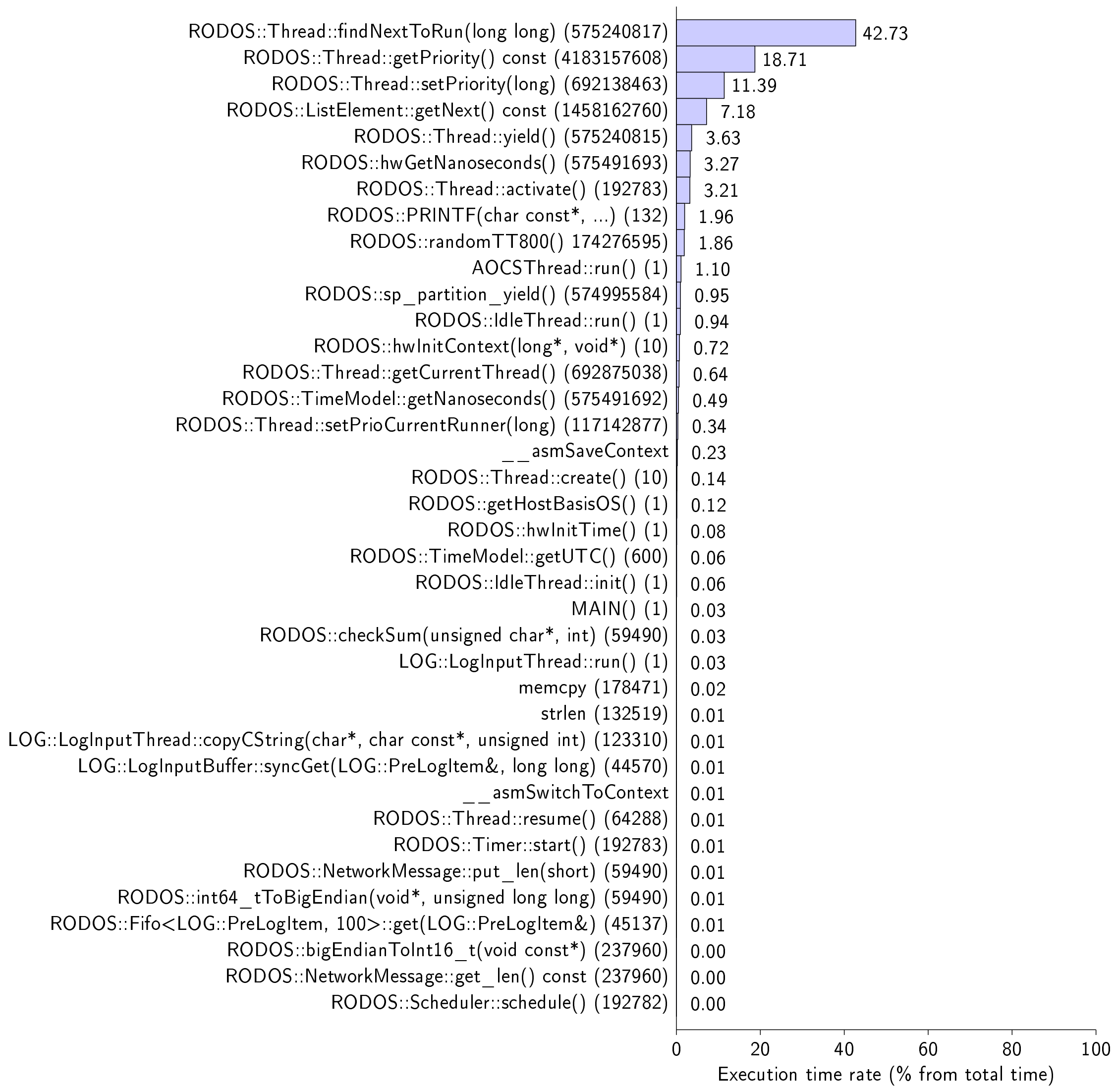


Figure 18.16: Use Case 2 with the Monitoring F/W on Linux/X86 (over UDP) (results from GPROF): Execution time spent in each function given as percentage of total running time. The function call counts are written in parenthesis.

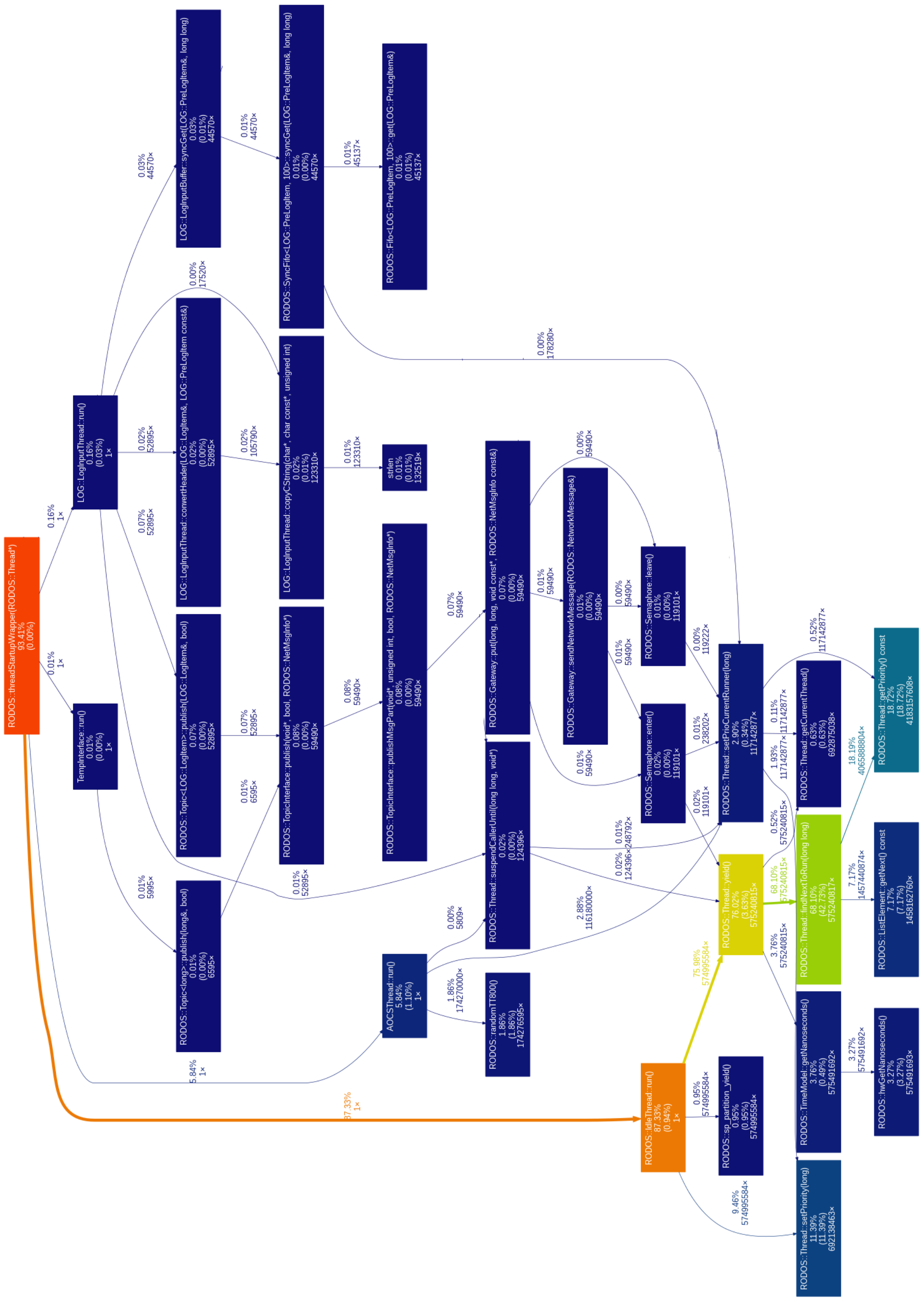

RODOS::threadStartupWrapper(RODOS::Thread*)
LOG::LogInputThread::run()
TempInterface::run()
LOG::LogInputBuffer::syncGet(LOG::PreLogItem&, long long)
LOG::LogInputThread::convertHeader(LOG::LogItem&, LOG::PreLogItem const&)
RODOS::Topic<LOG::LogItem>::publish(LOG::LogItem&, bool)
RODOS::Topic<long>::publish(long&, bool)
RODOS::SyncFifo<LOG::PreLogItem, 100>::syncGet(LOG::PreLogItem&, long long)
LOG::LogInputThread::copyCString(char*, char const*, unsigned int)
RODOS::TopicInterface::publish(void*, bool, RODOS::NetMsgInfo*)
RODOS::Fifo<LOG::PreLogItem, 100>::get(LOG::PreLogItem&)
strlen
RODOS::TopicInterface::publishMsgPart(void*, unsigned int, bool, RODOS::NetMsgInfo*)
AOCSThread::run()
RODOS::Gateway::put(long, long, void const*, RODOS::NetMsgInfo const&)
RODOS::Thread::suspendCallerUntil(long long, void*)
RODOS::randomTT800()
RODOS::Gateway::sendNetworkMessage(RODOS::NetworkMessage&)
RODOS::Semaphore::leave()
RODOS::Semaphore::enter()
RODOS::IdleThread::run()
RODOS::sp_partition_yield()
RODOS::Thread::setPrioCurrentRunner(long)
RODOS::Thread::yield()
RODOS::Thread::getCurrentThread()
RODOS::Thread::findNextToRun(long long)
RODOS::TimeModel::getNanoseconds()
RODOS::Thread::setPriority(long)
RODOS::Thread::getPriority() const
RODOS::ListElement::getNext() const
RODOS::hwGetNanoseconds()

Figure 18.17: Use Case 2 with Monitoring F/W (over UDP): Textual call graph of results from GPROF-profiling.

18.4 TEST RESULTS FROM USE CASE 3

Within the tests of use cases 1 and 2 a lot of emphasis has been put on testing the Monitoring F/W prototype in terms of static, dynamic and communication changes after integration into the final boot image (see Paragraph Structure of Tests on Page 156). Therefore, the tests regarding the real-world usage scenario of the Monitoring F/W prototype – the DLR School_Lab experiment FloatingSat – will focus on two aspects which are so far missing: (1) the additional bandwidth utilization by sending monitoring messages over the RODOS middleware, and (2) the usability experiences of the F/W by the developers of the FloatingSat experiment.

18.4.1 *Test Fixture*

The test setup is already depicted in Figure 16.17. In order to measure the bandwidth utilization by the RODOS middleware messages the corresponding RODOS UART gateway used for the Bluetooth-based communication was modified by adding a byte counter. After startup of the FloatingSat boot image the experiment has been run for five minutes without intervention (e.g. without sending TCs). During this time the summed-up byte-counts were printed out to a Linux terminal using a second (cable-based) UART connection to the FloatingSat. During this time the following data was transmitted from the FloatingSat to the groundstation[85]:

NORMAL TM SENT EVERY 200MS:

- Struct `TM_orientation` (velocity & angle values)
- Struct `TM_light` (values of the two light sensors)
- Struct `TM_power` (values of voltages and currents of the battery and the solar panels)
- Struct `TM_mode` (mode values)

THREADS SENDING LOG MESSAGES:

- `Alive`-thread: `DEBUG`-message sent every 10 seconds
- `Camera`-thread: `INFO`-message sent every 5 seconds
- `Mode`-thread: `INFO`-message sent every second
- `IMU`-thread: `DEBUG`-message sent every 5 seconds
- `Control`-thread: `INFO`-message every second
- `ADC`-thread: `DEBUG`-message sent every 5 seconds
- `Light`-thread: `DEBUG`-message sent every second

85 The term *groundstation* is used here as a synonym for the desktop computer running the FloatingSat GUI (see Figure 16.17

As it was already explained in Section 16.4.3.1 it is left to the programmer at which places standard TM messages are used (necessary for operating the FloatingSat; to be displayed in the main GUI seen in Figure 16.4.3.1), and where he integrates the log statements in his code (necessary for debugging purposes; to be displayed by the corresponding appenders seen in Figures 16.19a and 16.19b).

18.4.2 *Test Results*

The measurements of the bandwidth utilization using the test setup and fixture described above were carried out at three different log levels: OFF, INFO and DEBUG. The results can be seen in Figure 18.18.

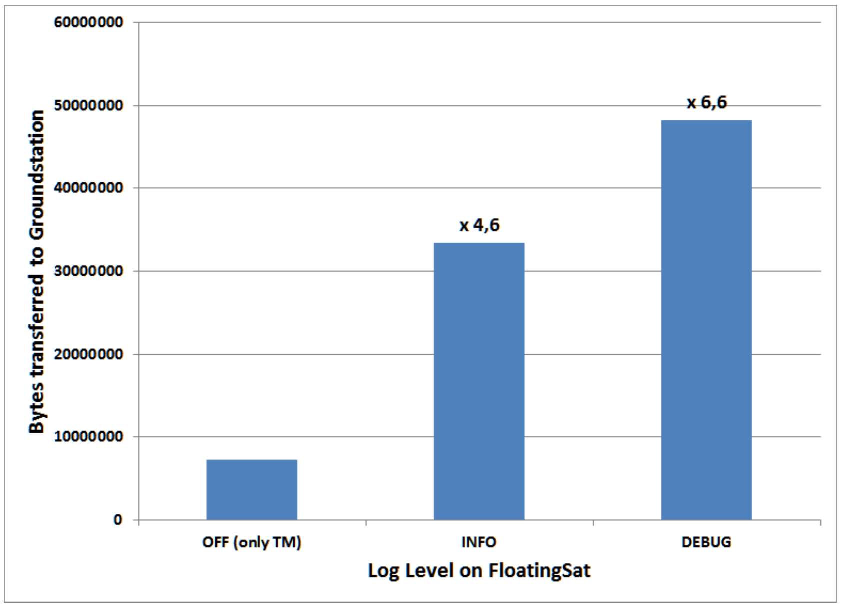


Figure 18.18: FloatingSat Bandwidth Utilization at different Log Levels

18.4.3 *Test Analysis*

Within this use case the proportion of middleware messages containing monitoring information in the form of monitoring items is relatively small in comparison to the number of the rest of middleware messages which contain normal TM. Nevertheless, the additional bandwidth utilization after switching the log level from OFF to INFO (4,6 times higher) and DEBUG (6,6 times higher) is significant, as it can been in Figure 18.18.

18.4.4 *Usability*

This use case differs from the first two not only in its technical setup (see Section 16.4.3.1) and its intention to serve as a real-world scenario: it is also the first use case in which the S/W developer of both – the embedded FloatingSat S/W and the non-embedded counterparts on the School_Lab's control computer – can be understood as a pure user of the Monitoring F/W prototype, as he was not involved in the F/W's development itself. Therefore, this section contains a summary of the key points from the direct user feedback in terms of usability and benefits of the F/W's prototype.

PRO'S First of all and mentioned as the most important point from the developer is the fact that in this use case the Monitoring F/W prototype provides a comfortable way to get access to needed debugging information under undisturbed operational conditions (see explanation at the beginning of Section 16.4.3.1). Giving an example, he used the F/W for getting information about the correct operation of the light sensors and the onboard file system. These values are not needed as TM for normal operation, but are most valuable in the case of a malfunction of the FloatingSat.

Debugging during normal Operation

Another benefit was the integration of log-statements in the source code of the `Modes`-application (see Figure 16.18) with the intention to report the reception of a TC. As the FloatingSat does not use a protocol like CCSDS/PUS there is no acknowledge of reception and correct execution of TCs implemented. With the Monitoring F/W this features was relatively easy added to the system by the developer by integrating corresponding log-statements - using the DEBUG log level.

TC Acknowledge without CCSDS/PUS

Moreover, especially after intense debug sessions resulting in the integration of multiple logging statements the Monitoring F/W revealed its strength in terms of monitoring data traceability. Hastily typed-in PRINTF-statements containing only basic information (like e.g. only variable values without additional textual information in the form of strings and/or formatting information) tend to mess up and lead to a cluttered console output, where the developer can only hardly figure out which output message originates from which PRINTF-statement. The uniform, configurable output technique by the F/W's appenders and layouts overcome this shortcoming by displaying information like application- and thread-name of each statement (see Figures 16.19). Hereafter the developer was able to trace each of the debugging outputs to its original location within the source code.

Traceability of Monitoring Data

Finally the F/W was not only functional valuable during design, implementation and operation of the enhanced FloatingSat experiment, also the easy-to-use API of the F/W was emphasized by the developer after finishing his work.

