SpaceTeamSat1
Dataflow Game Document

01.04.2023

Version 1.1

Revision History

Revision	Date	Author(s)	Description
0.1	28.05.2021	SpaceTeamSat1	First draft
1.0	10.08.2021	SpaceTeamSat1	Initial version
1.1	01.04.2023	Patrick Kappl	Reworked title page and improved citations

List of Acronyms

ACK	Acknowledgment (signal)
ANT	Antenna System
AX.25	A data link layer protocol
CAM	Camera Module
CDF	Concurrent Design Facility
CCSDS	Consultative Committee for Space Data Systems
CFDP	CCSDS File Delivery Protocol
COBC	Communication Module and On-board Computer
CSBI	CubeSat Bus Interface
DS	Deployment Switch
DT	Deployment Timer
EDU	Education Module
EPS	Electrical Power System
ESA	European Space Agency
FEC	Forward Error Correction
FRAM	Ferroelectric Random Access Memory
FW	Firmware
FX.25	Protocol extension to AX.25
GS	Ground Station
HW	Hardware
MCU	Microcontroller Unit
MPPT	Maximum Power Point Tracking
MTU	Maximum Transmission Unit
SC	Solar Cells
RAM	Random Access Memory
RBF	Remove-Before-Flight pin
RF	Radio Frequency
RTC	Real Time Clock
RODOS	Realtime Onboard Dependable Operating System
RTOS	Real-time Operating Sysem
SatNOGS	Satellite Networked Open Ground Station
SPI	Serial Peripheral Interface (bus)
SRAM	Static Random Access Memory
STS1	SpaceTeamSat1 – name of CubeSat mission and CubeSat platform
SW	Software
UI	User Interface
UCI	Umbilical Cord Interface

Contents

1 Introduction

This document describes the dataflow of the **CubeSat mission SpaceTeamSat1 (STS1)** of the TU Wien Space Team. We developed the dataflow architecture during a game which relates to the Concurrent Design Facility (CDF) utilized by the European Space Agency (ESA).[1] In the CDF setup of ESA the interplay between all subsystems is designed accordingly and any related dependencies are checked – during normal operation of the satellite and also considering exceptional operation modes. This allows to determine any errors which might occur during the design phase of the satellite. In this respect various operation scenarios, e.g., normal operation, unexpected results, power shortage, etc., are considered. Therefore, the hard- and software (HW and SW) design as well as the mechanical design are evaluated thoroughly. In our so-called Dataflow Game the deduced system architecture, as described in "CubeSat: SpaceTeamSat1, Preliminary Design I: System Architecture", is extended and discussed thoroughly. Furthermore, details of certain operation modes are discussed and defined. The goal of the Dataflow Game is to find any errors in the design at an early stage of development as well as to define the SW architecture from a high-level point of view. Moreover, it gets clear, which hardware components are mandatory. We started the game by setting up our initial system architecture in miro[2]. Assuming that all components on the CubeSat as well as the ground station (GS) are fully functional, we started to play typical operation scenarios of the CubeSat and deduced the HW and SW architecture accordingly. This includes the definition of RF commands, as well as corresponding HW components, which are necessary for transmitting and receiving data to and from the CubeSat as well as storing data on the CubeSat and on the ground. For the implementation of the game persons were assigned to each part of the communication chain between GS and CubeSat and acted as defined in our initial design approach. This triggered discussions on various design decisions and thus leading to a more detailed and specific SW and HW design.

In the future, this document will be extended with new insights. As the mission and design matures a more detailed definition for all subsystems will be available. Note that the Dataflow Game will be played frequently to deepen the knowledge of the CubeSat platform and its operation.

Starting with the ground segment in chapter 2, the RF commands that can be sent from the GS are listed to get an overview of the CubeSat's capabilities. Afterwards, the user interface (UI), which will be realized with COSMOS[3] and SatNOGS[4], is introduced as well as the primary GS, which transmits and receives data to and from the CubeSat. Additionally, the SatNOGS network will be used to downlink beacons and data from the CubeSat. Finally in chapter 2, the communication management with its protocols is explained from a high-level perspective. Chapter 3 describes the HW components of each subsystem, i.e., COBC, EPS and EDU. This information was defined in more depth during the Dataflow Game. The SW architecture is described in chapter 4. It is separated into the communication and on-board computer (COBC) part and into the education module (EDU) part, as both systems require separate treatment due to their complexity. In the last chapter, **??**, open points and major topics which need to be reworked are documented for further processing.

2 Ground Segment

The GS is used to communicate with the CubeSat while it is in orbit. Currently (28.05.2021), it is planed to have a single primary GS which is able to communicate – uplink and downlink – with the CubeSat, while we have multiple GSs which can receive beacon information or any other downlink data. The multiple GSs network is accessed by utilizing SatNOGS.[4] The primary GS is located in A-2340 Mödling and is operated by Dr. Lars Mehnen (OE3HMW). For clarification and clarity the complete ground infrastructure, which consists of the primary GS, the multiple GS network (SatNOGS) and relevant infrastructure (servers, clients, etc.) is denoted as ground segment. It is of high importance that the primary GS has built-in automation mechanisms to upload and download data, as it is required to execute these tasks autonomously and independently of physical user access time. Especially, the downlink data shall be stored efficiently for analysis. The whole dataflow needs to be reliable for successful communication and data storage. Moreover, the primary GS needs to detect and follow the CubeSat as soon as it is in range for the transmission of newly generated commands. This approach should allow us to be able to communicate automatically with the CubeSat as well as manually, by physically accessing the primary GS in person and transmitting commands to the CubeSat without the proposed automation mechanism. Furthermore, we are using the SatNOGS network, which allows us to gain access to multiple GSs around the globe for downlink. This enables receiving beacons and other downlinked data from the CubeSat STS1, after instructing it, via the primary GS, to send it, even if STS1 is not in communication range of our primary GS. From a high-level architecture point of view the complete ground segment consists of the user interface (UI), which runs on a computer located at Lars' home, the actual GS setup (incl. antennas and other supporting infrastructure), which is physically located and accessible at Lars' home in Mödling, and the communication management. An overview of the setup is illustrated in fig. 2.1.

The colored boxes in fig. 2.1 belong to the primary GS (see section 2.2), whereas COSMOS and SatNOGS are described in section 2.3 and section 2.4, respectively. The white boxes are not explicitly described in the following, as further details need to be clarified in future. However, a short description is given here:

1. Remote:
   This represents a protected and secured, via User ID and password, remote access from a computer to the TU Wien Space Team GS Server.
2. GS – Lars PC:
   The primary GS with its incorporated modules relevant for the CubeSat radio frequency (RF) communication. In this respect more details can be found in section 2.2.
3. Antenna + radio device:
   The antenna with its CubeSat tracking mechanism and the RF hardware unit.
4. Relais – Power down:
   Legally it is required to always be able to disable the RF station. Therefore, a relais which allows to power down the RF unit needs to be installed.
5. Handy:
   To trigger the "Relais – Power down" at any time a cell phone which is capable to receive text messages will be installed. This allows to deactivate the power of the radio unit.