Easy-to-use API

CON'S It has to be mentioned that the F/W's prototype revealed a weakness when it comes to add new appenders besides the already implemented console appender: the `Appender`-class is implemented in a way that only strings can be handled properly in terms of layout. The GUI appender which was developed in this use case (see

Figure 16.19a) had to handle these strings manually. It would have been better to detach the layout from the `Appender`-base class by providing a specialized appender which offers these kinds of concrete layouts as an option to the developer. In any case, `LogItems` should be send directly to new appenders by the `doAppend`-method of the `Appender`-class. By doing this the developer is able to decide by himself how to handle these log items. For an overview on the current implementation the reader may have a look at the corresponding sequence diagram in Figure 16.3.

18.5 FINAL TEST ANALYSIS & COMPARISON

Within this section the test results of all three use cases of the Monitoring F/W prototype will be compared, taking into account the individual comparison results from Sections 18.2.3, 18.3.3 and 18.4.3. As already done in each of these comparison sections, we will start with some remarks about the difficulties during testing. Then – later on in this section – a closer look at some general findings regarding the design of the F/W will be taken. The section is completed by a section regarding safety analysis.

18.5.1 *RODOS Version*

As the reader may have noticed, during the testing of the prototype of the Monitoring F/W different RODOS version were used: use case 1 is based on the version 111, use case 2 is using version 110 and within use case 3 even two different RODOS versions were used: version 113 for the F/W's embedded part and version 116 on the F/W's receiver side running on a desktop PC. In general it can be stated that for the different tests performed the RODOS version is not relevant, as the differences between the used versions which could potentially influence the testing are only very small and have no effect on the test results[86]. More important is to use a RODOS version higher than 110, as from this number on – besides a global HAL – a new version of the RODOS gateway (cf. to Section 5.2.1.2) was introduced which contains modifications needed by the Monitoring F/W to function properly. While the source code stays exactly the same, the file system structure of the F/W was adapted to that of RODOS ("api"-, "src"- and "support"-directories) when switching from version 110 to 111, thus making it easier to migrate the F/W's source to the actual RODOS directories later on (the F/W currently resides under the "support_libs"-directory of RODOS).

18.5.2 *Difficulties during Tests*

There have been some difficulties during the performed tests with the Monitoring F/W prototype – especially when comparing the results with each other. But as already explained in Paragraph Focus and Quality of Tests on Page 156, as long as the tests stay on a phenomenological basis these irregularities can be accepted. They only need to be addressed when a fine-granular in-depth analysis of the F/W is needed.

86 Further information regarding the new features introduced in the current RODOS version and a brief history of past versions can be found in the `VERSION` text file within the root directory of the current RODOS source code repository.

18.5.2.1 *Varying Measurement Techniques*

As already explained in Paragraph Profiling Technique on Page 157 the different tools (GRMON & GPROF) used for measuring the profiling values are based on different measurement techniques. This makes a direct comparison of the results gathered almost impossible, and even a relative comparison on a phenomenological basis is difficult.

18.5.2.2 *Varying H/W Platforms*

Further difficulties arose during the comparison of results from different H/W platforms. This is most likely due to varying H/W specific features, like e.g. different BSPs resulting in different libraries to be linked together or the presence/absence of a floating point unit. Because of this, e.g. comparing the results from the use case 1 "Leon3->Leon2 switching" to the use case 2 "Leon3->Linux switching" makes no sense. It also does not make sense to repeat the LEON2 test with use case 2 due to the persistent communication-bug (see Paragraph LEON2 UART Handshake Problem on Page 171), and – vice versa – it also does not make sense to repeat the Linux-only tests for use case 1 due to the interference of the RODOS scheduler with the Linux scheduler (see Section 18.5.2.4 below).

Another area which is influenced by platform specific details is the size of the boot images. Here, especially a comparison between the RODOS boot image which is started on the embedded target host and the RODOS file which runs on the development host under Linux is not possible. The main reason lies in the fact, that tools like GRMON do not load the whole boot image into the RAM of the target H/W, but only parts of it. Therefore, the focus was on the comparison of the results from the test performed with and without the Monitoring F/W prototype on the same platform.

18.5.2.3 *Finding Common Methods for Comparison*

As a result from the above described varying measurement techniques and H/W platforms it is hard to find common methods/functions which are sampled during all test results and can be used for comparison. This can be best observed when testing use case 2 on LEON3- & Linux-based environments and trying to compare the results (see for example the Paragraph Message Transport on Page 197).

18.5.2.4 *Linux as Test Environment*

As already mentioned on Page 192, the Linux scheduler strongly interferes with the RODOS scheduler. This also affects the middleware-based message transport (see Section 18.5.6.1 and Footnote 88). Similar effects were also observed in numerous other projects, although they used different measuring techniques. It turned out that testing under Linux (or any other guest operating system) is not suitable in order to make reliable and repeatable statements about the resource-consumption and timing behavior of a S/W which is meant to be run embedded. Therefore it is better to test directly on the target H/W.

18.5.3 *Findings: Static Changes*

As already explained in Section 18.5.2.2 for the comparison of the first two use cases, we will mainly concentrate LEON3-based tests as the test environment which both have in common.

18.5.3.1 *Findings: Lines of Code*

The calculation of the effective LoC with and without the Monitoring F/W prototype can be found in Section 18.1.

While the LoC of the Monitoring F/W API are constituting only 17% (file size only 4%) of the whole RODOS API, this changes drastically for the LoC of the files contained in the source (SRC) directories of both S/W components: here, the LoC of the Monitoring F/W amounts almost three quarter (73%) of the LoC within the RODOS core, the total file size is also significantly higher (37% of the total file size of the RODOS core).

As a conclusion we can say that even if the LoC within the application layer stays exactly the same when using the F/W prototype, this does not hold for the core components. Especially for an embedded system the amount of LoC of the Monitoring F/W prototype is too high and needs to be reduced within the flight version of the F/W.

18.5.3.2 *Findings: Boot Image Size*

The average increases in the size of the boot images are apparent in both use cases. Taking the mean values, for use case 1 there is a moderate increase of 16%, whereas for use case 2 the increase in boot image size is about 40%. Looking at the LEON3-based tests as the common test environment for both use cases, from use case 1 to use case 2 the boot image size is almost doubled and the number of symbols almost tripled (cf. to the corresponding paragraphs on Pages 165 and 181).

One of the reasons behind this increase may be the increased number of log statements which are transformed to `LogItems`. This structure – used to handle the content of the log messages – consumes too much static memory. This is mainly caused by the strings which are stored within the `LogItems`. To solve this issue we could limit the length of the strings by design, setting it to a fixed size. But this would also limit the amount of information which can be transported to the receiver side – opening up e.g. possibilities for misinterpretations on the receiver side[87] if the length of the strings is too short due to limited resources of the system. Therefore, for the Monitoring F/W flight version we have chosen a different approach – called *pre-processing* – which is explained in detail in the corresponding Section 17.3.3.

18.5.4 *Findings: Dynamic Changes*

18.5.4.1 *Findings: Load from Scheduler*

Comparing the results of the Monitoring F/W prototype usage from use case 1 (cf. to Section 18.2.3) to the results from use case 2 (cf. to Section 18.3.3) leads to contradictory findings: while for the Scheduler-related threads a load-increase in use case 1 when switching from LEON3- to LEON2-based H/W can be observed, the load from the same threads goes down in use case 2 when switching from LEON3-based H/W to a pure Linux-based environment. As explained in Section 18.5.2.2, these differences are most likely originated in the varying test-platforms and their specific behavior – here in terms of scheduling.

Comparing the results from the tests on the common LEON3-based H/W platform (see Sections 18.2.1.4 and 18.3.1.4) reveals a significant increase in both use cases which is lower in use case 2:

- `Thread::findNextToRun(long long)`:
  430% increase in use case 1 ⇒ 300% increase in use case 2
- `hwTrapSaveContextContinue`:
  510% increase in use case 1 ⇒ 215% increase in use case 2
- `__asmSwitchToContext`:
  530% increase in use case 1 ⇒ 225% increase in use case 2
- `Scheduler::schedule()`:
  460% increase in use case 1 ⇒ 265% increase in use case 2
- `hwSysTrapTrampoline`:
  1010% increase in use case 1 ⇒ 325% increase in use case 2

This findings imply that – even though the usage of the Monitoring F/W prototype has a big impact on the runtime behavior of the RODOS scheduler within a reduced, minimalistic test scenario – this impact does not increase but actually goes down when changing to a more realistic scenario.

Additional Load from Scheduler high but decreasing

As a final conclusion we can say that the additional processor load resulting from the scheduling of the additional threads of the monitoring F/W prototype

- is too high in a scenario, where the Monitoring F/W is the main actor, but

87 E.g. by the people working in the checkout- or control-rooms during a satellite project

- seems to go down when more applications are integrated into the system.

The latter statement would require further research which can be an area of expansion in the future.

18.5.4.2 *Findings: Load from Log-Threads*

Coming to the processor load directly caused by the Monitoring F/W, the percentages of the sampling rates of the corresponding threads is almost negligible when comparing use case 1 to use case 2. This holds for the overall comparison from Section 18.2.3 to Section 18.3.3 as well as for the direct comparison of the LEON3-based test scenarios in Sections 18.2.1.4 and 18.3.1.4.
Especially for use case 1 we were expecting the biggest influence of the Monitoring F/W, because there is almost no load from other applications. This indicates that the core components of the Monitoring F/W makes economic use of the resources regarding the load on the processor.

18.5.5 *Findings: Message Transport*

18.5.5.1 *Findings: Load from Middleware Threads*

In general we can say that there are a very low sampling rates from all functions involved in sending messages. In addition, the overall comparison of use case 1 with use case 2 (see Sections 18.2.3 and 18.3.3) shows a decrease when switching to LEON2-based H/W in use case 1 (in use case 2 there was no relative comparison possible).
For the LEON3-based test scenarios (see Sections 18.2.1.4 and 18.3.1.4) we can say that for use case 1 the load is negligible and for use case 2 still very low. But an additional important finding resulting from the comparison of the absolute values with and and without the F/W in these two scenarios is, that the increase is much lower in use case 2. The fact that there is already a higher load when using PRINTF in use case 2 leads us to the assumption, that the load coming from sending monitoring messages over the RODOS middleware is not as high as initially expected after analysis of use case 1.

18.5.5.2 *Findings: Middleware Data Throughput*

The test results from use case 3 (see Section 18.4) reveals the high utilization of the RODOS middleware by the messages from the Monitoring F/W prototype. Therefore and as a lessons-learned this drawback has significantly influenced the design section of the flight version of the Monitoring F/W which was equipped with a powerful preprocessing approach to eliminate as much redundant information to be transmitted as possible (see Section 17.3.3).

18.5.6 *General Findings*

18.5.6.1 *Message Transport not robust enough*

Loss of Monitoring Messages becomes apparent during Transport

During testing of the scenario using the Monitoring F/W in Section 18.2.1.3 it was found that some of the some monitoring messages were lost during transport from the Nexys3 board to the Linux host. This is primary caused due to a bug in the UART gateway of RODOS (cf. to Section 16.1, Figure 16.1 for details about the RODOS gateway functionality), where some middleware messages are lost from time to time. This becomes especially apparent within the test cases in which the log messages contain long strings: the Monitoring F/W is currently implemented in a way that big monitoring messages resulting from long strings are split over several (up to eight and sometimes even more) middleware messages (cf. to Section 16.1.2.1 about implementation details). With this feature the user has the possibility to send very big messages over the network. Though, the drawback of this design is that if one single middleware message is lost the whole monitoring message is corrupt and cannot be reconstructed. For example, this can be observed in Line 37 of Listing 18.5. This problem gets even more serious when sending monitoring messages from the embedded system to a connected host, where the RODOS-image (as the message receiver) runs on-top of another operating system [88]. A separate test under Linux – in which messages are being sent constantly without a break – revealed, that the interference from the Linux- and the RODOS-scheduler is causing serious problems: after every received messages an interrupt shall wake up the RODOS receiver thread, but due to the high frequency of incoming messages the Linux scheduler does not activate the corresponding Linux process fast enough. Therefore, a lot of messages were lost (overwritten) – in this additional test almost 10% of the messages were affected. This means that if a monitoring message is e.g. split over eight to ten single middleware messages, almost no monitoring messages can be re-constructed on the receiver side – leading to a total loss of this monitoring information.

88 This effect can also be observed in other software systems and F/Ws being tested on an underlaying operating system:

- In the paper of Sreenuch et al. [78] it is described how fluctuations caused by a Java Virtual Machine (as the F/W's software base) and Windows XP (as the host OS) have strong impact on the test results.
- The test of the flight S/W of the MASCOT mission [38] were not only performed on the final H/W, but also in a fully virtual environment. During some of these tests the non-deterministic nature of the virtual, Linux-based test environment was the reason that the test failed. Further investigations revealed that in these test cases the pre-defined timing constraints were not met due to time slips (e.g., the reception of a telemetry packet has being logged a bit later than expected).

As a conclusion we can summarize that improvements have to be undertaken for future versions of the Monitoring F/W regarding the robustness of the message transport:

Message Fragmentation and Reception Acknowledge

- monitoring messages shall not be split/fragmented over several middleware messages, and
- the correct reception of the monitoring messages must be guarantied.

18.5.6.2 *Output Buffer Handling & Missing Message IDs*

As explained in Section 18.3.3, the buffer-handling of `LogInputBuffer` on the embedded side already caused some problems with the transport of monitoring messages. The same holds for the output-side and the reception of messages by the `LogOutputThread`. These problems can probably be avoided by the introduction of a unique ID for every monitoring message. Like it was practiced in Section 18.5.6.1, in order to prove this assumption and do some further investigation, we undertook some additional tests with the following findings:

Because `LogInputBuffer` as well as `LogOutputThread` are both using `SyncFifo` for synchronization purposes[89] the messages are received in the correct order. With this design no problems occur – even if more than one `log`-message is being sent from one thread. But this changes if messages get lost during message transport, like explained in the above Section 18.5.6.2: if for example the buffer of the `LogOutputThread` is full, separate log messages are mixed into one:

```
[INFO] TestApp.TestThread: Battery Current (mA): Battery Volt-
age (mV): 4838
```

Here we can see that from the first message the actual value and the final `endl`-statement were lost. Therefore, the `LogOutputThread` has not recognized this message as completed and continued with the next incoming message (which was also only received partially). If there is only one log message per thread this behavior improves a little bit:

```
[INFO] Test2App.Test2Thread: Battery Current (mA): Battery Cur-
rent (mA): 870Battery Current (mA):
```

Also in this example, two messages were falsely mixed together, but at least it was always the same log message. The main problem seems to be that incomplete messages on the receiver side are not correctly recognized, but discarded. This problem could be avoided in future

Missing Message ID

89 Either by inheritance or by direct object creation, cf. to the implementation Section 16.2 of the prototype for further details, and also to Section 16.1.3 on buffer design.

versions of the Monitoring F/W by the introduction of a unique message ID for every monitoring message which would facilitate the recognition and separation of different monitoring messages from each other, even if they originate from the same thread.

18.5.7 *Safety Analysis*

Depending on the mission criticality different types of safety mechanisms and analysis techniques have to be applied to ensure the correct functional and timing behavior of the embedded S/W. Currently the Monitoring F/W prototype is not included into a S/C's flight S/W. Therefore, no major emphasis was put on undertaking a static code analysis, a schedulability analysis or detailed measurements of the memory consumption. This was changed for the flight version of the F/W to be developed for the Eu:CROPIS satellite mission (cf. to Section 19). Yet, it has to be mentioned that e.g. the prototype's buffering and message transport on the embedded side was designed in way to avoid a blocking of the embedded S/W which can occur either within the process of collecting monitoring data or during sending them out.

18.6 final remarks

After analyzing in detail the test results from the Monitoring F/W prototype in Section 18.5, these final remarks shall serve as a conclusion and short summary of what was achieved and hence pave the way for the development of the prototypes successor – the flight version of the Monitoring F/W.

The initial test-goals identified in the Paragraph Focus and Quality of Tests on Page 156 were mostly fulfilled. The functional behavior was correct for all three use cases defined in Section 16.4. Furthermore using the Monitoring F/W prototype does not significantly increase the resource consumption of the embedded S/W. The higher load initially produced by the scheduler went down when changing to a more realistic scenario. However it has to be mentioned that serious concerns arose after looking on the increase in size (LoC and boot image size) and bandwidth utilization after the F/W's integration – both are too high.

But as the major goal of the design and development of the F/W's prototype was on serving as a testbed for the new idea of Unified Monitoring, the focus was on demonstrating the benefits of the new monitoring capability and – equally important – the usability of the new F/W. As these goals were fulfilled with great success, the test results of the prototype will serve as a lessons-learned for the flight version, thus a lot of the findings here will be addressed in the General Design Driver-Section herein. Especially the traceability of the monitoring items will be further developed which also result in enhanced query options for the user. In addition the overall F/W size will be reduced as well as the transport of monitoring information will be optimized.

19 TESTING THE MONITORING FRAMEWORK FLIGHT VERSION

A thorough testing of the F/W's flight version is essential as it is supposed to be integrated as an additional application into the flight S/W of the CompactSat mission Eu:CROPIS. Therefore the demands on the flight version tests are much higher than on the prototype.

For this reason, the test approach of the prototype (see Paragraph Structure of Tests on Page 156 was adopted: on the one hand tests regarding the static- (LoC and boot image size) and dynamic changes of using the F/W flight version in comparison to printf-statements were taken over, creating an appropriate test scenario for the latter one. But in addition these tests were extended by adding module tests and by making the profiling tests much more detailed than the ones undertaken for the prototype.

Test Restrictions

Some of the final tests required for every mission critical embedded S/W could not be done: although the Eu:CROPIS flight S/W development is currently in phase C (see Chapter 6 for details about the ECSS project phases) the final flight H/W is not yet available. Therefore there are currently no results from tests performed on the EM, Qualification Model (QM) or FM (cf. to Figure A.3) of the satellite's H/W. Also the H/W support from the libCOBC S/W library (see Section 17.1.2) for the development breadboards is not completed: currently no data output and configuration from/to the Nexys3-boards [90] is supported – neither over a serial (UART) connection nor over the standard TM/TC interface. And as both F/Ws – prototype as well as flight version – depend on the communication capabilities of the underlying S/W infrastructure (RODOS/libCOBC), at current state tests directly on the H/W are not possible. This is particularly unsatisfactory due to the fact that the tests of the prototype revealed the importance of these kinds of H/W-based tests (cf. e.g. to Sections 18.5.2.4, 18.5.6.1 and Footnote 88).

Facing this state of current S/C development, the following restrictions for the flight version tests arise:

- Testing can take place on the development host running Linux only, using UDP for testing the communication between the embedded and non-embedded F/W parts shown in Figure 17.7 [91].
- Resulting from this, the implemented flight version communication improvements in order to save bandwidth – in particular

90 These are indeed the same boards as the ones used during the prototype tests of use case 1 and 2 (cf. to Sections 18.2 and 18.3).

91 This is similar to the prototype test of use case 2 under Linux (cf. to Section 18.3.2

the new pre-processing approach and the numerous configuration possibilities at runtime – could not be tested in a real-world end-to-end scenario. Therefore no quantified measurements are available to verify the expected improvements over the F/W's prototype in this area.

- Detailed schedulability analysis makes only sense directly on the H/W, so this had to be skipped also[92].

19.1 UNIT TESTS

In order to fulfill the requirements on safety critical S/C flight S/W and to be most robust against future changes (e.g. for a possible integration of the F/W as a dedicated service into the libCOBC library), the F/W's flight version is thoroughly tested by unit tests on module level [see 28, sec. 6.1]. Using *GoogleTest*[93] as the underlying test infrastructure, almost 100% code coverage could be achieved without detecting major errors. Special emphasis has been put on testing very specific and also marginal cases, implementing multiple helper- and mock up classes in order to test as many modules in isolation as possible.

19.2 PROFILING TESTS

The profiling tests which were undertaken for the prototype proved to be extremely useful to get detailed insights into the dynamic changes after integration the F/W as a new S/W component into the system. For the flight version the profiling tests were even more extended, using the toolset Valgrind[94] to analyze a test scenario similar to the one used for prototype testing of use case 2. Two test-threads running on top of the libCOBC library were created and then used to produce load on the processor and – at the same time – to feed the F/W's flight version with monitoring data. Each test was repeated three times, each test-run took five minutes. Like in the prototypes use cases no configuration commands were sent during this time. As depicted in Figure 19.1, the tests were undertaken using a pure virtual Linux-based environment and utilizing an internal UDP connection.

More details about the test fixture and an in-depth description of the measured results can be found in [28, sec. 6.2][95].

The profiling tests were focusing on two key aspects:

92 What is more, for doing an in-depth schedulability analysis also the majority of the Eu:CROPIS S/W applications have to be implemented which is not the case at the time of writing this thesis.
93 http://code.google.com/p/googletest/
94 http://valgrind.org
95 See also Figure 6.2 herein as an example for a textual call graph of the flight version's profiling results – similar to the prototype call graphs shown in the Figures 18.13, 18.15 and 18.17)

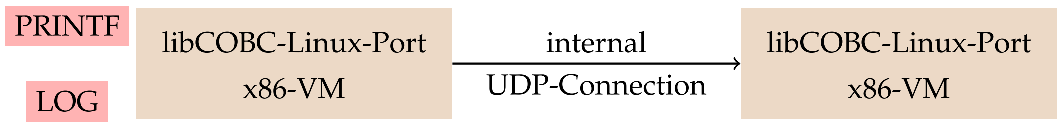


Figure 19.1: Setup of Test Use Case for the Flight Version

1. the behavior of the F/W's flight version in comparison to the usage of printf (similar to the testing focus of the prototype use cases 1 and 2 in Sections 18.2 and 18.3), and
2. the scalability of the F/W in dependence on the amount of monitoring data to be collected from the applications.

19.2.1 *Processor Load*

RELATIVE TO PRINTF For the comparison of the processor load all the measured values from all involved threads on the embedded side were summed up. After doing this, a comparison of the printf-based test scenario with the scenario, where only the `log`-methods of the flight version were used, revealed, that in the latter case the processor load was 1.75-times higher.

SCALING Repeating the tests with a stepwise increase of monitoring messages (8, 32 & 128) sent per second showed a linear growth in the processor load, as it can be seen in Figure 19.2.

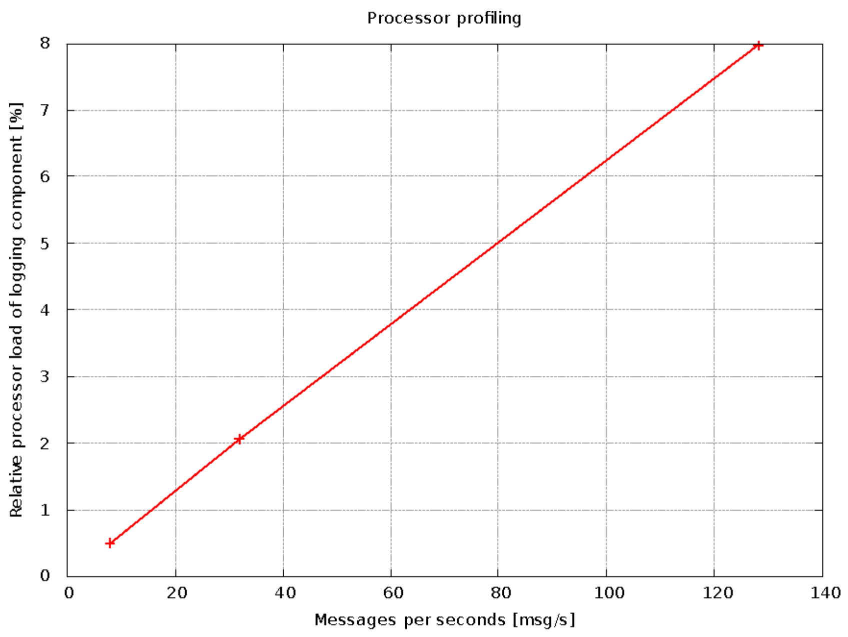


Figure 19.2: Growth of Processor Load in Dependence on the Number of `log`-Statements within the Source Code [see 28, pic. 6.3]

19.2.2 *Memory Consumption*

RELATIVE TO PRINTF Figure 19.3 shows the memory consumption of the printf test scenario in comparison to the F/Ws flight version. Only the statically allocated stack memory was measured, as the dynamic heap memory allocation is not used within the embedded side of the S/W. The increase rate after switching over from printf- to log-statements is 1.9.

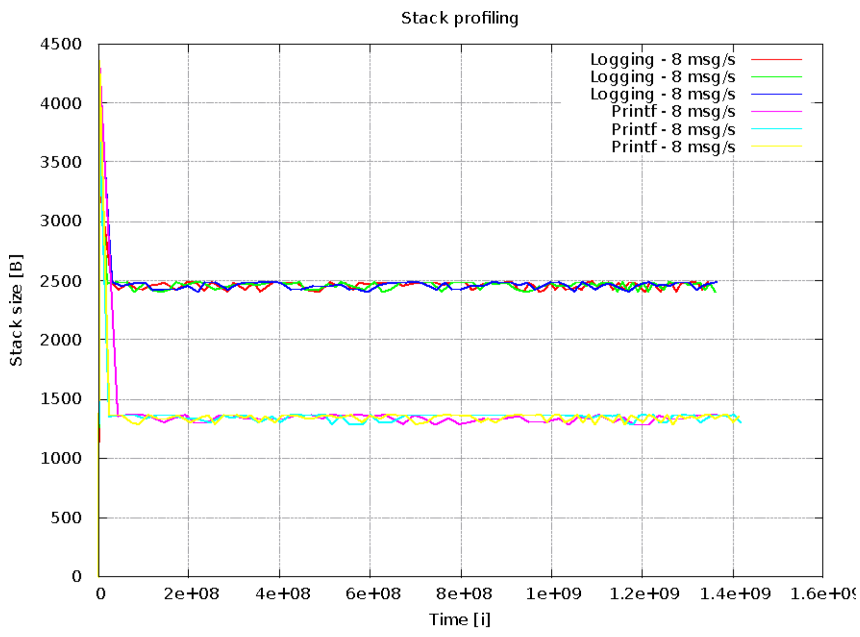


Figure 19.3: Stack-Consumption of Monitoring F/W in Comparison to printf [see 28, pic. 6.4]

SCALING As for testing the memory load, the tests regarding the memory consumption were repeated with a varying number of monitoring messages (again 8, 32 & 128) sent per second. Here Figure 19.4 reveals that the stack consumption does not increase, but rather stays on a constant level which is independent from the collected amount of monitoring data.

19.3 STATIC CHANGES

LOC Like it has been performed for the prototype in Section 18.1, also for the embedded part of flight version the LoC were counted and compared to the LoC of libCOBC-core components, namely the *rtos*- and the *time*-components [see 13, sec. 3]:

	Netto LoC
libCOBC CORE	3020
log CORE	2276

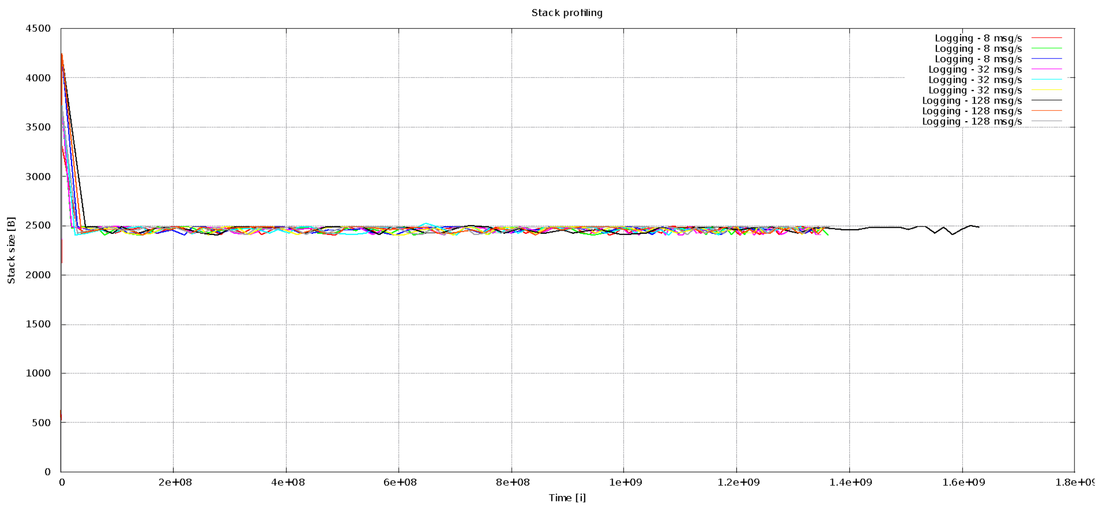


Figure 19.4: Stack-Consumption of Monitoring F/W with increasing Number of Monitoring Messages per Second [see 28, pic. 6.5]

BOOT IMAGE SIZE Table 6.5 in [28] reveals, that the size of the boot image after integration of the Monitoring F/W flight version is more than twice as high as in the printf-based scenario (175,5kB with printf-in comparison to 415kB with log-statements).

19.4 STATIC CODE ANALYSIS

In order to ensure not to have any deadlocks or errors within the code a static code analysis of the embedded parts of flight version was undertaken – using the tool *Polyspace* [49] [50]. As a result it turned out that no critical errors were present, so it can be concluded that also during operation no crashes of the flight S/W due to the integration of the Monitoring F/W will occur. The detailed annotated analysis can be found in [28].

19.4.1 *JSF Coding Standards*

In addition to the static code analysis based on formal methods, Polyspace also offers the possibility of checking the source code in terms of the compliance with specific coding-styles and -standards. The libCOBC library is programmed with the adherence to the JSF AV C++ coding standard (short: JSF++) [11], following strictly embedded design guidelines. As the F/W's flight version is running upon libCOBC and might also be integrated later on as an additional service herein (see Section 22.2.1.2), Polyspace was used to verify that both – the libCOBC as well as the flight version's source code – are compliant to the JSF++ standard.

19.5 FINAL REMARKS

Within this section we will summarize the findings during testing of the Monitoring F/W's flight version. In addition, where applicable, a comparison to the findings from the prototype will take place – especially the Final Remarks Section on Page 215. This will lead us to the upcoming EVALUATION Chapter where the final analysis after the development and testing of the both F/W versions will take place, referencing to the current shortcomings listed Chapter 13.

FUNCTIONALITY Functionally the majority of the requirements on the Monitoring F/W are fulfilled. This holds for the initial requirements placed on the prototype as well as for the further developed requirements on the flight version (see corresponding Appendices A.4 and A.5).

STATIC CHANGES

LOC Comparing the LoC of the prototype and the flight version with the core components of the S/W they are running upon reveals optimization potential in terms of further reducing their complexity: their LoC amount is 75% of the LoC of the RODOS core (see Section 18.5.3.1) as well as of the libCOBC core respectively.

BOOT IMAGE SIZE 16% and 40% increase in the size of the final boot image of the prototype (cf. to Section 18.5.3.2) and the doubling in size of the flight versions boot image indicates, that also here a need for optimization is evident. Nevertheless, these dramatic increases likely depend on our constructed test scenarios and might relativize later on, as the size of the test boot images is still in the order of KBytes, the final boot images – with all applications and libraries integrated – will be in the order of MBytes.

DYNAMIC CHANGES The profiling results from the flight version are as promising as the ones from the prototype: both revealed an acceptable overhead for the embedded system taking into account the new functionalities provided. Even if the profiling of the flight version threads was not as sophisticated as the one for the prototype – RTOS threads responsible for the additional scheduling effort and the message transport were not considered – the profiling of the flight version covered a whole new test area: the scalability of the F/W in dependency upon the amount of collected monitoring data. It has been shown that even if the number of log statements which are processed per second is increasing, the stack consumption does not increase, and the additional load on the processor increases linear. The not-changing stack consumption has its roots within the fact, that the log methods are processed sequentially, and that no dynamic memory allocation is allowed. The linear increase of the processor load with the increasing number of log messages is expected due to higher processing effort. Would the increase be e.g. exponential the usage of flight version would be extremely limited and major work would have been needed in order to enhance the scalability and fix this load-leak[96].

What is particularly important is the introduction of the security mechanism coming within the flight version and explained in Section 17.2.6 (Safety Aspects): only by the implementation of these techniques it is ensured that the load increase and stack overhead of the Monitoring F/W flight version stay on this acceptable level.