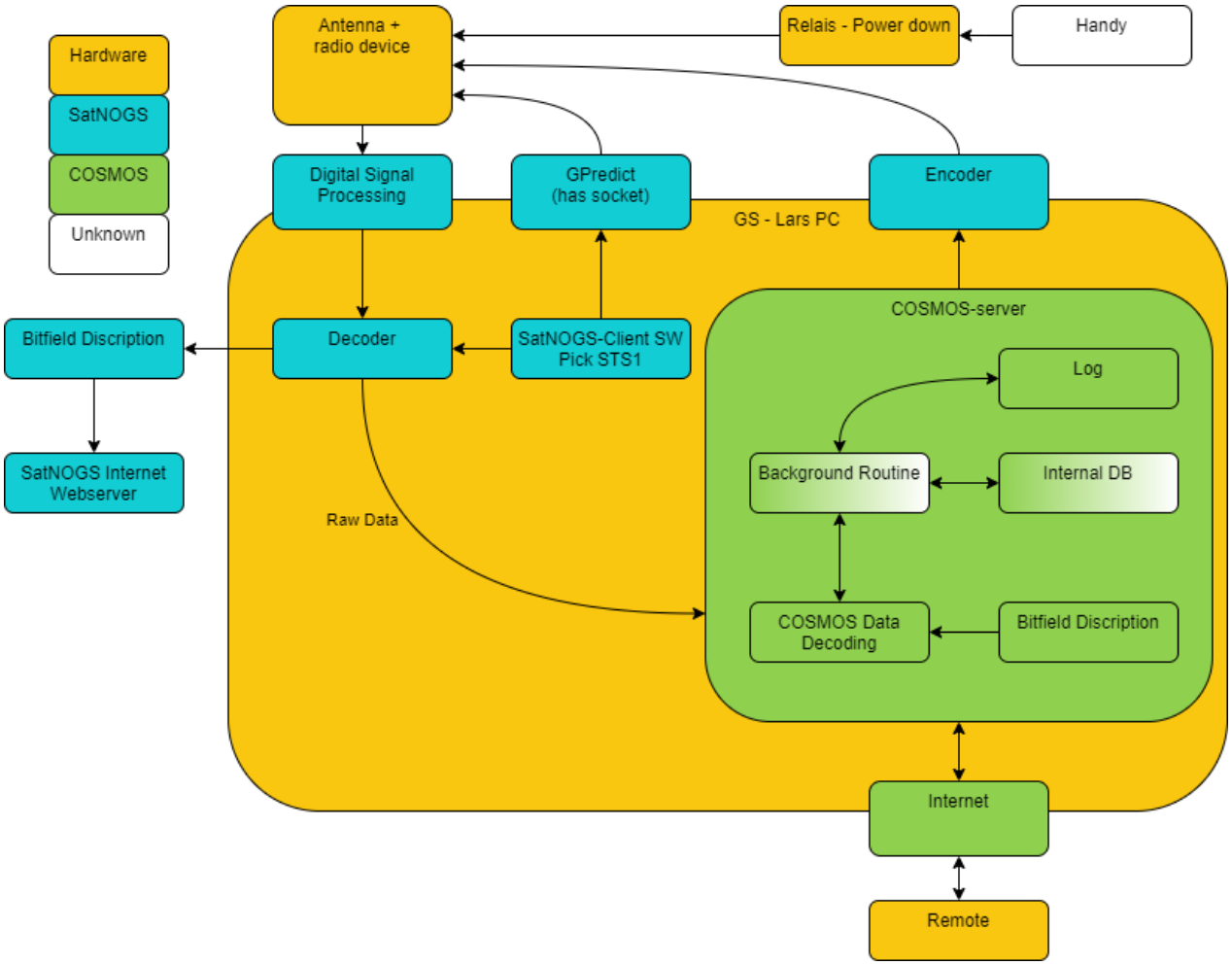

Figure 2.1: Overview of the high-level architecture of the ground segment. The boxed areas group components which belong to the same system. The individual components are described in the following sections throughout the document.

6. SatNOGS-Client SW Pick STS1:
   This is part of the SATNogs SW which runs on the Groundstation, it has STS1 selected and points the Antennas to it, via GPredict.

2.1 List of Commands

In the following all commands, which allow to communicate with the CubeSat, are listed and briefly described. These commands are separated according to their relevance for the dedicated subsystem. This includes general commands, EDU-related commands and firmware- (FW)-related commands. Note that justifications of the commands are not explicitly stated here.

Status: 28.07.2021

General Commands:

- Stop Antenna Deployment Mechanism
  Prevent triggering the antenna deployment mechanism after startup. This command will be sent

as soon as proper RF signals are received from the CubeSat to prevent continuously triggering the antenna deployment mechanism. The CubeSat answers with an ACK.

- Give latest X–Y beacons
  Request entries X to Y of the telemetry memory. The CubeSat answers with the requested data.
- Disable CubeSat TX
  Disable CubeSat RF transmission. This mechanism needs to be implemented to prevent continuous RF transmission of the CubeSat, which is required from a legal point of view. The CubeSat cannot send an answer.
- Enable CubeSat TX
  Enables CubeSat RF transmission. The CubeSat answers with an ACK.
- Set threshold
  Allows to set/change values of certain parameters, e.g., timings, RF parameters, the threshold value to activate the EDU, etc. The CubeSat answers with the parameter and the newly set value.
- Read threshold
  Allows to read values of certain parameters, e.g. timings, RF parameters, the threshold value to activate the EDU, etc. The CubeSat answers with the parameter and the corresponding value.
- Reset now
  Triggers a reset of the COBC. Can be used, e.g., to flash a newly uploaded FW image to the STM32F4 microcontroller unit (MCU) implemented on the COBC. The CubeSat cannot send an answer. However, the reset can be recognized by analyzing the reset counter contained in the beacon.

EDU Commands:

- General purpose upload
  Uploads an archive, which will be sent to the EDU module, more precisely, the RaspberryPi. This archive can contain anything we want to upload to the EDU, including python libraries, python files and images. Firstly, the corresponding file is selected in the UI. Afterwards the file is compressed and a checksum is calculated. Moreover, the generated package receives a program ID. The CubeSat answers whether the Python file was successfully written to the COBC file system or not.
- Queue bauen
  Uploads a Queue, which determines the execution order of the Python files. The generation of the Queue shall be implemented in the UI. A queue entry consists of the program ID, which can directly be assigned to a program name, the queue ID, the start time and the maximum execution time. The program and queue IDs are used to uniquely identify every execution of a program and the corresponding results file. The start time determines at what time the program is executed, as we are using absolute times it uses the epoch time. The CubeSat answers whether the queue was successfully written to the process memory or not.
- Return list of available results
  Requests a list of all available result files in the COBC file system respectively the relevant memories of the COBC. The CubeSat answers with the list.
- Return specific results
  Requests a certain results file, uniquely identified by the queue ID, stored in the COBC file system respectively the relevant memories of the COBC. The CubeSat answers with the results file.

- Delete file from COBC file system
  Deletes a certain file, denoted by the filename, stored in the COBC file system respectively the relevant memories of the COBC. The CubeSat answers with a confirmation after successfully deleting the file.
- (Return list of files in COBC file system)
  Requests a list of all files stored in the COBC file system respectively the relevant memories of the COBC. The CubeSat answers with the corresponding list. Note that, currently it is not clear if this command is mandatory.
- (Return list of programs on RasPi)
  Requests a list of all Python files stored on the EDU module respectively on the RaspberryPi. This can be done be utilizing a Python script on the Raspberry Pi itself. The CubeSat answers with the corresponding list. Note that currently it is not clear if this command is mandatory.

Firmware Commands: A more detailed explanation of the whole FW update process, which uses these commands, is given in section 4.1.1. In general, the FW memory consists of two separate memory benches as well as two variables, namely `activeFwImage` and `backupFwImage`. Those variables are required to boot the COBC. Further details regarding the boot up are illustrated in the corresponding flowchart in fig. 4.2.