Constant and tolerable Overhead is ensured

USABILITY There is currently no direct feedback from the application developers of the EU:CROPIS S/W regarding the usability of the F/W's flight version – like it resulted from the tests of the prototype's use case 3 in Section 18.4. Nevertheless, the expected acceptance from this kind of end-users is even greater than with the prototype: the requirements and the user interface of the flight version (cf. to Section 17.2.3) were designed hand-in-hand with the developers of the AOCS S/W application and the libCOBC S/W library - taking their feedback and design suggestions into account. The similarity to the usage of printf and the good extensibility shown in Paragraph 17.3.5 are additional benefits of the flight version in terms of handling.

MATURITY Last but not least, the additional tests which were undertaken for the flight version like the amount of available unit tests, the static code analysis with Polyspace and the adherence to the JSF++ coding standards are making this version of the Monitoring F/W a mature flight software component – ready to be flown with the Eu:CROPIS satellite and integrated into the libCOBC library.

96 The wording is inspired by the corresponding term for an unwanted increasing memory consumption – the *memory leak*.

20 EVALUATION

Within this section we will have a look at the requirements being setup in Chapter 15 and the achievements made during design, development and test of the Monitoring F/W prototype as well as its flight version.

As already mentioned in Section 15 the initial requirements for the prototype shown in Diagram 15.2 (and in more detail in Appendix A.4) are addressed and continued within [63, sec. 4.1]. This thesis also contains a *Validation* Section 4.2 which verifies the functional and non-functional requirements with respect to their fulfillment by the design and implementation of the prototype.

In addition, also the requirements which are used for designing the flight version of the Monitoring F/W (cf to Section 17.2) are described in detail in [28, sec. 4.1]. They were traced with respect to their fulfillment in a corresponding Requirements Traceability Matrix (RTM) which can be found in the appendix Section A.5.

Therefore, the focus in this chapter lies on summarizing the achievements made by the development of both Monitoring F/W versions. We will do this by going through the list of current shortcomings which is presented in Chapter 13 and show, how these where addressed in terms of improving the State-of-the-Art in S/C monitoring.

20.1 WHAT WE HAVE ACHIEVED - ADDRESSING THE CURRENT SHORTCOMINGS

Within this section we will have a look at the shortcomings of the current monitoring techniques which were identified in Chapter 13. We will see if and how the points listed could be addressed by the Unified Monitoring idea and its realizations – the Monitoring F/W prototype and flight version – in order to improve the situation.

20.1.1 *Addressing the Functional Shortcomings*

USAGE IN OPERATIONS The Monitoring F/W has proven its usability in operational scenarios in an impressive way.
For the F/W's prototype especially the use case 3 revealed the benefits and previously impossible ways of looking into the innards of the onboard avionics system (cf. to Section 16.4.3). The FloatingSat experiment setup is very close to a real space mission in the way that the satellite is physically not accessible during its operation. The air bearing table and the closed chamber with the star background forbid the

usage of a cabled connection in order to access the console-based debugging outputs from the PRINTF or ERROR-statements inserted into the source code of the embedded S/W. For a detailed description of the benefits resulting from the usage of the F/W's prototype in the operational scenario of the FloatingSat cf. to the Usability-part within Section 18.4.

USAGE IN CASE OF A SYSTEM CRASH In case of a system crash the capabilities of tracing back the faulty behavior which led to the crash (see Figure 11.1) strongly depend on the design of the onboard avionic system: if it is possible to access historical TM when the OBC boots up again after a crash, then of course also the stored information from the Monitoring F/W's flight version can be transferred to earth (cf. to the Housekeeping and Integration of Monitoring Framework- Section 17.1.3) for further details).
In either way the possibility of adjusting the granularity of monitoring information in order to look in greater detail into a suspicious subsystem makes the F/W most valuable when running FDIR activities after a system crash. This is described in the next paragraph.

CONFIGURABLE GRANULARITY The configuration of the Monitoring F/W in terms of the log granularity at compile-time and runtime is an important requirement which was already set at an early planning stage of the F/W. It was fulfilled with great success from the prototype and the flight version: for the prototype the initial global log level can be changed at runtime per TC. For the flight version this configuration in terms log level adjustment is possible even on a per-application basis. The evolution in the recording of the relevant S/C data – as shown in Figure 3.2 – can therefore be considered as accomplished.

COMPLIANCE WITH SPACE CODING STANDARDS The field of application of the F/W's prototype is currently limited to breadboard- and laboratory-experiments. Therefore no effort was undertaken to force its compliance to the RODOS coding standards. This changed with the development of the F/W's flight version, where special emphasis was put on ensuring that the coding standards set for the whole flight S/W of the Eu:CROPIS project were also applied to the flight version's source code [97].

DYNAMIC PRESENTATION OF INFORMATION The dynamic presentation of monitoring information when sending them to the non-embedded parts of the F/W is one of the central ideas of this thesis.

97 The Eu:CROPIS coding standard is a tailored version of the JSF AV coding standard [11].

Appenders and layouts[98] take over this functionality which is an important part of the concept of Separation of Concerns as presented in Section 14.2 (see esp. Figure 14.2). For both F/W implementations appropriate appenders were developed. Besides the standard console appenders especially the `GUIAppender` developed for the prototype's use case 3 (cf. to Section 16.4.3.2) has to be highlighted here: running at the same time parallel to the `ConsoleAppender` – but addressing a different kind of user (namely the pupils experimenting in the DLR School_Lab) – it reveals the great advancement over the conventional RODOS PRINTF technology in a most vivid way. This becomes particularly obvious in direct comparison with the terminal output shown in Listing 12.2.

ENHANCED TRACEABILITY DURING OPERATION With the introduction of the Monitoring F/W the extensive error search in lengthy log files which contain the S/C's historical TM belongs to the past: the monitoring protocol itself, the storage of information within the F/W and way the F/W processes the monitoring information all the way through – from the S/C to the groundstation – provide completely new debugging and monitoring possibilities. The evolution in the traceability of relevant S/C data which is depicted in Figure 3.3 is accomplished by the design of the F/W's prototype and flight version and can be seen most clearly in the way the monitoring information is presented to the end user by appropriate handlers (see paragraph above).

20.1.2 *Addressing the Performance Shortcomings*

The main points regarding the current performance were listed in Section 13.2. Some of them were already addressed in Section 19.5.

Improvements have to be made in terms of memory consumption, as both F/Ws have too many LoC and an significant increase within boot image size. This results – among other things – from the fact the F/Ws are highly intrusive and violate the *principle of observability* (cf. to Section 13.2.5)[99]. This outcome is mitigated to some extend by the fact, that the load-increase on the processor is only a little bit higher in comparison to printf-based debugging scenarios (see e.g. Section 19.2). However, for the flight version this overhead is deterministic, meaning that it is constant over time and has a linear scalability in terms of the amount of monitoring data. Taken into account

98 The terms *appender/layout* and *handler/formatter* are interchangeable and used in parallel during this thesis.

99 A possible attempt for decreasing the LoC size of both F/Ws would be to find unused code components (methods, classes, ...) by looking into detail into the textual call graphs resulting from the profiling. Regarding the increase of the boot image size it was already mentioned in the *Static Changes*-paragraph in Section 19.5 that this might relativize when all S/W components are integrated in the final boot image.

the functional achievements presented in Section 20.1.1, these drawback are mostly acceptable.

On the other hand the improvements being made regarding the runtime performance also have to be mentioned: both F/Ws are designed in such a way that they take into account the embedded environment they will be working in, putting special design- and test-effort into key points like thread safety or static memory allocation (cf. to Section 16.1.3 and the Deterministic Buffer Handling-Paragraph on Page 141) which also improves the F/W's reaction to interrupts. For the flight version these real-time aspects were even more emphasized as within this F/W version also the efficient network transport in terms of the saving of bandwidth was a main concern.

Taking into account the amount of monitoring data gathered (cf. e.g. to the "Enhanced Traceability during Operation"-Paragraph above) and the low transmission bandwidth available, the F/W's flight version is well prepared for the resulting challenges: in order to keep the overhead for the monitoring data to be transferred on a minimal level (trying to provide deterministic overhead also here), new techniques like pre-processing (see Section 17.3.3) and compression were introduced. Besides the configurability at runtime which already exists for the prototype, the flight version also avoids the sending of static information and concentrates on the transmission of relevant information by the usage of application-specific debug levels.

Last but not least it has to be mentioned that both F/Ws need further testing not only in terms of schedulability and data transmission directly on the flight H/W, but also regarding the functionality of real-time configuration.

20.1.3 *Addressing the Usage Shortcomings*

As described in Section 13.3.1, reducing the effort in writing, testing and changing the application-specific source code by combining debugging and housekeeping into one information world was a major goal for the development of the Monitoring F/W. The functionality achieved and the performance analysis undertaken (see Sections 20.1.1 and 20.1.2 made clear that the F/W has this potential. Nevertheless, Section 20.2 below (and also Section 17.1.3) explains the reasons for currently not going for a complete replacement of the traditional housekeeping within a satellite mission. Therefore, these effort-related shortcomings could be only partially fulfilled. This is also the reason why the error susceptibility, when maintaining and syncing both information worlds, does not improve considerably which is also true for the code readability. However, all of these shortcomings would vanish immediately after switching over the Monitoring F/W completely.

On the other hand it has to be mentioned that the development and usage of the various F/W appenders (see Sections 16.4.3.2 and 17.3.5) already brought massive improvements regarding the learning curve for new tools in every development phase of a S/C. As the F/W together with its appenders can be used during the whole S/C's lifecycle, the unified access to the monitoring data makes the usage of heterogeneous toolset superfluous (cf. to Figure 3.1).

Uniform Console Output

Another benefit which was not even apparent when designing the Monitoring F/W results from the fact, that all developers involved in the coding process have typically their own taste/preference regarding the usage of printf-statements and especially the format-strings herein. When debugging the final boot image with all integrated S/W components this might lead to a cluttered and confusing console output. By using the Monitoring F/W and its corresponding appenders the output of monitoring data is unified to large extent – yet leaving it open to the programmer to develop an appender according to his special needs.

20.2 TOWARDS A COMPLETE REPLACEMENT OF THE TRADITIONAL S/C HOUSEKEEPING

As the preceding Section 20.1 has revealed, a lot of improvements over the current state of the art has been made by introducing the idea of Unified Monitoring and its implementation within the Monitoring F/W.

Missing Points to complete the End-to-End Scenario…

Nevertheless, some issues are still open. One of the most important ones is the support for the End-to-End scenario as introduced right at the beginning of Chapter 15, meaning a complete replacement of the traditional housekeeping capabilities of a S/C by the Monitoring F/W. The main reasons for not going for the full solution within this thesis are:

- The Monitoring F/W needs an *in-flight testing* before replacing the housekeeping as one of the mission critical parts of the onboard S/W. At current state a failure of the F/W within the Eu:CROPIS mission would not include the risk of a complete mission loss. Therefore the flight version of the F/W can be seen as an in-orbit technology demonstrator before performing a housekeeping replacement within the next compact satellite mission (see Section 17.1 for details about the compact satellite missions yet to come and their timeline).

- Currently, there is no support for the monitoring messages from the TM and TC data formats used. This especially concerns the ECSS PUS services used for the TM/TC of the Eu:CROPIS mission. Like explained in the corresponding flight version design Section 17.2.2, monitoring information is sent to ground using normal TM, as well as the configuration of the F/W is undertaken using the normal TC chain[100]. Especially the integration of monitoring messages into the real-time TM which is displayed by the various tools used in the control rooms of the space operation centers (cf. to Section 9.3 would be of enormous benefit for very early reactions in the case of FDIR events.

- In addition – even if the PUS services would support the transmission of information from the Monitoring F/W – there is no operational support from the groundstation. To be more specific, the non-embedded parts of the F/W's prototype and flight version (see e.g. Figures 16.2 and 17.7) will have to be integrated into the mission's operational infrastructure.

. . . and how to solve them.

Once a successful F/W operation is proven during the envisaged first compact satellite mission, the following points will have to be addressed in order to solve the issues addressed above and to prepare the F/W for the housekeeping replacement in the next mission:

- Integration of the transmission of monitoring messages and the configuration of the F/W within the standard PUS services. For this a dedicated handler could be developed (cf. to the concept of logger, handler and formatter explained in Section 16.1) which takes over this task. For the F/W flight version the required extension points are described in the Paragraph Changing the communication technique on Page 150. Possible PUS services for sending down the source packets containing the monitoring information would be service no. 3 (Housekeeping) and service no. 12 (Onboard Monitoring Service).

- If the information and configuration of the Monitoring F/W is successfully integrated into the CCSDS/PUS infrastructure, also the ground segment has to be complemented with appropriate handlers and formatters in order to collect and display the received monitoring information as well as to configure the F/W by TC. Ideally, these handlers/formatters are integrated or connected to the *SCOS* [85] and/or *SATMON*[101] S/W which are used e.g. at the German Space Operations Center (GSOC) and the European Space Operations Center (ESOC).

After these hurdles are cleared the Monitoring F/W will be ready to be used within the whole space system.

100 Like mentioned in Paragraph Changing the communication technique on Page 150, the integration of the monitoring packages into corresponding CCSDS/PUS services is work in progress.
101 `http://www.moltek.com/index.php/satmon`

Part VIII

CONCLUSIONS

21 SUMMARY

Within this thesis we present a new philosophy in monitoring S/Cs: the unification of the various kinds of monitoring techniques used during the different lifecylce phases of a S/C. This goal is accomplished by the development of a new methodology – called *Unified Monitoring* – which embeds a unified access to status information into the S/C development process. In addition, a new infrastructure was developed to put the Unified Monitoring methodology into practice: the *Monitoring F/W* universal monitoring system.
The challenging requirements set for this F/W are:

- Separation of Concerns as a design principle (dividing the steps of logging from registered sources, sending the monitored information to connected sinks and displaying them accordingly)
- usage during all mission phases
- usage by all actors (EGSE engineers, groundstation operators, etc.)
- configurable at runtime, especially regarding the level of detail of logging information
- very low resource consumption

We first developed a prototype of the Monitoring F/W as a support library for DLR's own research RTOS RODOS. This prototype was tested on dedicated hardware platforms relevant for space (development boards with LEON-based processors), and also on a satellite demonstrator used for educational purposes in DLR's own School_Lab facility.

As a second step, the results and lessons learned from the development and usage of this prototype were transferred to a real space mission: the first satellite of the DLR compact satellite series. Within this mission, the software of the avionic subsystem was supplemented by the flight version of the Monitoring F/W which enhances the traditional housekeeping capabilities and adds extensive filtering and debugging techniques for monitoring and FDIR needs to the mission's operational scenario. As for the prototype, the F/W is completed by non-embedded counterparts running on the development computers as well as on the EGSE in the integration room, making it most valuable already in the earliest stages of S/C development.

22 OUTLOOK

Future work in the area of Unified Monitoring and the usage of the Monitoring F/W can be divided into

- further tests of the F/W's flight version,
- upgrading the existing use cases,
- the usage of the new methodology and its corresponding F/W in completely new areas of application by extending the functionality which was implemented so far, and
- the development of an underlying data model which will serve as the theoretical foundation for the Unified Monitoring.

22.1 FLIGHT VERSION TESTING

As already mentioned in Paragraph Changing the communication technique on Page 150 and also at the beginning of Chapter 19, the tests of the flight version could not be carried out completely due to some technical restrictions. Ongoing development progress regarding the current implementation of the F/W's flight version and the Eu:CROPIS satellite bus will overcome some of these shortcomings:

1. An implementation of an interface for the F/W's flight version which supports the CCSDS/PUS-based TM/TC chain will enable the sending of monitoring data to ground as offline TM, and also the configuration of the F/W at runtime.
2. The integration of the libCOBC S/W library on the engineering models of the Eu:CROPIS OBC.
3. The development of the missions-dependent S/W applications (like thermal- and power control).

The first two points will enable an end-to-end testing of the functional and performance behavior of the F/W's flight version. Together with the latter one a detailed schedulability analysis of the final boot image on the flight H/W during the operational scenario will be possible.

22.2 UPGRADING CURRENT USE CASES

22.2.1 *Upgrading the embedded F/W Part*

22.2.1.1 *Historical Monitoring Data*

Although the FDIR functionality of the F/W's flight version (see Section 17.2.6) allows some kind of insight into historical monitoring data, there is still room for improvement: both F/Ws – prototype as well as flight version – might be extended in a way that their central buffers handling does not discard monitoring data with insufficient log levels, but flag them as historical data[102]. Doing this would path the way to downlink the current monitoring data as realtime TM, and the flagged data as historical TM using different virtual channels [see 20, sec. 13.2].

22.2.1.2 *Operating System Monitoring*

Within the current use cases of the Monitoring F/W prototype as well as within the Eu:CROPIS mission the monitoring of information was limited to the application layer only (cf. e.g. to Section 16.1.1). But it would be highly beneficial in terms of debugging capabilities if the usage of the Monitoring F/W would also be possible from lower S/W layers. The low-level debugging techniques within these layers still heavily rely on printlining (see beginning of Chapter 11 and also Section 12.1.1). The F/W's integration herein would allow to detect errors even earlier and hopefully avoid their propagation to higher levels in the embedded system (see Figure 11.1).

First steps into this direction would be the integration of the F/W's prototype into the RODOS core layer (see Figure 5.5) and of the F/W's flight version as an additional service into the libCBOC library (see Figures 17.3 and 17.4). This would allow to monitor at least parts of the underlying S/W platform and is of course limited on how much the current implementation of the Monitoring F/W depends on its usage of OS-inherent functionalities (like e.g. threads).

22.2.2 *Upgrading the non-embedded F/W Part*

Future activities regarding the further development of the Monitoring F/W are not limited to the usage within the next compact satellite project. We aim to include the – so far missing – support from the groundstation as well. This can be achieved (1) by transferring the monitoring messages via the communication chain to ground, and (2) by adding appropriate tools for real-time configuration and display of real-time information to the infrastructure of the control room (cf. to the corresponding Section 20.2).

Integration into ESA's Common Core Initiative

It may be advantageous for this intend that the European Space Agency (ESA) has recently started a new activity with the goal to introduce a new groundstation infrastructure: the *Common Core*-activity. The envisaged design of the Common Core will support multiple protocols and tools [66, 67, 68], e.g. PUS [20] is as well supported as its possible successor, the Mission Operations Services (MOS) [9]. This drastic upheaval in the currently established infrastructure – based

102 This of course strongly depends on the onboard storage capabilities.

on CCSDS-protocols and using tools like SCOS and SatMon – offers the possibility to integrate new tools and techniques. For the Monitoring F/W's non-embedded part this would be a good chance to develop and integrate new appenders which are working hand-in-hand with the other Common Core tools. From this point of view, the *embedded* F/W part can be understood as the satellite's counterpart of ESA's ground-based Common Core initiative.

All in all we can say, that if the extensions described above are accomplished all prerequisite are fulfilled in order to perform a complete replacement of the traditional housekeeping functionality of a satellite with the new Unified Monitoring and its realization, the Monitoring F/W. This would lead to a seamless integration of the Monitoring F/W not only into to the S/C itself, but into the whole space system[103].

22.3 EXTENSION TO NEW USE CASES

Besides extending the current usage scenarios of the Monitoring F/W, one can also think about introducing the idea of Unified Monitoring to completely new fields of application.

New Monitoring Infrastructure for Launchers

One of these areas is the avionic system of future launchers, providing them with a completely new monitoring infrastructure. First steps into this direction will be taken within a current EU project which just started at the time writing this thesis [16]. Herein it is foreseen to increase the avionics system performance of current launchers (e.g. the Ariane 5 ECA[104]), inter alia by the integration of an enhanced version of the Monitoring F/W, and later on transfer the experiences gained to future launcher systems.
The central idea presented in this proposal is to equip and extend the current embedded parts of the Monitoring F/W with self-configuring techniques and mechanisms in order to selectively observe, pre-process, and compress sensor data. Currently, TM is collected using a quite limited number of predefined acquisition frames that depend on the launcher's mission and with a low resolution of sensor data. On board launchers, the timeline for diagnostics based on TCs is too tight. The self-configuring Monitoring F/W that automatically focuses on irregularities achieves latencies that cannot be matched by a remote operator. Using this technology within launcher avionics it will be possible to select the most important data to monitor, record and transmit. The ratio between monitored raw data and sent data will be higher than 2500, while today this ratio is 1. This drastic increase

103 Cf. to [52], a space system is defined as a complex system, consisting of the combination of all subsystems responsible for transportation, infrastructure, communicating facilities as well as all humans involved in operating these systems – on earth and in space.

104 `http://www.esa.int/Our_Activities/Launchers/Launch_vehicles/Ariane_5_ECA2`

of processed onboard sensor data enables a more thorough analysis of mission data, providing additional measures to increase reliability and improving the observation of onboard physical phenomena.

Besides the self-configuring mechanism, a formal model (cf. to Section 22.4) of the Monitoring F/W will be developed in order to ensure

- that the self-configuring Monitoring F/W must not overload the system with too much data traffic, and
- that the monitoring reacts to detected faults guaranteeing safe operation of the system.

Further details regarding this future use case can be found in Appendix A.7[105].

22.4 MONITORING DATA MODEL

Another important effort to be undertaken in the future would be the development of a stronger theoretical background for the different S/C information worlds presented in Chapter 9. In order to establish a basis for this background a common data model to describe the monitoring data has to be developed – maybe using formal languages like TLA+/TLC as specified in Lamport [41] (cf. to Verhulst et al. [82] for an example of the usage of these languages during the development of an RTOS).

The monitoring data model shall be used during all project phases (see Chapter 6), within all S/C information worlds (see Chapter 9 and by all actors (see Chapter 7). For the development of this model we would therefore have to take the following parameters and questions – among others – into account:

SOURCES OF INFORMATION Which entities (technologies, tools, etc.) store and distribute the information into the single information world (satellite housekeeping, telemetry application, . . .)?

SINKS OF INFORMATION Which entities (technologies, tools, etc.) receive the information from the single information world (development host computers, EGSE checkout equipment, groundstation tools, . . .)?

DISTINCTIVE FEATURES OF INFORMATION How is the information to be exchanged between the sources and sinks structured and what is it composed of (content, data types, amount of data, . . .)?

105 Not mentioned here are further benefits identified in the area of additional services which could be provided towards a launcher customer. For example, the Monitoring F/W could be further extended to be also used for doing compressed video supervision or supplementary payload health monitoring.

In addition, also the suitability of the resulting data model in terms of compression- and filtering-abilities has to be examined.

Towards Determinism

Having such a data model would not only provide us with a better knowledge of the system state, but also bring us to – or at least nearer to – a deterministic behavior of the whole system: if we have a detailed and exact knowledge of the current system state by using the Monitoring F/W we might be able to determine all possible resulting states in the future by using appropriate prediction methods.

Part IX

APPENDIX

A APPENDIX

A.1 SysML Notations

Within this thesis we make extensive use of the Systems Modeling Language (SysML) which is an extension of the UML. Please refer to the book of Tim Weilkiens (see [84]) for a detailed description of all available diagram types. For the diagrams used in this dissertation the reader will find a short explanation regarding the used notation within this appendix chapter.

A.1.1 *System Context Diagram*

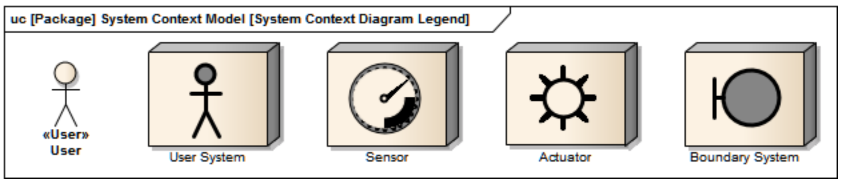

uc [Package] System Context Model [System Context Diagram Legend]
«User»
User
User System
Sensor
Actuator
Boundary System

Figure A.1: System Context Diagram Legend

A SysML system context diagram as shown in Figure A.1 is a special form of an UML use case diagram. The following symbols are used:

USER A human actor of the system

USER SYSTEM An external system (interacting with the system to be developed)

SENSOR Environmental sensor

ACTUATOR Environmental actuator

BOUNDARY SYSTEM An external system with special importance for the system to be developed

A.1.2 *Communication Diagram*

The symbols used in the communication diagrams are shown in Figure A.2. They have the following meaning:

ENTITIES are objects representing system data.

BOUNDARIES are objects that interface with system actors.

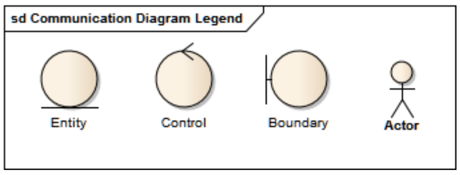


Figure A.2: Communication Diagram Legend

CONTROLS are objects that mediate between boundaries and entities. They orchestrate the execution of commands coming from the boundary by interacting with entity and boundary objects.

A.2 MODEL PHILOSOPHY

The following two pictures give an exemplary overview of the models to be developed within each level of design (Figure A.3) and their integration into the AIV plan (Figure A.4). Both diagrams were taken from the official *Eu:CROPIS* project documentation (cf. to "EC-SYS-PL-0027-AIV Plan.pdf") – however, they are rather typical for most satellite missions.

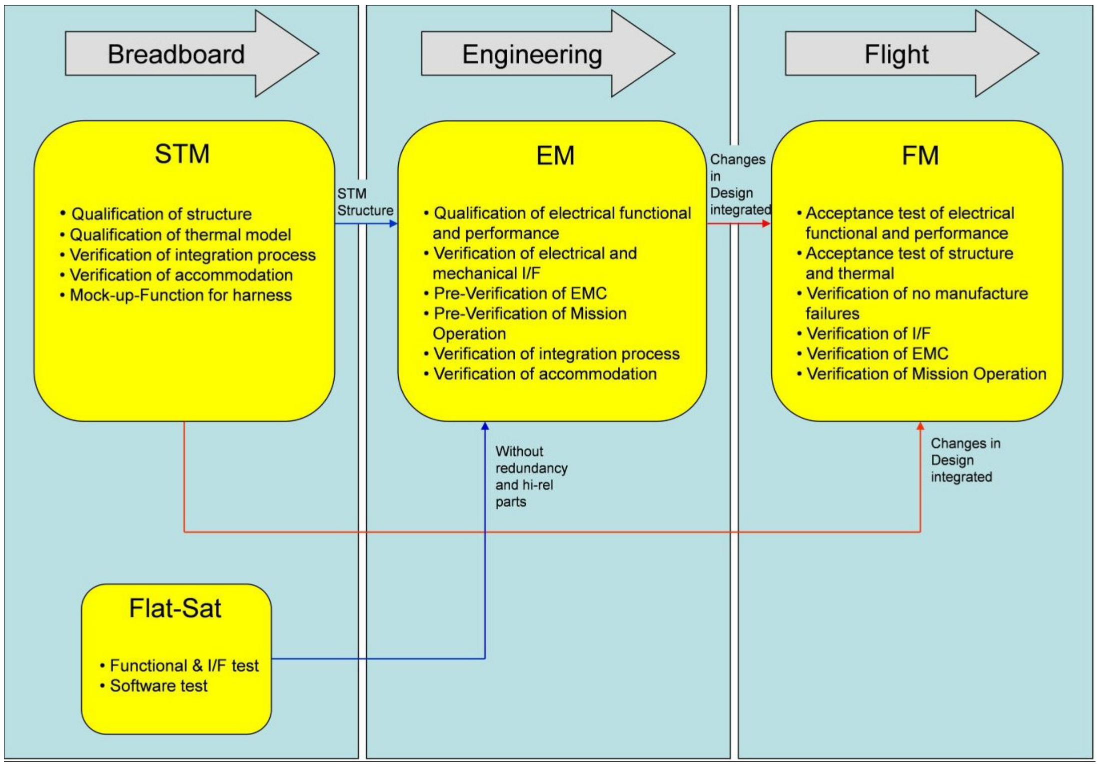
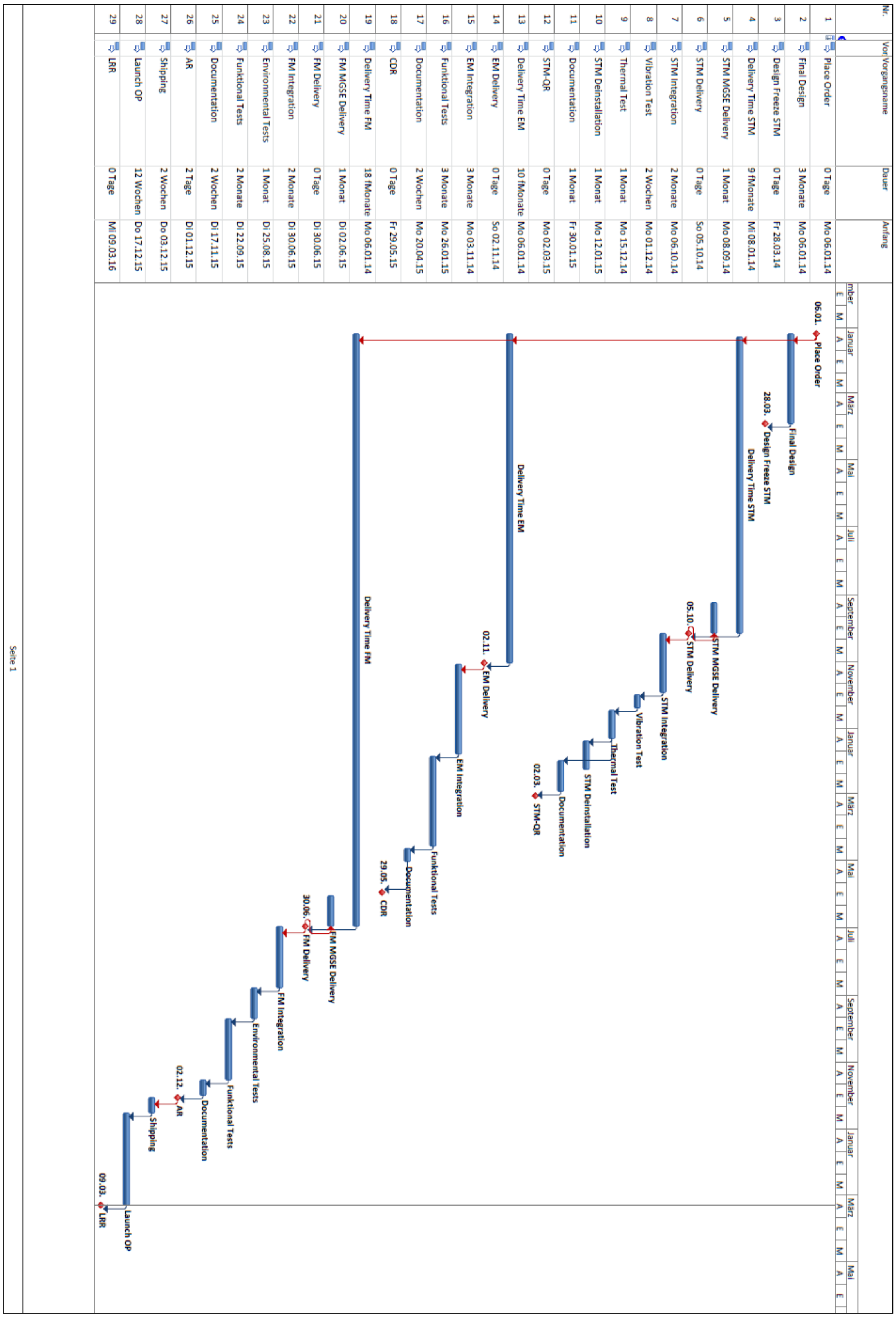

Breadboard
Engineering
Flight
STM
• Qualification of structure
• Qualification of thermal model
• Verification of integration process
• Verification of accommodation
• Mock-up-Function for harness
STM Structure
EM
• Qualification of electrical functional and performance
• Verification of electrical and mechanical I/F
• Pre-Verification of EMC
• Pre-Verification of Mission Operation
• Verification of integration process
• Verification of accommodation
Changes in Design integrated
FM
• Acceptance test of electrical functional and performance
• Acceptance test of structure and thermal
• Verification of no manufacture failures
• Verification of I/F
• Verification of EMC
• Verification of Mission Operation
Without redundancy and hi-rel parts
Changes in Design integrated
Flat-Sat
• Functional & I/F test
• Software test