- FW upload
  Allows to upload a new COBC FW image. In our self-developed COSMOS-based UI the image and the relevant FW memory are selected. It shall also prompt the user and ask for permission. Afterwards, the checksum over the whole image is calculated and appended. The CubeSat answers whether the new FW is stored successfully on the corresponding FW memory or not.
- Set active FW
  This command contains the following information: The dedicated memory bench and a corresponding checksum. This command leads to the following changes in the system which are additionally illustrated in **??**. First, the received checksum is compared with the checksum in the dedicated memory bench. If they match, the backupFWImage gets set to the current activeFWImage. Afterwards, the activeFWImage gets set to the received and corresponding FW image. Finally, a reset is triggered, leading to a reboot of the bootloader (fig. 4.2), which loads the new FW to the STM32F4. This command does not sent a specific answer, however it is possible to check if the switch of the FW worked by analyzing the beacons, as they contain the active FW.
- Set backup FW
  Sets which of the two FW memory benches contains the backup FW. Again, the checksum of the FW image is calculated over the image as a safeguard to prevent accidentally switching to the wrong backup FW. Only if the sent checksum and the one from the desired image in the chosen FW memory are equal, the backup FW is changed. The CubeSat answers which FW memory the backup FW is set to.
- Check FW integrity
  Checks the integrity of the FW image in the chosen FW memory. This is done by calculating the checksum of the image and comparing it with the checksum stored within the corresponding memory. The CubeSat answers whether the integrity is given or not.

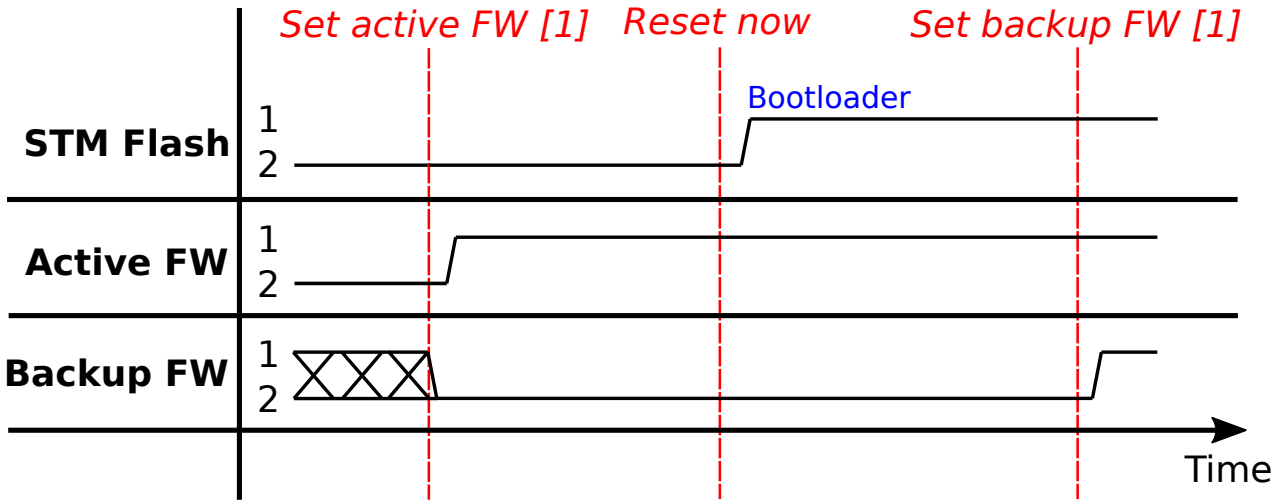

Figure 2.2: Triggered mechanisms after transmitting command “Set active/backup FW”.

2.2 Primary Ground Station

The primary GS of Lars Mehnen is used for general communication activities and in particular up- and downlink communication with the CubeSat. Additionally, the SatNOGS network will be used for downlink activities. To be more precise this includes beacons and downlinked payload data from the CubeSat. The TU Wien Space Team has full access to the primary GS and will be legally allowed to use it. Lars Mehnen is a long time supporter of the TU Wien Space Team and has a track of successful communication links to other CubeSats in space. The control SW for the GS is on a physical computer located directly at the primary GS. Preventing complexity and increasing usability, the SW architecture of the ground segment is divided into different sections, as illustrated in fig. 2.1. We are currently working on the CubeSat architecture and therefore have not defined the ground station architecture in depth yet.

SatNOGS,[4] a SW library which is installed on the primary GS is used for the package management, which includes encoding of the to-be-sent packages as well as decoding of received packages. Moreover, SatNOGS enables us to constantly receive downlink data, as it consists of a worldwide network of GSs. Further details are given in section 2.4.

COSMOS[3] is used for the data management and user interaction, which includes command storage and handling, logging of communication approaches as well as the handling of received and to-be-sent packages. COSMOS is recommended by ESA and many CubeSat missions already work with this framework. Further details regarding COMOS are given in section 2.3. Currently, we are neither planning to use any acknowledge system nor to use a retry counter for sending commands from the GS to STS1. During development we will see how well this approach works and if we need to adapt. In addition, Python scripts will be used, as not all topics can be covered by COSMOS or SatNOGS. Especially, for tasks like data analysis, etc., these Python scripts will be utilized. These topics are discussed in section 2.4.

2.3 User Interface and COSMOS

The UI allows to create the commands, described in section 2.1, and to manage the queue of commands which are currently waiting to be transmitted at the next flyby of STS1. Note that all corresponding data is stored on the GS database, physically located at the primary GS. For the realization of the UI and other relevant parts in this communication chain, COSMOS[3] will be used. Using COSMOS

allows to transfer the whole ground segment infrastructure to other GSs, without much adaptions. The high-level architecture of the GS is shown in fig. 2.1, whereas the relevant parts regarding COSMOS are described in the following.

COSMOS server

For data management and automatic communication we use COSMOS. Firstly, it is used to store all commands we want to send to the CubeSat and all the packages we receive (COSMOS Data Decoding, Internal DB). Secondly, it is used to run the automation mechanism which manages communication, meaning it checks at what time we want to send what command and if we are able to do so (Background Routine). Furthermore, all sent messages are stored in a log file for legal and tracking reasons (Log). The COSMOS server is connected to the Internet, meaning that a second COSMOS SW can connect to it. This allows us to access the GS remotely. For some of the parts we are not yet sure if COSMOS can cover these topics. For required software solutions which cannot be covered by COSMOS itself, other programs and libraries are necessary. In this respect, details can be found in section 2.4.

2.4 SatNOGS

SatNOGS is used as a client SW for several use cases. Firstly, it is used for predicting and pointing of the antenna to the CubeSat (SatNOGS Client SW Pick STS1, GPredict). Secondly, it is used to encode the outgoing packages (Encoder) and decode the incoming packages from the RF part of the GS (Digital Signal Processing, Decoder). Finally, it is used to publish the incoming transmissions to the SatNOGS website (Bitfield Description, SatNOGS). Publishing also includes packages where the FEC failed, allowing us to analyse even error-afflicted downlink data and to interpret them manually.[4]

If we want to use multiple receiving GSs with SatNOGS we will use a Python script to fetch all the received packages from the SatNOGS website and provide them to COSMOS in a similar manner as the SatNOGS code. From a COSMOS perspective this requires the same implementation as the SatNOGS interface.

An additional use case of SatNOGS is that it can be integrated into the educational objective of the STS1 mission, by giving students the task to build their own SatNOGS GS. This could help us to increase our downlink capabilities (see chapter 2) as well as give students a nice hands-on experience on a project which then works even without STS1 in orbit. Moreover, it allows students to get familiar with space technologies by independently analyzing downlinked data of other CubeSats.

Supporting SW

Some of the SW-related program parts are not yet or only partly defined (Bitfield Description, Background Routine, Logging, Internal DB) at the moment. Moreover, actual implementations are under investigation. We are currently in the assessment of COSMOS and SatNOGS. If their capabilities do not match our needs, we might use Python for missing parts or interfaces.