Figure A.3: Eu:CROPIS Model Philosophy



Nr.	Vorgangsname	Dauer	Anfang
1	Place Order	0 Tage	Mo 06.01.14
2	Final Design	3 Monate	Mo 06.01.14
3	Design Freeze STM	0 Tage	Fr 28.03.14
4	Delivery Time STM	9 fMonate	Mi 08.01.14
5	STM MGSE Delivery	1 Monat	Mo 08.09.14
6	STM Delivery	0 Tage	So 05.10.14
7	STM Integration	2 Monate	Mo 06.10.14
8	Vibration Test	2 Wochen	Mo 01.12.14
9	Thermal Test	1 Monat	Mo 15.12.14
10	STM Deinstallation	1 Monat	Mo 12.01.15
11	Documentation	1 Monat	Fr 30.01.15
12	STM-QR	0 Tage	Mo 02.03.15
13	Delivery Time EM	10 fMonate	Mo 06.01.14
14	EM Delivery	0 Tage	So 02.11.14
15	EM Integration	3 Monate	Mo 03.11.14
16	Funktional Tests	3 Monate	Mo 26.01.15
17	Documentation	2 Wochen	Mo 20.04.15
18	CDR	0 Tage	Fr 29.05.15
19	Delivery Time FM	18 fMonate	Mo 06.01.14
20	FM MGSE Delivery	1 Monat	Di 02.06.15
21	FM Delivery	0 Tage	Di 30.06.15
22	FM Integration	2 Monate	Di 30.06.15
23	Environmental Tests	1 Monat	Di 25.08.15
24	Funktional Tests	2 Monate	Di 22.09.15
25	Documentation	2 Wochen	Di 17.11.15
26	AR	2 Tage	Di 01.12.15
27	Shipping	2 Wochen	Do 03.12.15
28	Launch OP	12 Wochen	Do 17.12.15
29	LRR	0 Tage	Mi 09.03.16

Figure A.4: Eu:CROPIS AIV Plan

A.3 TET TELEMTRY VALUES

Table A.1: Extract from the Telemetry Values of TET-1

Telemetry Value	Explanation
Application *Temperature Control*	
BIT_HEATING_BATERIES	Heater turned on/off
BIT_TEMPCNTR_ENABLED	Temperature control is enabled/disabled
Application *Power Control*	
BIT_STRING_SHUNTED	Solar arrays shunted/not shunted
BIT_CHARGECNTR_ENABLED	Battery charge control enabled/disabled
STRING_SHUNT_CNT	Counter for shunting of solar arrays
Application *Downlink Manager*	
BIT_DOW_ENABLED	Downlink enabled/disabled
BIT_DOW_HI_BITRATE	Downlink high transfer rate selected/ not selected (bus and payload using downlink at the same time)
BIT_DOW_ONLY_SBC	Only the bus is using downlink/bus and payload are using downlink at the same time
Application *Software Watchdog*	
BIT_WAT_TIMING_CRITICAL	Time to next Watchdog reset (not) critical (<500ms)
BIT_WAT_RESET_ENABLED	Watchdog may (not) reset the node
Application *Surveillance*	
BIT_SUR_STACK_CRITICAL	Stack overflow may (not) occur
SW_VERSION	Software version of the boot image
SAT_MODE	Integer number representing the mode of the satellite
MIN_STACK	If under 1 KByte a Worker-Monitor-Switchover is initiated in order to prevent crash of computing node

Continued on next page

Table A.1 – continued from previous page

Telemetry Value	**Explanation**
MEM_CHECK_CNT	If memory check counter is not increasing, a Worker-Monitor-Switchover is initiated
BOOT_CNT	Boot counter
BOOT_ADR	Boot address
PARITY_ERRORS	Number of parity errors
BOARD_ID	The Id of the active node
Application *Commander*	
BIT_CMD_HW_IF_OK	Hardware interfaces (not) OK
BIT_CMD_WAITFORDATA	Data blocks (not) arriving
BIT_CMD_DATA_OK	Last data block was (not) OK
BIT_CMD_UPLOAD_ACTIV	Upload data transmission (not) ongoing
REASON_FOR_SAFEMODE	Values representing reasons for the satellite to go to safe mode [106]
CMD_CC_IMM_CMD	Command-counter for immediate commands
CMD_CC_TIMETAGED_CMD	Command-counter for time-tagged commands
CMD_SAFEMODELIST_VERSION	Version of the list of commands to be executed when the satellite is in safe mode
CMD_DUL_STATE	Status of data upload
Application *Redundancy Manager*	
OTHER_NODE_BOOT_CNT	Boot counter of redundant node
OTHER_NODE_BOOT_ADR	Boot address of redundant node
OTHER_NODE_TIME	Time since boot of redundant node
OTHER_NODE_UTC	UTC time of redundant node
OTHER_NODE_LAST_UPDATE	Last update time of redundant node
OTHER_NODE_SAT_MODE	Satellite mode of other node
Application *Time Manager*	
LOCAL_TIME	Time since boot
UTC	UTC time
Application *Housekeeper*	
HK_RATE	Housekeeping data rate period

Continued on next page

Table A.1 – continued from previous page

Telemetry Value	Explanation
Application *Navigation*	
GPS_STATUS	GPS receiver status
ALAMANAC_WEEK	Almanac week
ONS_STATUS_FLAGS	ONS status flags
ONS_WGS_POS_X	X position (in cm)
ONS_WGS_POS_Y	Y position (in cm)
ONS_WGS_POS_Z	Z position (in cm)
Common Error Counters	
ERRCNT_SUM	Sum of all error counters
ERRCNT_INDEX	Error counter index
ERRCNT_VAL	Error counter value

Table A.2: Extract from the Analog Values of TET-1

Analog Value	Explanation
Currents	
SBC_CPU_CURR	Satellite Bus Controller (SBC) Currents
SBC_MAIN_MEMORY_CURR	*ditto*
SBC_PARITY_MEM_CURR	*ditto*
SBC_SHADOW_MEM_CURR	*ditto*
SBC_ALTERA_CURR	*ditto*
SBC_FLASH_MEM_CURR	*ditto*
SBC_5V_CURR	*ditto*
SBC_3_3V_CURR	*ditto*
S_C_LOAD_CURRENT	Bus current
BATT_CHARGE_CURRENT	Battery current
SA_LOAD_CURRENT	Solar array current
NVS_CURRENT	NVS current
PAYLOAD_CURRENT	Payload current
PAYLOAD_ARRAY_CURRENT	Payload 9 current
Voltages	
PAYLOAD_ARRAY_VOLTAGE	Payload 9 voltage

Continued on next page

Table A.2 – continued from previous page

Analog Value	**Explanation**
MAIN_PWR_BUS_VOLTAGE_1	Battery voltage
MAIN_PWR_BUS_VOLTAGE_2	Bus voltage
Charges	
BATTERY_CHARGE_SG1	Battery Stack 1 charge
BATTERY_CHARGE_SG2	Battery Stack 2 charge
Temperatures	
BATTERY_STACK_TEMP_1	Battery Stack 1 temperature
BATTERY_STACK_TEMP_2	Battery Stack 1 temperature
MAIN_PLATE_TEMP_1	Main Plate temperatures
MAIN_PLATE_TEMP_2	*ditto*
MAIN_PLATE_TEMP_3	*ditto*
MAIN_PLATE_TEMP_4	*ditto*
TEMP_SBC_A	SBC A temperature
TEMP_SBC_B	SBC B temperature
PDH_BOARD_1_TEMP	PDH Board 1 temperature
PDH_BOARD_2_TEMP	PDH Board 2 temperature
TEMP_S_B_TRANSPONDER_1	S-Band Transponder 1 temperature
TEMP_S_B_TRANSPONDER_2	S-Band Transponder 2 temperature
MAGNETOMETER1_TEMP	Magnetometer temperature
PAYLOAD_PLATFORM_TEMP_1	Payload-related temperatures
PAYLOAD_PLATFORM_TEMP_2	*ditto*
PAYLOAD_TEMP_1	*ditto*
PAYLOAD_TEMP_2	*ditto*

A.4 MONITORING FRAMEWORK: REQUIREMENTS ON THE PROTOTYPE

Within this appendix section the requirements diagram shown in Figure 15.2 will be explained in more detail.

Functional Requirements

«requirement» *Status:* Mandatory *Priority:* High *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
Anforderungen an die Funktionalität und Bedienung des Frameworks.

Distributed Monitoring

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
As the RODOS framework itself, the monitoring framework shall be also able to run distributed, meaning to act also over network boundaries.

Divide content from representation/formatting

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
When sending monitoring/debug information of the middleware, only the pure information shall be send over the RODOS middleware layer.

The receiver of these kind of information could be a monitoring component (=application), which is integrated into the RODOS framework as another building block. The component can than also be used for receiving telecomands for configuration of the framework.

All king of formatting will take place later on, preferrably on the client side. A possible client could be a ground station, which connects to the monitoring framework using appropriate handler/formatter.

Reception of commands for configuration of the framework

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

Record all kinds of Logging & Monitoring Informationen of a Satellite

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
In particular:

- Debug Informationen from PRINTFs
- Telemetry-Information, initially being sent to some kind of commander apllication

Support for different log frequencies

«requirement» *Status:* Proposed *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
In order to avoid unnecessary traffic load on the middleware by sending monitoring information the frequency of recording and sending these information should be adjustable.

In the extreme case the user should be able to set the frequency to zero, which is equal to turning off the framework, meaning that no more information is recorded. Yet the framework should stay configurable (cf. to requirement Configuration during runtime)

Support for log granularities

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
Log-Level (Critical, Information, Debug, ...)

Configuration during runtime

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* High
Phase: 1.0 *Version: 1.0*
The following properties of the framework shall be configurable during runtime:

- Log-Level (Critical, Information, ...)
- Log-Frequencies
- Turning on/off the framework
- Message distribution: locally and/or over network boundaries using gateways
- connected handler (sinks) & formatter

Resource Requirements

«requirement» *Status:* Mandatory *Priority:* High *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

Runtime behavior of the overall system shall be not affected.

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
Functionality of the RTOS, which are responsible for steering and controlling of the satellite, shall not be affected by the monitoring framework.
This is especially important when accessing sensors, actuators and antennas.

Minimal memory consumption at runtime.

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
Memory consumption during runtime shall be minimal, as resources are always limited in embedded systems.

Minimal processor usage at runtime

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
The time consumption of the processor at runtime should be as minimal as possible, as this would have effects on the timing and therefor functional behavior of the embedded system.

Technical Requirements

«requirement» *Status:* Mandatory *Priority:* High *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

Design based on existing frameworks

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*
We don't want to reinvent the wheal. There are several embedded logging framework on the market (c-logger, logging-cpp, ...) but none of them especially designed for satellites. Netherless a frame shall be chosen, which does the separation of concerns (cf. to requirement Divide content from representation/formatting) and is divided into

components similar to monitoring, handler & formatter.

Easy to understand API

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

The API of the framework shall be easy to use from the point of view of the application programmer. Ideally, functions like PRINTF or the operator "<<" should be used, because they are well known to C++ programmers (cf. to requirement Implementation in C++)

Good expandability

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

It should be easy to add additional Log-Handler, Log-Level, etc.

Implementation in C++

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

As the framework should be integrated into the RODOS framework and it' middleware, the programming language shall be C++.

Integration in RODOS Middleware

«requirement» *Status:* Mandatory *Priority:* Medium *Difficulty:* Medium
Phase: 1.0 *Version: 1.0*

A.5 MONITORING FRAMEWORK: REQUIREMENTS ON THE FLIGHT VERSION

This section contains the Requirements Traceability Matrix (RTM) for the flight version of the Monitoring F/W.

ID	Object Text	Status	Comment
	Subsystem Description		
	The following document lists all requirements that are placed onto the Command and Data Handling Logging Component (CDH-MON). The logging component will supplement the CDH software subsystem. It is meant for storing all kind of logging information and therefore enhancing the housekeeping functionality of the satellite. The design of the system will be driven by three major design goals: - keeping all kinds of information - scalability - performance The requirement implementation is described within the design file EC-CDH-RP-XXXX_CDH_Monitoring_Design_Definition_File.		
	Subsystem Requirements		
	Functional		
CDH-MON-27	Logging of debug information (formerly known as PRINTF)	implemented	Implemented in „ApplicationLoggerBase::logPreprocessed“
CDH-MON-15	Configurable by telecommand: - log granularity (depth of information, source of information) - log level (criticality of informtation)	implemented	A specialization of „ControllerLinkBase“ transmits configuraton packets to a specialized „ControllerBase“, which passes them to „Central::processConfigPacket“.
CDH-MON-41	Telecommands shall be receiveable via CCSDS telecommand interface and via EGSE (UART)	not implemented	Not implemented due to missing UART and CCSDS telecommand implementation in libCOBC.
CDH-MON-42	The logging output shall be send via CCSDS telemetry and via EGSE (UART)	not implemented	Not implemented due to missing UART and CCSDS telemetry implementation in libCOBC.

CDH-MON-39	The log level shall be configurable on a subsystem level.	implemented	Each „pus::Application“ can have a object of „ApplicationLogger“. And target APIDs are passed via configuration packets to „Central::processConfigPackets“. For managing all application loggers, an implicit list is used.
CDH-MON-40	In case of an FDIR trigger the logging component shall output historical logging data	partly implemented	Limited support for historical data after a FDIR trigger. For more historical data an improved buffer handling is needed. During copying of message packets in the application logger buffers to the central buffer only packets with valid log levels are being copied. Non valid packets (log level too high) are being overwritten.
CDH-MON-48	FDIR mode shall have a separate log level.	implemented	„ApplicationLoggerBase::m_applicationLogLevelFdir“.
CDH-MON-21	Protocol consisting of: - application-ID - subsystem-ID - message-ID - variable data	implemented	Implemented in class „Header“.
CDH-MON-26	Divide logging content from representation ("separation of concerns")	implemented	Logging content (messages) is represented by „RawDataPacket“ (encoded) and „DataPacket“ (decoded) and displayed by a specialized „DataAppenderBase“.
CDH-MON-35	The amount of generated logging data per time shall have a global upper limit.	implemented	Global upper limit is the size of the central buffer.
CDH-MON-49	The amount of generated logging data per time for each logger instance shall have an upper limit.	implemented	Each buffer size of the application loggers are setting the upper limit per subsystem. Additionally a byte quota can be set via telecommand.
CDH-MON-36	The logging component shall avoid sending constant data which can be known at the receving side.	implemented	Format string of each logging message is replaced by a message Id with „LogPreprocessor“.
CDH-MON-43	The logging component shall provide houskeeping data about its internal state	partly implemented	Due to missing implementation of the housekeeping in libCOBC the logging component doesn't provide housekeeping data. But the buffers are recording the buffer state in „BufferStatus“.
	Performance		
	Using the Monitoring Component shall result only in a minimal overhead in terms of ressource usage. The functional behaviour of the system shall not be affected.	implemented	Stack usage is constant and its size depends on the amount of logging messages. CPU usage increases linearly with increasing amount of messages.
CDH-MON-9	The size of the final boot image shall increase no more then <TBC> %		

CDH-MON-11	The memory used by each thread shall increase no more than <TBC> %		
CDH-MON-13	The processor usage shall increase no more than <TBC> %		
CDH-MON-14	Usage rate of the software bus shall increase no more than <TBC> %		
CDH-MON-33	The logging component shall not block the control flow of the application	implemented	Tested with unit tests, an integration test, profiling tests and a static code analysis.
CDH-MON-34	Changing the log level shall not affect the performance of the application.	implemented	Tested with an integration test.
CDH-MON-38	Data shall be send with minimal overhead (e.g. binary instead of ascii representation, compression etc.).	implemented	7 bit encoding and ZigZag encoding is used for integers.
	Implementation		
CDH-MON-17	The Monitoring component shall run on top of the Application Software Interface provided by the Eu:CROPIS S/W subsystem. (cf. to document EC-CDH-ICD-0098)	implemented	Runs on top of libCOBC.
CDH-MON-45	The logging component shall have its own application identifier (APID)	not implemented	Due to missing telemetry/telecommand implementation in libCOBC the logging component doesn't have an own APID.
CDH-MON-19	The implementation language shall be C++03. (cf. to document EC-CDH-TN-0099)	implemented	Implementation language on the embedded side is C++03 and on the ground station side C++11.
CDH-MON-22	Based on existing designs (Maius Logger, RODOS Monitoring Framework): - global logger component - application logger (small)	implemented	Multiple application loggers, but no global application logger. „Central“ copies all data packets of all application loggers periodically.
CDH-MON-24	Easy-to use API (e.g. by overloading C++ operators)	implemented	„ApplicationLoggerBase::log“ is similar to printf.
CDH-MON-37	The API shall be type-safe.	implemented	Type-safe with the traits „DataTrait“.

CDH-MON-28	Good extensibility	implemented	Classes for transmitting and receiving data and configuration packets are all base classes which need to be specialized („DataTransmitterBase“, „DataReceiverBase“, „ControllerBase“, „ControllerLinkBase“). Data appenders for displaying the data can be specialized from „DataAppenderBase“. Configuration inputters can be specialized from „ConfigInputterBase“. Data types can be added in „DataTrait“.
CDH-MON-47	Good portability	implemented	Runs on top of libCOBC. Portability depends on it.