2.5 Communication Management

The communication management is a demon (a program running in the background) running on the primary GS which is connected to the GS database. The GS database includes the queue of commands that is currently waiting for the next flyby of STS1. The demon manages the communication protocol

Layer 1: Physical	RF + Modulation + Channel Coding
Layer 2: Data Layer 3: Network Layer 4: Transport Layer 5: Session Layer 6: Presentation	CCSDS
Layer 7: Application	Data + Timestamp + UniquePackageID

Figure 2.3: OSI model for the communication stack of STS1. This model will be updated as soon as more details are defined. (Status: 31.07.2021)

(Acknowledges) for each data package sent to and received from STS1. It is also responsible for timing the transmission of uplink packets to the CubeSat.

The primary GS is always in a receive state, where it waits for incoming transmissions, once a beacon is received from STS1 it switches to a "possible TX" state. In this state it is then possible to send packages to the CubeSat. This concept means, that during normal operation a beacon is always required for the GS to start sending packages. After a given time, about twenty minutes or after not receiving a correct answer from STS1, the GS exits the "possible TX" state and returns to the receive state.

Data is transmitted in packages. Both communication partners have a transmission queue that holds completely built packages in RAM waiting to be sent. As the maximum transmission unit (MTU) per package will be rather small (in the order of a few kilobytes), there should be sufficient RAM to store enough packages for the queue to not run dry (on a real time system, at least two full packages have to be allocated).

Both lower-level protocols that are currently considered – The CCSDS protocol stack and AX.25 – provide a form of virtual channel separation. This is used both for separation of unrelated data (e.g. commands and a parallel, long-running file transfer) and for prioritising different data channels for the transmission queue (beacon packages are more important than command packages which are more important than data transfers).[5] To always be able to communicate with the CubeSat, a small receive gap is left after each beacon even if there is data in the transmission queue. This allows us to interrupt larger data streams regularly for important and crucial uplink data. In the case that this simple system shows evident insufficiency during testing, a more complex transmission timeslot management has to be implemented, allowing to uplink important data at any time. Due to these two constraints, the upper protocol layers need to be able to handle very long round-trip times, if an answer is expected. This also ensures that longer transfers can continue over multiple flyovers and if more important transfers have to interrupt the long transfer.

For a more didactic insight into the communication concept of the CubeSat, the OSI model, as proposed by CCSDS is used.[5] fig. 2.3 shows the simplified OSI model, which is considered in this context. Note that, certain layers are either not well-defined at moment as further progress in the development is necessary or not clearly separated by the CCSDS protocol stack. Nevertheless, it gives an insight on our thoughts how it will be implemented.

As the requirements for ACKs vary between different applications, ACKs are not handled by the lower protocol levels. Beacons do not require or expect an ACK as they are mainly used for live telemetry data and as a heartbeat signal from the CubeSat. Nevertheless, most of the commands transmitted to the CubeSat receive a response. In this respect, details can be found in section 2.1.

The corresponding response is either a simple ACK or a full response packet, containing data. Most notably, data transfers require the most complex ACK mechanism, as sending an ACK after every packet would result in a huge overhead and thus leading to a reduced bandwidth. As the file transfer protocol is not designed yet, the ACK mechanism will probably work on a basis of blocks, where the CubeSat sends a whole block of the dedicated file and only expects an ACK at the end of the block, which answers which blocks (if any) needs to be resent. A system like this minimises overhead while being relatively simple to implement and allowing us to use receive-only GSs like the SatNOGS network to increase our downlink coverage, thus lowering the overall time required to downlink large datasets.

Another design requirement to be able to take advantage of multiple receiving GSs, is that the protocol is required to uniquely identify a package even if it is received by multiple GSs with inaccurate clocks. This is accomplished by adding a counter field to each sent package. The counter is incremented with each package and rolls over to 0 when reaching the highest possible value. This, in combination with the possible slightly inaccurate receive-timestamp added by the GS is enough to make sure that packages received by multiple GSs can be identified as the same package (thus only processing it once and not multiple times).

3 CubeSat Hardware

The following chapter describes the HW of the involved subsystems of the CubeSat platform. All stated aspects were derived from the Dataflow Game, as discussions and design decisions lead to a detailed insight into the required HW components. Moreover, certain SW architecture decisions directly affect the HW. Note that the block schematics of the individual subsystems shown in the document "CubeSat: SpaceTeamSat1, Preliminary Design I: System Architecture" are still valid. However, more details regarding the design can be found here. As the name of this document already depicts, only dataflow-relevant topics are described, e.g. extraction of beacon data, storage of data, processing of data, etc.

3.1 COBC

In the scope of this document special attention is given to the RF communication and the memory management of the COBC, as these two systems have the highest impact on the dataflow. The deduced statements are merely the consequences which established during discussions of the SW architecture, in the process of the Dataflow Game.

In general, the COBC needs to evaluate, process and store all incoming and outgoing data packets from, respectively to the RF communication module. In consequence, its main capability is to organize all data stored on the CubeSat and manage the memory of the CubeSat. Therefore, the focus in this document lies on the storage and processing of relevant data. The COBC provides the timer for an internal watchdog, which is taking care of resetting the COBC if no communication with the GS succeeds section 4.1.1. Note that from a physical point of view no dedicated memory HW cells are defined yet, however, the logical description of the memory is deduced and presented here. In general, the following blocks (Section 3.1.1 to 3.1.5) build the logical memory cells of the CubeSat platform. Figure 3.1 gives an overview of the relevant memories involved.

3.1.1 STM32 internal

As the STM32F411RE is used, the mentioning of "STM32" equals to "STM32F411RE" in this section. The corresponding STM32's internal memory consists of one Flash (512 Kbyte) and one SRAM (128 Kbyte) which do not have a specific usecase at the moment. The flash stores the FW, the bootcode and three non-volatile variables required for the bootcode. These variables are the "notOkCounter", which counts the SW resets, indicating faulty FW images. Note that it will be reset after a successful communication with the GS. The other two additional variables, are dedicated for the "activeFWImage" and the "backupFWImage". Both FW-image variables are crucial to run the bootloader. Furthermore, the flash can be split up into several logical subsections to prevent unauthorised overwriting. The STM32 also provides three USARTs which are capable to be used to communicate with our EDU module. Also, the STM32 runs at 100 MHz on a 32-bit Cortex-M4 CPU, which additionally includes a memory protection unit. Further information are available at the STM32's datasheet (https://www.st.com/resource/en/datasheet/stm32f411re.pdf).

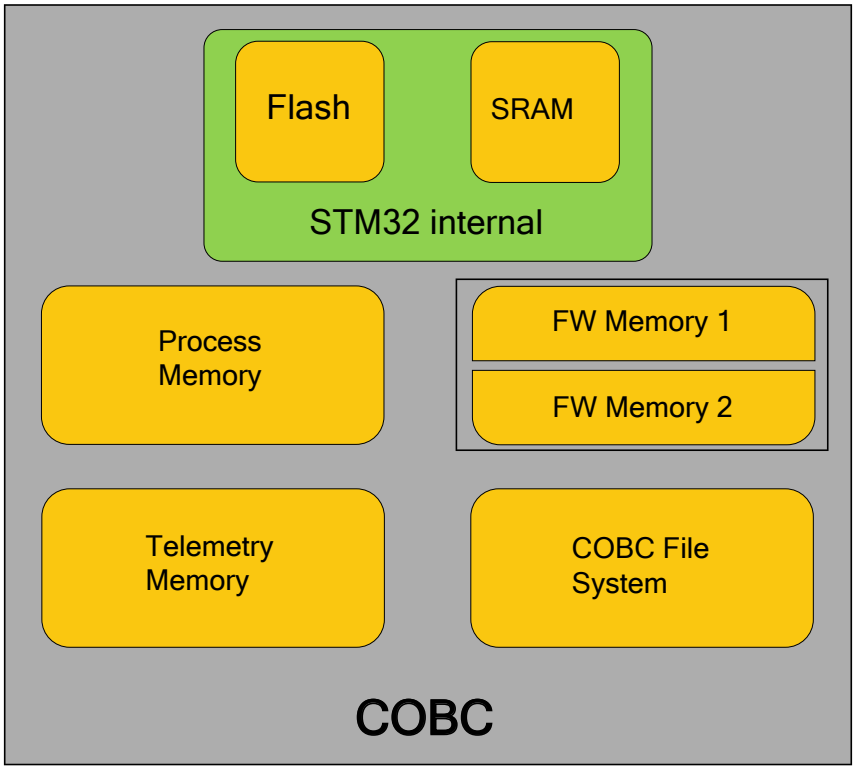

Figure 3.1: Overview of the COBC architecture and its relevant memories. At the moment no dedicated memories are selected nor details are available. (Status: 31.07.2021)

3.1.2 Process Memory

The process memory includes the "process queue" and the "process status and history" which are stored in a persistent memory like a flash or FRAM section 4.1.7 section 4.1.7. The "process status and history" is stored via a ring buffer. This allows us to store as many entries from the past as possible without a file management system. Furthermore we do not expect to download results of a newer executed program and let the older results stored on the CubeSat. The "process queue" contains entries regarding the execution of Python files on the EDU module. The Process Memory is responsible for logging the status and history of all Python files which were sent to the EDU module.

3.1.3 COBC File System

The COBC file system includes all programs which get uploaded and all the results generated by the EDU which were already sent to the COBC. These data are required to be stored in a persistent memory, e.g. a flash memory or FRAM. It is important that the COBC File System merely stores compressed raw data without any RF communication protocol data header. This reduces overhead and keeps the communication part separated and therefore more distinguishable. Additionally, check sums are attached to any file to ensure integrity.

3.1.4 Firmware Memory

The FW memory includes two memory regions which includes one FW image per memory region. Additionally, its check sum is calculated. Those are again stored in a persistent memory, e.g. a flash memory or FRAM. The actual FW images are distinguished between the active FW image and the spare, backup FW image, which is used for FW updates. During such updates the actual active FW serves as the backup image in case the updated FW version turns out to be corrupted. Details regarding the update process are documented in section 4.1.1.

3.1.5 Telemetry Memory

The Telemetry Memory contains the telemetry-related data relevant for the beacon. These are stored in a persistent memory, e.g. a flash memory or FRAM. The dataflow of relevant sensors and subsystems to the Telemetry Memory needs to be directed by the STM32. Currently we assume that we want to use a 1 GBit Flash ring buffer. The size is derived from the assumption that we have a 1 kbit beacon which would lead to about 11 MByte of telemetry data for one month. As we want to store at least two month of data on the flash the flash needs to be larger than 22 MByte. A ring buffer is used due to its simplicity and because we assume that we want to have all telemetry data, we will download the older data before downloading the newer ones, therefore the older data can be deleted first.

3.2 EPS

From a dataflow point of view the EPS only provides housekeeping data passively. This means that the COBC accesses the data on the EPS by HW logic. Note that, currently (Status: 11.07.2021) no dedicated bus is defined yet. The COBC stores data on the "Telemetry Memory" (see section 3.1.5). These data will then be used for the generation of the beacon or upon a request command. At the moment (Status: 28.05.2021) no dedicated parameters are defined. However, the following are considered:

- Battery temperature

- Battery voltage
- Battery current minimum
- Battery current maximum
- Battery current average
- Solar cell voltage
- Solar cell current minimum
- Solar cell current maximum
- Solar cell current average
- Battery OK

3.3 EDU

The EDU consists of three parts: The interface to the COBC (Section 3.3.1), the RaspberryPi HW (Section 3.3.2) and the sensors (Section 3.3.3).

3.3.1 EDU Interfaces

These interfaces are included in the HW connector, which connects all subsystem PCBs with each other and is also called CubeSat Bus Interface (CSBI). The interface consists of all the connections the EDU subsystem has with other subsystem, namely the communication connection to the COBC (most likely realized via SPI), the heartbeat connection to the COBC, the startup pin to the COBC, the info pin to the COBC, for further information regarding the heartbeat see section 4.1.7. Obviously, the EDU module needs to be powered by the EPS. The communication establishment and initialization is described in section 4.1.7.

3.3.2 EDU RaspberryPi

The RaspberryPi HW consists of a RaspberryPi compute module[6] and the power supply management. The power supply management consists of a logical switch which turns the power supply of the RaspberryPi off in case the startup pin, which is connected to the COBC, is not pulled up. Note that a certain voltage regulation is necessary to operate the RaspberryPi.

3.3.3 EDU Sensors

The EDU sensor board consist of the sensors and its own power supply, providing corresponding voltages for the individual sensors. The final set of sensors is not yet fully defined but the CubeSat will have at least the following sensors: Magnetometer, thermometer, accelerometer, gyroscope, dosimeter, voltmeter(which measures the voltage of the solar cells), strain gauge, etc.

4 CubeSat Software Architecture

This chapter describes the SW architecture of the CubeSat platform and was mainly derived during the Dataflow Game. Discussions and "simulations" of real dataflow scenarios, e.g., upload a Python file to read out a thermal sensor, or update the FW on the STM32, allowed us to develop the COBC SW (=FW) in a top-down approach. For the implementation of the SW architecture, RODOS[7] (Realtime Onboard Dependable Operating System) will be used – an operating system for satellites developed by the University of Würzburg. It uses priority controlled preemptive scheduling, i.e., every thread gets a priority and executing threads can be interrupted and resumed at a later time. To pass information between threads the publish–subscribe pattern is used where user defined data is published via so called topics and other threads can subscribe to them to get updates for it. RODOS also provides a real time middleware which makes it easy to develop modular software on top of it. Applications and modules can be implemented independently and interchanged easily since they are encapsulated in so called building blocks with well defined interfaces. The main design philosophy behind RODOS is to keep everything as simple nevertheless robust as possible which makes it ideal for the COBC SW.

In the following all tasks and threads managed and operated by RODOS are described. They define the whole SW architecture, i.e., what the SW can and should do. Starting with the COBC in section 4.1 a detailed insight on all corresponding tasks is given. As mentioned previously this includes the FW and bootloader management as well as the normal operation mode of the CubeSat platform, including initialization, antenna deployment and the management of telemetry data. The second part which is directly affected by the SW architecture is the EDU operation, which is described in more detail in section 4.2. Note that, the COBC always acts as the Master on the CubeSat platform and that the EDU is always in slave mode. This prevents common issues which occur in multi-master systems, like "arguing" which master is responsible etc.

4.1 COBC

The COBC SW architecture is the heart and brain of the SW concept, as all subsystems directly communicate with the COBC. Naturally it is the master in the presented concept. Note that RODOS is actually just a framework of libraries, so everything related to the operating system is compiled into the FW and can be changed with updates of the FW. This allows us to, e.g., modify topics during the operation of the CubeSat.

4.1.1 Firmware and Bootloader

As briefly mentioned before, the FW is a synonym for the whole COBC SW. It controls and manages most everything on board STS1. The bootloader is a small program also located on the STM32 that, as the name suggests, is executed first when booting and is able to flash a new FW on the microcontroller. As mentioned later, the COBC FW can be updated while STS1 is in orbit. This, however, does not apply to the bootloader, meaning that if it gets corrupted by, e.g., radiation, there is no way to repair it. Therefore the size of the bootloader must be as small as possible to reduce the risk of damaging it.