A.6 COMPILE-TIME CONFIGURATION OF FLIGHT VERSION: PARAMETERS.H FILE

Listing A.1: parameters.h File of the Flight Version of the Monitoring F/W

```
/*
 * Copyright (c) 2014, German Aerospace Center (DLR)
 * All Rights Reserved.
 *
 * See the file "LICENSE" for the full license governing this code.
 */
#ifndef COBC_LOG_PARAMETERS_H
#define COBC_LOG_PARAMETERS_H

#include <stdint.h>

#include <cobc/parameter/spp.h>

#ifndef UNITTEST
/**
 * Meta logger level setting (internal logging feature).
 *
 *  0: Off
 *  1: ERROR only
 *  2: WARN and higher level
 *  3: INFO and higher level
 *  4: All
 */
#define MLOG_LEVEL 4
#else
/**
 * Meta logger level during unit tests (default: 0).
 */
#define MLOG_LEVEL 0
#endif
namespace cobc
{
namespace log
{
/**
 * Priority of the central thread
 */
static const uint8_t centralThreadPriority = 128;
/**
 * Stack size of the central thread
 */
static const size_t centralThreadStackSize = 4096;
/**
 * Period in microseconds of the central thread for copying the packets
 * from the application loggers to the central buffer.
 */
static const uint32_t centralThreadPeriodMicroseconds = 500000;
/**
 * Priority of the DataTransmitterBase thread
 */
static const uint8_t dataTransmitterBaseThreadPriority = 128;
/**
 * Stack size of the DataTransmitterBase thread
 */
static const size_t dataTransmitterBaseThreadStackSize = 4096;
/**
 * Priority of the ControllerBase thread
 */
static const uint8_t controllerBaseThreadPriority = 128;
/**
 * Stack size of the ControllerBase thread
 */
static const size_t controllerBaseThreadStackSize = 4096;
/**
 * Sets the usage of MessageCounter in the header.
 */
static const bool headerContainsMessageCounter = true;
/**
 * Sets the usage of ApplicationId in the header.
 */
static const bool headerContainsApplicationId = true;
/**
 * Sets the usage of ThreadId in the header.
 */
static const bool headerContainsThreadId = true;
/**
 * Sets the usage of TimeStamp in the header.
 */
static const bool headerContainsTimeStamp = true;
/**
 * Maximum packet size in bytes
 */
static const uint32_t maximumPacketSize =
        parameter::spp::maximumTelemetryApplicationDataLength;
/**
 * Width of the output of floating point values. Uses printf syntax.
 */
static const uint16_t decoderWidthFloat = 0;
/**
 * Precision of the output of floating point values. Uses printf syntax.
 */
static const uint16_t decoderPrecisionFloat = 3;
/**
 * Width of the output of floating point values (double precision). Uses
 * printf syntax.
 */
static const uint16_t decoderWidthDouble = 0;
/**
 * Precision of the output of floating point values (double precision). Uses
 * printf syntax.
 */
static const uint16_t decoderPrecisionDouble = 6;

} // namespace log
} // namespace cobc

#endif // COBC_LOG_PARAMETERS_H
```

A.7 FUTURE USE CASE: NEW MONITORING INFRASTRUCTURE FOR LAUNCHERS

As already introduced in the OUTLOOK Section 22.3, one of the possibilities for a new field of application of the Unified Monitoring methodology and the Monitoring F/W could be within the avionics system of current and future launchers by extending it with extensive self-configuration mechanisms. The more detailed description within this appendix section was mainly taken from the work description of the EU project *MaMMoTH-Up* ("Massively extended Modular Monitoring for Upper Stages") which has just started at the time writing this thesis [16].
Monitoring in current launcher avionic technology – like in most of todays S/Cs – has two aspects:

- internal monitoring of the C&DH system which is useful for debugging purposes and TM of the C&DH system itself, and
- monitoring events on the launcher system outside the C&DH system, e.g. from other actuators or sensors.

On the one hand, the complete data volume produced on board is too large to be sent to the groundstation as TM data. On the other hand, by selecting only pre-defined pieces of data (as it is done today), valuable information about unexpected events are lost. This makes optimization difficult and – in the extreme case – understanding failures is also hard.

SELF-CONFIGURING EXTENSIONS OF THE MONITORING F/W

Shifting the main Purpose of the Monitoring F/W from Debugging to Data Processing

While the use cases of the Monitoring F/W prototype and its succeeding flight version (cf. to Sections 16 and 17) mainly aimed at debugging purposes, the central role of the F/W within the launcher avionics will be to extensively collect and process onboard data. For this various techniques will be considered to effectively condense data and identify relevant expected as well as unexpected events.

The precise schedule for a launch precisely predicts which sensor data is relevant at what point during mission time. This information will be used to store a pre-defined schedule of expected events in the Monitoring F/W. In this way the F/W is able to detect deviations from these expectations on the fly and to collect the respective data. Similar concepts will be developed for internal monitoring of the C&DH system. After selecting relevant data for TM information, compression techniques will be applied. Even though limited, selected sensor data of previous launches is available on ground today for the evaluation of data compression schemes. Besides standard compression algorithms, well-known effective approaches for specific data processing (e.g., determination of maximum value, minimum value, average value, fast Fourier transform) will be considered. In case of unexpected events, selection and compression strategies may not be sufficient to send all data over the TM link with strictly limited bandwidth. Therefore, data will always be prioritized in real-time. In case of overloading a best-effort approach will deliver the data of highest priority at first.

The self-configuring extensions of the Monitoring F/W will use this prioritization strategies and the matching against expected behavior in order to select the most important data in real-time without intervention by an operator who would not be able to react during a launch.

TECHNICAL REALIZATION In order to design and implement the self-configuring extensions of the Monitoring F/W which are described above, additional requirements have to be taken into account. The requirements of the F/W prototype and flight version were driven by the needs for a medium size satellite for a research mission allowing for interaction with a human operator (cf. to the corresponding requirements listed in the Appendices A.4 and A.5). Even most of these requirements are still relevant and valid, they are dominated by the following additional requirements from the self-configuring extensions to be developed for this new field of application:

- providing real-time data processing and prioritization within the launcher's flight schedule,
- full functional compatibility with respect to the existing launcher Telemetry Data Acquisition Unit (french abbrev. *UCTM*), and
- the fulfillment of the functional requirements from the dependability and data processing extensions.

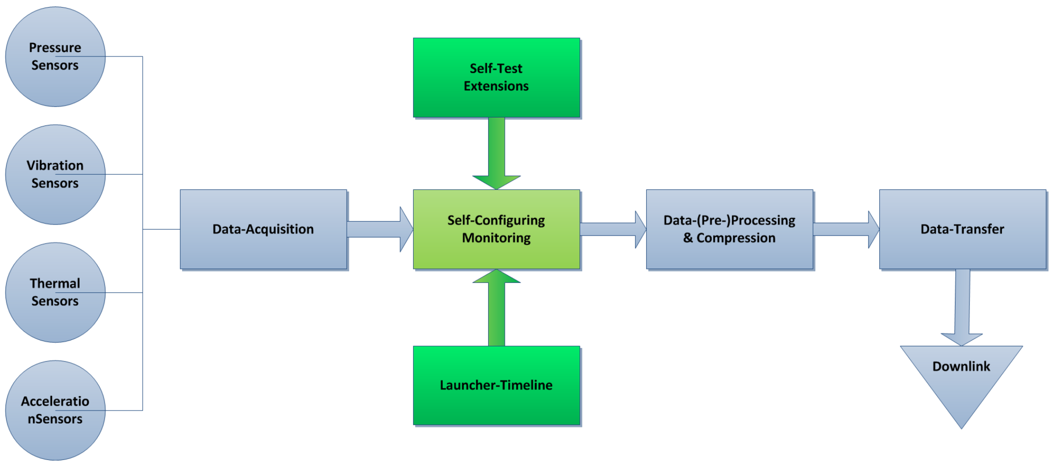


Figure A.5: MaMMoTH-Up Processing Chain

In order to fulfill these requirements new components will be gradually integrated: First, we will extend the Monitoring F/W with an

Introducing a new Monitoring Processing Chain. . .

extensive self-configuring mechanism. With this extension we will be able to selectively observe, pre-process, and compress sensor data. The F/W will be triggered by intelligent self-testing capabilities integral to the system and re-focus attention in case of unexpected events in order to achieve a currently unavailable observability into the system. Because of the tight timeline of a launcher this kind of re-focusing would not be possible from a remote operator due to high latencies.
Second, the pre-definition of the launcher's flight schedule will trigger the Monitoring F/W on fixed time points (e.g. jettisoning of upper stage or payload). By doing this, we will be able to monitor relevant information with a high fidelity at the time when they are supposed to occur during the mission. If storage capacity becomes low, eventually lowering the resolution and frequency of the monitoring information for some other parts of the system may be necessary. At the end, all gathered monitoring data will be pre-processed and compressed with suitable compression algorithms. Figure A.5 depicts an overview of the described processing chain.

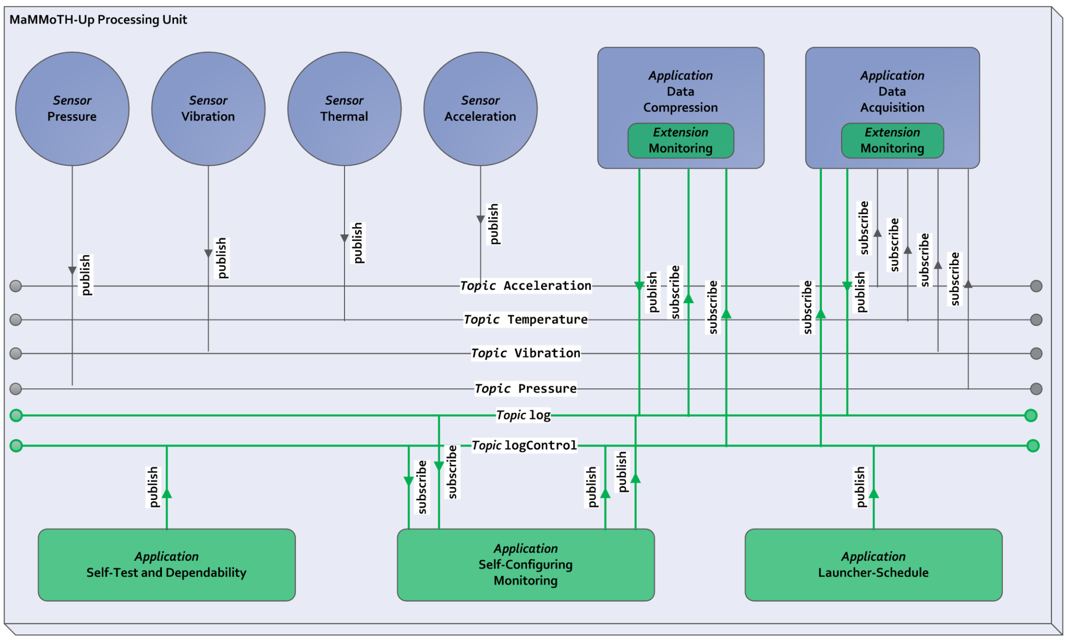


Figure A.6: MaMMoTH-Up Data Exchange using Middleware Topics

. . . and it's Realization.

In order to realize this processing chain, special emphasis has to be put on the data exchange mechanisms. As we want to keep the data transfer flexible, we propose the utilization of the middleware which is already the basis for the Monitoring F/W prototype (by running on top if the RODOS RTOS, cf. to Section 5.2.1.2) and also parts of the F/W flight version (by making use of the Eu:CROPIS/CompactSat-libCOBC library, cf. to Section 17.1.2). In either way, applications and encapsulated devices can publish information to so-called *topics*, to which other S/W components can subscribe upon. Like shown in Figure A.6, sensor interfaces may publish the data to their corresponding topics. The data acquisition application as a subscriber to these topics collects the data according to the configuration of the Monitoring F/W. Afterwards it publishes data to the *log*-topic which is used to exchange monitoring messages over the middleware. Hereafter the application responsible for data processing and compression can receive these monitoring messages for doing its data pre-processing task before downlink. Both applications can be controlled at runtime by the monitoring application through the *logControl*-topic which is used to configure the behavior of the subscribed applications. This kind of data transfer and control does not only allow to establish flexible communication paths: it also allows to leave the boundaries of the MaMMoTH-Up processing unit, if e.g. middleware gateways are added to the embedded system in order to communicate with the outside world or to distribute data processing over several units. This design will allow for far more flexibility and configurability in terms of data acquisition and processing in comparison to what is offered by the existing TM processing unit *UCTM* on board the upper stage.

BIBLIOGRAPHY

[1] Francisco Afonso. *Operating System Fault Tolerance Support for Real-Time Embedded Applications*. Phd thesis, Universidade do Minho, 2009.

[2] ASRA project team. SAVOIR General Recommendations for Spacecraft Monitoring and Control. Technical Report TEC-SW/12-539/JLT, ESA-ESTEC, May 2012. Issue 2, Revision 0.

[3] ASRA project team. SAVOIR Functional Reference Architecture. Technical Report TEC-SW/11-477/JLT, ESA-ESTEC, May 2012. Issue 3, Revision 0.

[4] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly, first edition edition, January 1999. ISBN: 1-56592-354-5, 191 pages.

[5] M.F. Barschke, K. Großekatthöfer, and S. Montenegro. Implementation of a Nanosatellite on-board Software based on Building-Blocks. In *4S Symposium Proceedings*, 2014.

[6] Scott Bell, David Kortenkamp, and Jack Zaientz. A distributed event architecture for space system comps. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 42:1–42:2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-665-6. doi: 10.1145/1619258.1619310.

[7] Ljerka Beus-Dukic. COTS Real-Time Operating Systems in Space. *The Safety-Critical Systems Club Newsletter*, 10(3):pp. 11–14, May 2011.

[8] Klaus Brieß and Hakan Kayal. *Grundkurs Satellitentechnik*. TU Berlin, Institut für Luft- und Raumfahrt, 2011.

[9] CCSDS. *Mission Operations Services Concept*, volume CCSDS 520.0-G-3. Washington, DC, USA, 3 edition, December 2010. Green Book (Informational Report).

[10] Caitlyn M. Cooke. Implementation of a Real-Time Operating System on a Small Satellite Platform. *Space Grant Undergraduate Research Symposium*, 2012. URL `http://spacegrant.colorado.edu/statewideprograms/research-symposium/`.

[11] Lockheed Martin Corporation. Joint Strike Fighter Air Vehicle C++ Coding Standards, December 2005. URL `http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards_Rev_C.doc`.

[12] Frank Dannemann. Towards Unified Monitoring of Spacecrafts. In *65th International Astronautical Congress, Toronto, Canada*. International Astronautical Federation, September 2014. URL `http://elib.dlr.de/90976/`. Paper IAC-14.D1.1.11.

[13] Frank Dannemann and Fabian Greif. Software Platform of the DLR Compact Satellite Series. In *Proceedings of 4S Symposium*, 4S Symposium, 2014. URL `http://elib.dlr.de/89344/`.

[14] Frank Dannemann and Sergio Montenegro. Embedded Logging Framework for Spacecrafts. In *DASIA 2013*, ESA Special Publication, 2013. URL `http://elib.dlr.de/83042/`.

[15] J.H.M. Dassen and I.G. Sprinkhuizen-Kuyper. *Debugging C and C++ Code in a Unix environment*, chapter Debugging techniques. 1999. URL `http://oopweb.com/CPP/Documents/DebugCPP/Volume/techniques.html`.

[16] DLR, EADS Astrium Space Transportation GmbH, AAC Microtec, and Politecnico di Torino. Massively extended Modular Monitoring for Upper Stages (MaMMoTH-Up). Horizon 2020 Proposal (Number: SEP-210132615), March 2014.

[17] ECSS. Space project management - project phasing and planning, April 1996.

[18] ECSS. Space engineering - software, March 2009.

[19] Jens Eickhoff. *Onboard computers, onboard software and satellite operations: an introduction ; with 33 tables*. Springer Aerospace Technology. Springer, Heidelberg [u.a.], 2012. ISBN 3642251692 and 9783642251696. XVIII, 282 S. ; 235 mm x 155 mm : Ill., graph. Darst.

[20] ESA-ESTEC. Ecss-e-70-41a packet utilization standard, January 2003. Space engineering - Ground systems and operations - Telemetry and telecommand packet utilization.