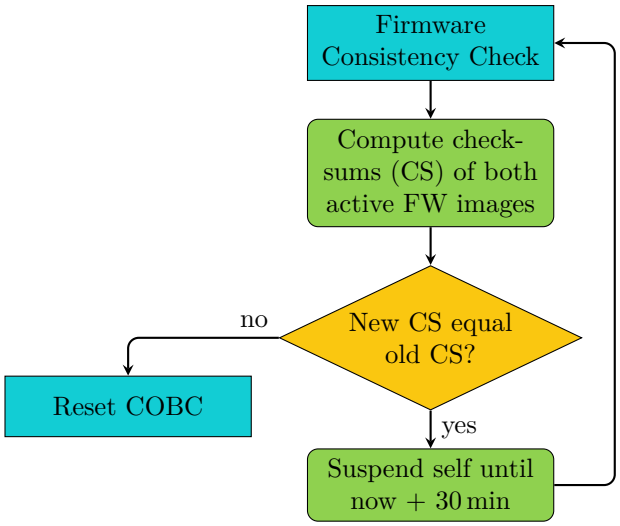

Figure 4.1: Flowchart of the thread, performing the FW consistency check. If the checksums differ, the COBC is reset since the bootloader (see fig. 4.2) also checks the FW images and repairs the faulty one.

To ensure a reliable operation of the CubeSat, we must guarantee that the FW does not get corrupted over time, e.g., by radiation. For this purpose we employ a mechanism of self-repair. Each FW image always includes a checksum. Additionally a copy of the active image, running on the STM32, is stored in one of the two external FW memories (see Section 3.1.4). In regular intervals, e.g., every 30 min, the checksums for both FW images (STM32 and external) are recalculated and compared to the old checksums, stored with them. If one pair of checksums differ, the image has been corrupted and the COBC is reset. This triggers the bootloader, which also checks the FW images, to overwrite the faulty one with the one that is still intact. Usually we do not consider double errors and do not worry about both checksums having changed, but this could lead to a reset loop: FW consistency check fails → reset → bootloader cannot repair anything because both FWs are corrupted → startup → FW consistency check fails. Therefore the FW consistency check is not executed right after booting but at least 15 min later. Figure 4.1 shows the flowchart of the thread that does this FW consistency check.

An additional feature of the COBC is that the FW can be updated even when the CubeSat is already in space. This is less crucial for a mission success since as much effort as possible should be put into making sure that the COBC FW works as intended and does not need to be updated while in orbit. Nevertheless, having the option for such an update is potentially very useful and handy, even though it adds quite a bit of complexity to the system. It requires a second external FW memory where the new image can be written to. Additionally, there is a variable that stores which external memory contains the active FW and another one that points to the location of a backup FW. Both of these variables are in a persistent memory. To update the COBC FW one has to go through the following procedure:

1. Upload a new FW image to the CubeSat and check its integrity.
2. Set the backup FW to the current FW.
3. Set the active FW to the new FW.
4. Reset the STM32 causing the bootloader to flash the new FW to its internal flash.
5. After verifying that the new FW works as intended, set the backup FW to the active FW.

More details on the exact commands that are sent to the CubeSat can be found in sections 2.1 and 4.1.6.

To prevent an update with a faulty FW from rendering the COBC useless, a backup mechanism is implemented. During initialization (see section 4.1.2) the internal watchdog timer is set to a low interval of, e.g., 2 h. Only when the FW is able to communicate with the ground segment, i.e., receive a valid message, this interval will be increased to, e.g., 24 h and the watchdog gets reset. In case that a new FW does not work, the STM32 will be reset frequently by the external watchdog. If too many of those resets happen, the bootloader (see fig. 4.2) switches to the backup FW. It should be mentioned here that the FW integrity check does not look at the backup FW. This means that it is advisable to verify that the new FW works as intended and set the backup FW to the active one in a timely manner. Otherwise, the backup FW might get corrupted in the meantime.

4.1.2 Initialization

After every reset the COBC must be set into a valid initial state. Figure 4.3 gives a rough overview of how this is done. The period of the watchdog timer is set to 2 h and only gets increased to 24 h after a successful communication with a ground station (see section 4.1.5).

Since the COBC does not know about the post deployment timer (it is realized in hardware) the initialization scheme is always the same.

4.1.3 Antenna Deployment

Figure 4.4 shows the flowchart of the antenna deployment thread. It is assumed that the solar cells' power is enough to successfully deploy the antenna, i.e., it works even with dead batteries. The initial 5 min wait is to decrease the frequency of potential resets caused by triggering the antenna deployment mechanism while too little power is available. The time span was chosen long enough such that a few beacons can be sent before the potential reset but short enough to allow trying the deployment a few times while the CubeSat is in the sun. The variable `antennaShouldBeDeployed` is stored in persistent memory and initially set to `true`. Only via a command (see sections 2.1 and 4.1.6) can it be set to `false`.

4.1.4 Telemetry

CubeSat telemetry allows the discovery of the satellite's state from the ground station. A telemetry package is broadcast by the CubeSat automatically every 30 s. This automatic broadcast is also known as "beacon". Its occurrence can only be suppressed, by completely deactivating outgoing CubeSat traffic via the corresponding command "Disable CubeSat TX". Beacon compilation and broadcast is handled by the "Send Beacon" thread. If this thread is called, while a receive operation is taking place, the RX channel is immediately deactivated, thus the receive operation is aborted and the associated data package is lost. Independently of whether a receive is currently active or not, the beacon package is always put first in the TX queue, without deleting the contents of the queue. To prevent a starvation of the RX channel, a 5 s listen period follows, after each beacon transmission. During this period, the TX channel is turned off and only RX is active. A flowchart describing the "Send Beacon" thread can be seen in fig. 4.5. The contents of a beacon are listed in table 4.1.

In order to be able to fetch beacons that have not been received by the ground segment, the telemetry packages are saved persistently on the CubeSat in the Telemetry Memory (see section 3.1.5). These packages can be requested from the ground station via the command "Give latest X–Y beacons" (see section 2.1).

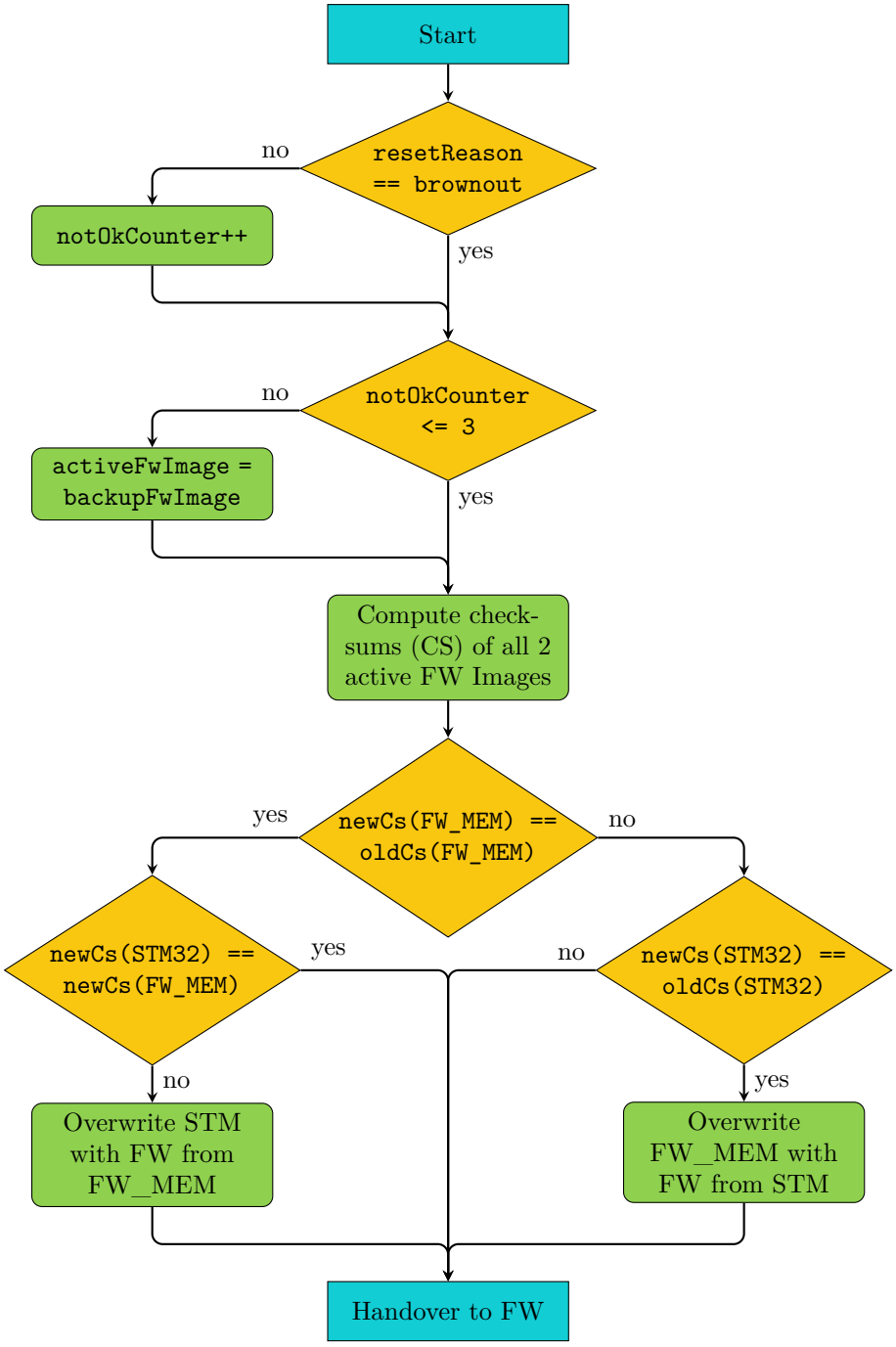

Figure 4.2: Flowchart of the bootloader of the COBC.

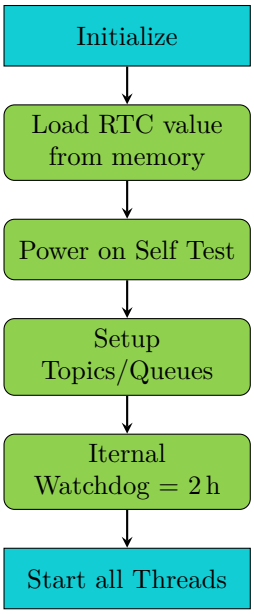

Figure 4.3: Flowchart of the initialization procedure of the COBC.

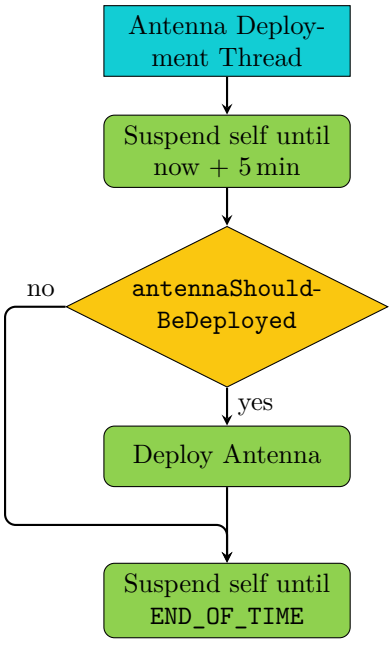

Figure 4.4: Flowchart of the thread that handles the antenna deployment mechanism.

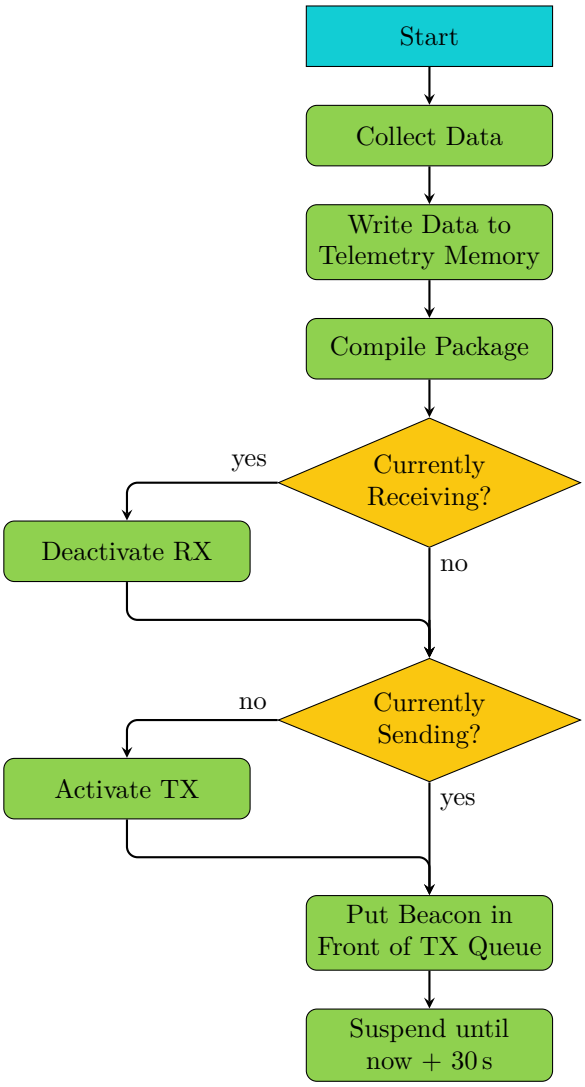

Figure 4.5: Flowchart describing the operations in the “Send Beacon 30s” thread.

Bootloader		
	Not-Ok Counter	Counts the occurences of negative firmware integrity checks
	Active firmware image	The version number of the firmware, that is currently running on the COBC
	Backup firmware image	The version number of the firmware, that is currently set as the backup firmware
	Hashes Ok	Flag that indicates whether the hashes of the backup firmware image is currently consistent.
EDU		
	EDU active flag	Binary flag that indicates whether the EDU module is currently active
	Current queue ID	ID that indicates the current position in the EDU queue
	Current progam ID	ID that indicates the currently running program
	Next queue ID	ID that indicates the next position in the EDU queue
	Next program ID	ID that indicates the next program to be executed
	New results available flag	Flag that indicates that new result files can be fetched from the EDU module
	Process Queue error flag	Flag that indicates an error regarding the process queue
Housekeeping		
	Reset counter	Number of resets, since mission start
	Last reset reason	The reason code of the most recent reset
	RODOS time	RODOS internal time
	Synchronized time	Best guess time, calculated by RODOS with data from the GPS module
	Antenna deployment status	Flag that indicates whether the antennas have been deployed
	Firmware integrity	Flag that indicates whether the firmware is consistent over all firmware memories
	Sensor data	A selection of sensor readings. The exact composition will be defined in future documents
Comm		
	Uplink FEC correctable errors	Count of errors that have been corrected by FEC
	Uplink FEC uncorrectable errors	Count of error that could not have been corrected by FEC
	Count bad RF packets	Count of packets that could not be decoded
	Count good RF packets	Counts of packets that have been decoded successfully
	Last received command	The most recent received command

Table 4.1: Beacon contents

4.1.5 Communication Management

This subsection describes the RF communication between STS1 and the ground from the perspective of the COBC. For a general overview of the communication with the ground segment see section 2.5. For the communication of the COBC with the EDU module see section 4.1.7.