[21] Ross Findlay, Olaf Eßmann, Jan Thimo Grundmann, Harald Hoffmann, Ekkehard Kührt, Gabriele Messina, Harald Michaelis, Stefano Mottola, Hartmut Müller, and Jakob Fromm Pedersen. A space-based mission to characterize the {IEO} population. *Acta Astronautica*, 90(1):33 – 40, 2013. ISSN 0094-5765. doi: http://dx.doi.org/10.1016/j.actaastro.2012.08.004.

[22] Stefan Foeckersperger, Klaus Lattner, Clemens Kaiser, Silke Eckert, Wolfgang Bärwald, Swen Ritzmann, Peter Mühlbauer, Michael Turk, and Philip Willemsen. Tet-1 - a german microsatellite for technology on-orbit verification. In *The 4S Symposium*, pages 1–11, Mai 2008. URL `http://elib.dlr.de/55134/`.

[23] Peter W. Fortescue and John P.W. Stark. *Spacecraft Systems Engineering*. Wiley, Chichester [u.a.], 3. ed edition, 2003. ISBN 0471619515 and 9780471619512 and 0470851023. XXVI, 678 S ; 25 cm : Ill., graph. Darst.

[24] Joachim Fuchs. Engineering Data in the MBSE context. In *ADCSS 2012*. ESA, October 2012.

[25] Aeroflex Gaisler. GRMON - Debug Monitor for LEON, 2013. URL `http://www.gaisler.com/index.php/products/debug-tools/grmon`.

[26] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Boston [u.a.], 36th print edition, 2008. ISBN 0201633612 and 9780201633610. XV, 395 S : Ill., graph. Darst.

[27] Maurizio Gavardoni, Michael Jones, Russell Poffenberger, and Miguel Conde. System monitor for diagnostic, calibration and system configuration. In *Proceedings of the International Test Conference on International Test Conference*, ITC '04, pages 1263–1268, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8581-0.

[28] Göksu, Murat. Entwurf und Implementierung einer zur Laufzeit konfigurierbaren Logging-Komponente für Satelliten. Master thesis, Universität Bremen, September 2014. URL `http://elib.dlr.de/91005/`.

[29] Fabian Greif. C&DH SW Design Definition File. internal doc-no. EC-CDH-RP-0056, rev. 1.1, January 2014.

[30] M. Grott, J. Knollenberg, F. Hänschke, E. Kessler, N. Müller, A. Maturilli, J. Helbert, and E. Kührt. The MASCOT Radiometer MARA for the Hayabusa 2 Mission. In *Lunar and Planetary Institute Science Conference Abstracts*, volume 44 of *Lunar and Planetary Institute Science Conference Abstracts*, page 1586, March 2013.

[31] Jan Thimo Grundmann, Robert Axmann, Volodymyr Baturkin, Martin Drobczyk, Ross Findlay, Ansgar Heidecker, Horst-Georg Lötzke, Harald Michaelis, Ekkehard Kührt, Matthias Lieder, Stefano Mottola, Martin Siemer, Peter Spietz, Gerhard Hahn, Sergio Montenegro, Anko Boerner, Gabriele Messina, Thomas Behnke, Matthias Tschentscher, Karsten Scheibe, and Volker Mertens. Small satellites for big science: the challenges of high-density design in the DLR Kompaktsatellit AsteroidFinder/SSB. In *COSPAR 2010*, Juli 2010. URL `http://elib.dlr.de/72581/`.

[32] J. Guo, D.C. Maessen, and E. Gill. Fractionated Spacecraft: The New Sprout In Distributed Space Systems. In *60. International As-*

tronautical Congress. International Astronautical Federation, October 2009. Paper IAC-09-D1.1.4.

[33] Brian Handley. Use Eclipse for embedded cross-development. *EE Times-Asia*, June 1-15, 2007.

[34] Ansgar Heidecker, Takahiro Kato, Olaf Maibaum, and Matthew Hölzel. Attitude Control System of the Eu:CROPIS Mission. In *65th International Astronautical Congress*. International Astronautical Federation, Oktober 2014. URL `http://elib.dlr.de/90977/`.

[35] G.J. Holzmann. Landing a spacecraft on mars. *Software, IEEE*, 30 (2):83–86, March 2013. ISSN 0740-7459. doi: 10.1109/MS.2013.32.

[36] IPC Technology Harmonisation Advisory Group. European Space Technology Harmonisation Technical Dossier - On-Board Software. Technical Note issue 4, revision 0, ESA, January 2014. Draft.

[37] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991. ISBN 978-0-471-50336-1.

[38] R. Jaumann, J.-P. Bibring, K.-H. Glassmeier, M. Grott, T.-M. Ho, S. Ulamec, N. Schmitz, H.-U. Auster, J. Biele, H. Kuninaka, T. Okada, M. Yoshikawa, S.-i. Watanabe, M. Fujimoto, and T. Spohn. A Mobile Asteroid Surface Scout (MASCOT) for the Hayabusa 2 Mission to 1999 JU3: The Scientific Approach. In *EGU General Assembly Conference Abstracts*, volume 15 of *EGU General Assembly Conference Abstracts*, page 2923, April 2013.

[39] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Real-time systems series. Springer, New York, NY [u.a.], 2. ed. edition, 2011. ISBN 1441982361 and 9781441982360. XVIII, 376 S. ; 24 cm : Ill., graph. Darst.

[40] Andreas Korff and Markus Schacher. *Modellierung von eingebetteten Systemen mit UML und SysML*. Spektrum, Akad. Verl., Heidelberg, 2008. ISBN 9783827416902. VIII, 295 S. : Ill., graph. Darst.

[41] Leslie Lamport. *Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.

[42] Norbert Lemke, Stefan Föckersperger, Silke Eckert, Hans Gronert-Marquardt, and Eckehard Lorenz. Mission Success of TET-1 for On-Orbit Verification. In *4S Symposium Proceedings*, 2014.

[43] Qing Li and Caroline Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003. ISBN 1578201241.

[44] Anders Lundgren and Lotta Frimanson. Mastering stack and heap for system reliability, August 2012.

[45] Olaf Maibaum. Betriebssysteme Vergleich. SHARC internal document no: SH-SCRV-TN-001_1-0, June 2012.

[46] N. Mandolesi and F. Villa. First/planck mission. In *Instrumentation and Measurement Technology Conference, 1999. IMTC/99. Proceedings of the 16th IEEE*, volume 2, pages 975–980 vol.2, 1999. doi: 10.1109/IMTC.1999.777007.

[47] John C. Mankins. Technology Readiness Levels: A White Paper. Technical report, NASA, Office of Space Access and Technology, Advanced Concepts Office, 1995. URL `http://www.hq.nasa.gov/office/codeq/trl/trl.pdf`.

[48] Ken Martin and Bill Hoffman. *Mastering CMake 4th Edition*. Kitware, Inc., USA, 4th edition, 2008. ISBN 1930934203, 9781930934207.

[49] MathWorks. *PolySpace Products for C++ 7 - User's Guide*, 2009.

[50] MathWorks. Static code analysis, 2014. URL `http://www.mathworks.de/discovery/static-code-analysis.html`.

[51] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFsr generators ii. *ACM Transactions on Modeling and Computer Simulation*, 4(3):254–266, 1994. ISSN 10493301. doi: 10.1145/189443.189445. cited By (since 1996)72.

[52] Ernst Messerschmid and Stefanos Fasoulas. *Raumfahrtsysteme: Eine Einführung mit Übungen und Lösungen*, volume 1. Springer Berlin Heidelberg, August 2000.

[53] MISRA Electrical Group. Guidelines for the Use of the C Language in Critical Systems (MISRA C), March 2013. ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF).

[54] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, first edition, June 2001. URL `http://www.advancedlinuxprogramming.com`.

[55] Sergio Montenegro. Ultra fast recovery. In *DASIA 2010*, 2010. 1st to 4th June 2010, Budapest (Hungary).

[56] Sergio Montenegro. Visionary data management system for nano satellites. In *9th IAA Symposium on Small Satellites for Earth Observation*, September 2013.

[57] Sergio Montenegro and Frank Dannemann. RODOS - Real Time Kernel Design for Dependability. In *DASIA 2009*, 2009.

[58] Sergio Montenegro and Frank Dannemann. The software architecture for TET and AsteroidFinder satellites. In *7th Symposium on Small Satellites for Earth Observation*, Berlin, 2009. DLR. URL `http://elib.dlr.de/91049/`.

[59] Sergio Montenegro and Frank Dannemann. Requirements-Driven Design of Small Satellites: TET and AsteroidFinder. In *7th Symposium on Small Satellites for Earth Observation*, Berlin, 2009. DLR. URL `http://elib.dlr.de/91044/`.

[60] Sergio Montenegro, Stefan Jähnichen, and Olaf Maibaum. Simulation-Based Testing of Embedded Software in Space Applications. In Huanye Hommel, editor, *Embedded Systems - Modelling, Technology and Applications*, pages 73–82, Berlin, 2006. Technische Universität Berlin, Institut für Technische Informatik und Mikroelektronik in Kooperation mit der Shanghai Jiao Tong University, Springer Netherland. ISBN 978-1-4020-4932-3. URL `http://elib.dlr.de/22470/`.

[61] Peter Mühlbauer, Jens Richter, Herbert Wüsten, Michael Turk, Philip Willemsen, Sven Müncheberg, and Stefan Föckersperger. Mission operation, ground segment and services for the german tet-1 microsatellite. In *4S-Symposium 2008*, Mai 2008. URL `http://elib.dlr.de/63560/`.

[62] Hartmut Müller. DLR-Compact Satellite - A DLR space technology and research platform, September 2013. URL `http://www.dlr.de/irs/desktopdefault.aspx/tabid-4399/7195_read-10807/`.

[63] Müller, Sven. Entwicklung eines Rahmenwerks zur Nachrichtenprotokollierung für das eingebettete Echzeitbetriebssystem RODOS. Diploma thesis, Universität Oldenburg, 2012. URL `http://elib.dlr.de/88788/`.

[64] Bernd Oestereich. *Developing Software with Uml: Object-Oriented Analysis and Design in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 020175603X.

[65] Office of Aviation Research and Development. Real-Time Scheduling Analysis. Technical report, National Technical Information Service (NTIS), Virginia 22161, 2005. URL `http://www.tc.faa.gov/its/worldpac/techrpt/ar05-27.pdf`.

[66] Mauro Pecchioli and Anthony Walsh. Objectives and concepts of the european ground systems common core (egs-cc). In *Simulation & EGSE Facilities for Space Programmes (SESP) Workshop*, 2012. URL `http://congrexprojects.com/2012-events/12c09/programme`.

[67] Mauro Pecchioli and Anthony Walsh. Overview of the technology candidates for the european ground systems common core (egs-cc). In *Simulation & EGSE Facilities for Space Programmes (SESP) Workshop*, 2012. URL `http://congrexprojects.com/2012-events/12c09/programme`.

[68] Nestor Peccia. The European Ground Systems - Common Core (EGS-CC) Initiative. Presentation, February 2012. URL `http://csse.usc.edu/GSAW/gsaw2012/agenda12.html`. Ground System Architectures Workshop (GSAW).

[69] Li Ping, Xiaoling Huang, and N. Phamdo. Zigzag codes and concatenated zigzag codes. *Information Theory, IEEE Transactions on*, 47(2):800–807, Feb 2001. ISSN 0018-9448. doi: 10.1109/18.910590.

[70] Robert L. Read. *How To Be A Programmer: A Comprehensive Summary*. CreateSpace, Paramount, CA, 2009. ISBN 1440439524, 9781440439520.

[71] Hubert Reile, Eckehard Lorenz, and Thomas Terzibaschian. The FireBird Mission - A Scientific Mission for Earth Observation and Hot Spot Detection. In Rainer Sandau, Hans-Peter Röser, and Arnoldo Valenzuela, editors, *9th. IAA Symposium on small satellites for Earth observation*, volume Digest of the 9th International Symposium of the International Academy of Astronautics of *Small Satellites for Earth Observation*. Wissenschaft und Technik Verlag, April 2013. URL `http://elib.dlr.de/83866/`.

[72] Andreas Schartel. Filter-Concept for Diagnostic Reports. Bachelor thesis, University of Würzburg, 2014. URL `http://elib.dlr.de/90396/`.

[73] Michael Schulze. A Highly Configurable Logging Framework In C++. June 2010. URL `http://drdobbs.com/cpp/225700666`. Published on Dr. Dobb's Online Journal (www.drdobbs.com).

[74] Felix Schwieger. DLR_School_Lab Bremen: Lageregelungsexperiment "FloatSat". Handbuch für die Bodenstation und den Satelliten, 2014. URL `http://elib.dlr.de/91054/`.

[75] Lance Self. Use of Data Mining on Satellite Data Bases for Knowledge Extraction. In Jim Etheredge and Bill Manaris, editors, *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Symposium (FLAIRS) Conference*. AAAI, 2000.

[76] Mohammed El Shobaki. *On-Chip Monitoring for Non-Intrusive Hardware/Software Observability*. PhD thesis, Uppsala University, 2004.

[77] David E. Simon. *An embedded software primer*. Addison Wesley Longman, Reading, Ma. [u.a.], 1999. ISBN 0-201-61569-X; 0-201-61653-X.

[78] T. Sreenuch, A. Tsourdos, and I. K. Jennions. Software framework for prototyping embedded integrated vehicle health management applications. *Journal of Aerospace Information Systems*, 11 (2):82–97, February 2014. doi: 10.2514/1.I010046.

[79] Damien Texier. How We Map The First Light - The Science Ground Segment of Planck. *ESA Bulletin*, (139):21–24, August 2009. URL `http://www.esa.int/About_Us/ESA_Publications/ESA_i_Bulletin_i_139_August_2009`.

[80] Henrik Theiling. Rodos LEON2 Port. Technical report, AbsInt Angewandte Informatik GmbH, 2010.

[81] Norbert Toth, Frank Dannemann, Montenegro Sergio, Walter Thomas, and Faisal Muhammad. Wireless Avionics for a Solar Sailer (GOSSAMER-1). In *DASIA 2012*, 2012. URL `http://elib.dlr.de/75610/`.

[82] Eric Verhulst, Raymond T. Boute, Jos Miguel Sampaio Faria, Bernhard H. C. Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS: Software Engineering for Reliable Embedded Systems*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 1441997350, 9781441997357.

[83] David Paul Waleczek. Portierung des Echtzeitbetriebssystems RODOS auf eine ARM7 Plattform. Master's thesis, Jadehochschule Wilhelmshaven/Oldenburg/Elsfleth, February 2010. Bachelor Thesis.

[84] Tim Weilkiens. *Systems engineering mit SysML/UML: Modellierung, Analyse, Design*. dpunkt.verl., Heidelberg, 1 edition, 2006. ISBN 389864409X and 978-389-86440-9-9. XV, 360 S. ; 24 cm : Ill., graph. Darst.

[85] M. Wiegand, G. Schmidt, and M. Hahn. Next generation avionics system for satellite application. In *DASIA 2003*, ESA Special Publication, 2003.

[86] L Wood. *Internetworking with satellite constellations*. PhD thesis, University of Surrey, June 2001. URL `http://epubs.surrey.ac.uk/704760/`.

[87] A. Zacchei, S. Fogliani, M. Maris, L. Popa, N. Lama, M. Türler, R. Rohlfs, N. Morisset, M. Malaspina, and F. Pasian. HouseKeeping and Science Telemetry: the case of Planck/LFI. In *47th Annual Meeting of the Italian Astronomical Society*, volume 3, pages

331–334, 2003. URL `http://sait.oat.ts.astro.it/MSAIS/3/PDF/331.pdf`.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

Final Version as of July 10, 2015 (`classicthesis` Final).

DECLARATION

Ich versichere, dass ich die von mir vorgelegte Dissertation selbständig angefertigt, die benutzten Quellen und Hilfsmittel vollständig angegeben und die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, in jedem Einzelfall als Entlehnung kenntlich gemacht habe; dass diese Dissertation noch keiner anderen Fakultät oder Universität zur Prüfung vorgelegen hat; dass sie – abgesehen von den auf Seite ix angegebenen Teilpublikationen – noch nicht veröffentlicht worden ist sowie, dass ich eine solche Veröffentlichung vor Abschluss des Promotionsverfahrens nicht vornehmen werde. Die Bestimmungen dieser Promotionsordnung sind mir bekannt. Die von mir vorgelegte Dissertation ist von Prof. Dr.-Ing. Sergio Montenegro betreut worden.

Würzburg, 2015

Frank Dannemann