The basic concept consists of a send queue, where other threads can add entries with readily packaged data and a priority, as well as a communication management thread. So far only two priorities are planned: beacons have a high priority, the rest has a low priority. The communication thread is executed regularly and frequently, e.g., every 10 ms. First of all it checks whether a beacon is due to send. If that's not the case, it listens for incoming messages and only then sends other data packets waiting in the queue. To notice that the RF module receives something it has an interrupt pin that triggers an event in the COBC FW. After every beacon the communication thread waits a certain amount of time in the order of seconds, in receive-only mode. This is important since it allows us to send commands from the ground even during a large data downlink. Managing RX and TX for the rest of the time can be done in multiple ways. For high data throughput the RF module is in RX (=receive) mode by default and only if nothing is received can non-beacon data be sent. This requires that the threshold for signalling incoming messages is set correctly, i.e., it doesn't trigger permanently due to noise and interference thus preventing sending any non-beacon data. A simpler, more robust possibility is firm time slotting. In this case there are periods of fixed time, e.g., 5 s, where the RF module is alternating between RX and TX (transmit) mode. This approach has obviously lower data throughput.

NB: Since the send queue is not stored in a persistent memory, it cannot be recovered after a reset. This is done on purpose to prevent ending up in a send-and-reset loop.

Sending data is relatively easy since the send queue contains packaged data, i.e., the data is already integrated into the protocol frame according to our (yet to be chosen) protocol with headers, a checksum, a unique packet ID, etc. and just needs to be sent to the RF module. Receiving a message requires more effort. It must be unpacked, checked for errors – which can hopefully be corrected with FEC (forward error correction) – and parsed. If all that was successful, the internal watchdog timer is reset to 24 h. Only after all that is one able to determine what to do with the packet. If the it is part of a data upload it is written to one of the external memories (see section 3.1). If it is a command, an entry in the command queue is added (see section 4.1.6). This distinction is necessary because commands are executed with quite a low priority, but incoming data transmissions should be redirected to the right memory as soon as possible to prevent the need for large buffers.

The first packet of every command contains a Unix timestamp that is as close as possible to the send time of the GS and used to get an estimate for the current time on board STS1. To prevent time jumps, the current value of the real time clock (RTC) is not just overwritten with the estimate. The offset will be reduced smoothly by slightly changing the clock frequency for a certain amount of time. If the offset is larger than a certain threshold a jump forward in time is still permitted but a backwards jump must never occur. The time jumps would most likely be necessary after a reset when the RTC starts at zero. To decrease the max possible offset and the number of jumps, the current value of the RTC is regularly stored in persistent memory and loaded during initialization (see section 4.1.2).

4.1.6 Command Handling

Apart from collecting, storing and sending telemetry data, the COBC does nothing on its own without receiving a command. As mentioned in section 4.1.5, if an incoming transmission is identified to be a command, it is added to the command queue. A low-priority thread processes this queue when there are no other more urgent or important things to do – like sending beacons. Section 2.1 gives a list of all available commands with short descriptions. It is possible to also add priorities to the commands

but such details have not been discussed so far. Note that the command queue is also not stored in a persistent memory, so it cannot be recovered after a reset. This is done on purpose to prevent ending up in a do-command-reset loop.

4.1.7 EDU Management

The EDU management is split up into two threads, one thread (fig. 4.6) is responsible for starting the RaspberryPi by setting a GPIO pin to high, transmitting all the new archives from the COBC file system to the RaspberryPi, telling the RaspberryPi to start executing a program or to terminate the currently active program. Once an archive in the COBC file system has been successfully sent to the RaspberryPi the COBC deletes it from the COBC file system. The other thread (fig. 4.7) is responsible for gathering status information from the RaspberryPi and initializing the communication if the EDU sets the info pin, which signals the COBC that it has a new result from a finished program.

It is planned to implement a heartbeat signal which is triggered by the RaspberryPi see section 4.2.1. This signal shall be used to tell the COBC if the RaspberryPi has a problem. Possible responses from the COBC could include resettign the RaspberryPi or turning it off until the next SW reset of the COBC.

To know when the EDU should run a program the COBC has two memories. the the “Process Queue” and the “Process Status and History”.

Process Queue

The “Process Queue” is one static memory block which only gets changed by the Queue bauen command. The “Process Queue” consists of a list of entries, one for each program execution on the RaspberryPi. This entries consist of the Program Id, Queue Id, start time and maximum execution time. The Program Id is the Id of the Program which gets executed on the RaspberryPi, the Queue Id is the unique identifier for the generated result files, the start time is the absolute time when the program on the RaspberryPi should be executed and the maximum execution time is the maximum time the RaspberryPi is allowed to execute the program, afterwards the program gets stopped. The “Process Queue” also includes one static variable the “currentQueueId” which gets changed by the “EDU Queue process” see fig. 4.6, it points to the next entry in the list and gets changed after each successful or unsuccessful execution of one program on the RaspberryPi, not including resets.

Process Status and History

The “Process Status and History” is realized as a ring buffer, it is a list containing an entry for all the programs which already ran on the RaspberryPi or should currently run on it. This entries consist of the Program Id, Queue Id and Status. The Program Id and Queue Id are used to match the result file to a Program Id and Queue Id in the “Process Queue”, those entries are the same, the Status indicates the status of the program execution, it has several statuses listed as follows:

1. Program should currently run on the RaspberryPi.
2. Program ran into the maximum execution time, and will not run again, nor is there a result file.
3. Program is finished but the result file is not yet stored on the COBC file system.
4. Program is finished and result file is stored on the COBC file system.
5. The result file has been sent to the GS.

6. The COBC received an ACK from the GS that the result file has been received.
7. The COBC is in the process of deleting the result file, but it is not yet finished.

The "Process Status and History" gets changed multiple threads, the status from one to three are written by the two threads listed in section 4.1.7 see fig. 4.6 and fig. 4.7. The status from four to seven are managed by the command handling thread which gets triggered after a command is received see section 4.1.5 and section 4.1.6, note that those threads are not yet defined in such detail as the other threads.

4.2 EDU

The EDU module's only objective is to execute Python code. To do this it has to fulfill some secondary objectives, i.e., receiving an archive from the COBC (section 4.2.2) and return the result from the finished execution of a Python file (section 4.2.4). By design each Python file is required to store all its data in one result file. This is done to reduce complexity in the result handling.

To manage all this, there is a background routine running on the RaspberryPi which manages the communication, the RaspberryPi internal time and the file system, see section 4.2.1. One major concept of the CubeSat is that the COBC can not be interrupted by lower priority systems, in this case the EDU. This means that all communication to the EDU needs to be triggered and controlled by the COBC. Once the RaspberryPi is started up, triggered via a GPIO pin from the COBC (section 3.3), it is ready to receive commands from the COBC:

1. The command to store an archive on the RaspberryPi file system. section 4.2.2
2. The command to execute one specific Python file. section 4.2.3
3. The command to stop the running Python file. section 4.2.3
4. The command to send a result file from a finished Python file. section 4.2.4
5. The command to send it a list of all the python files stored on the RaspberryPi to the COBC.
6. The command to do a time update for the RaspberryPi.

4.2.1 RaspberryPi background routine

The RaspberryPi background routine is a python script that gets triggered during startup of the RaspberryPi. It is the master over the RaspberryPi, meaning it is responsible for all the management, see the next sections. Additionally to its other tasks described in section 4.2 it also managed a heartbeat signal to tell the COBC taht the RaspberryPi is still functional. For that it hijacks the PowerLED which gets set to high during startup and puts a periodical signal on it. This signal is read out by the COBC.

4.2.2 Receive archive

When the RaspberryPi receives the command to store an archive as well as the archive itself, the RaspberryPi always stores the archive to it's root folder. After the successful reception the RaspberryPi unzips it, as the files in the archive are placed in the archive with the intent to be unziped in this way they were packed with the correct relative paths. The RaspberryPi does not do any further work with the unziped files until it is instructed to do so by the COBC. section 4.2.3

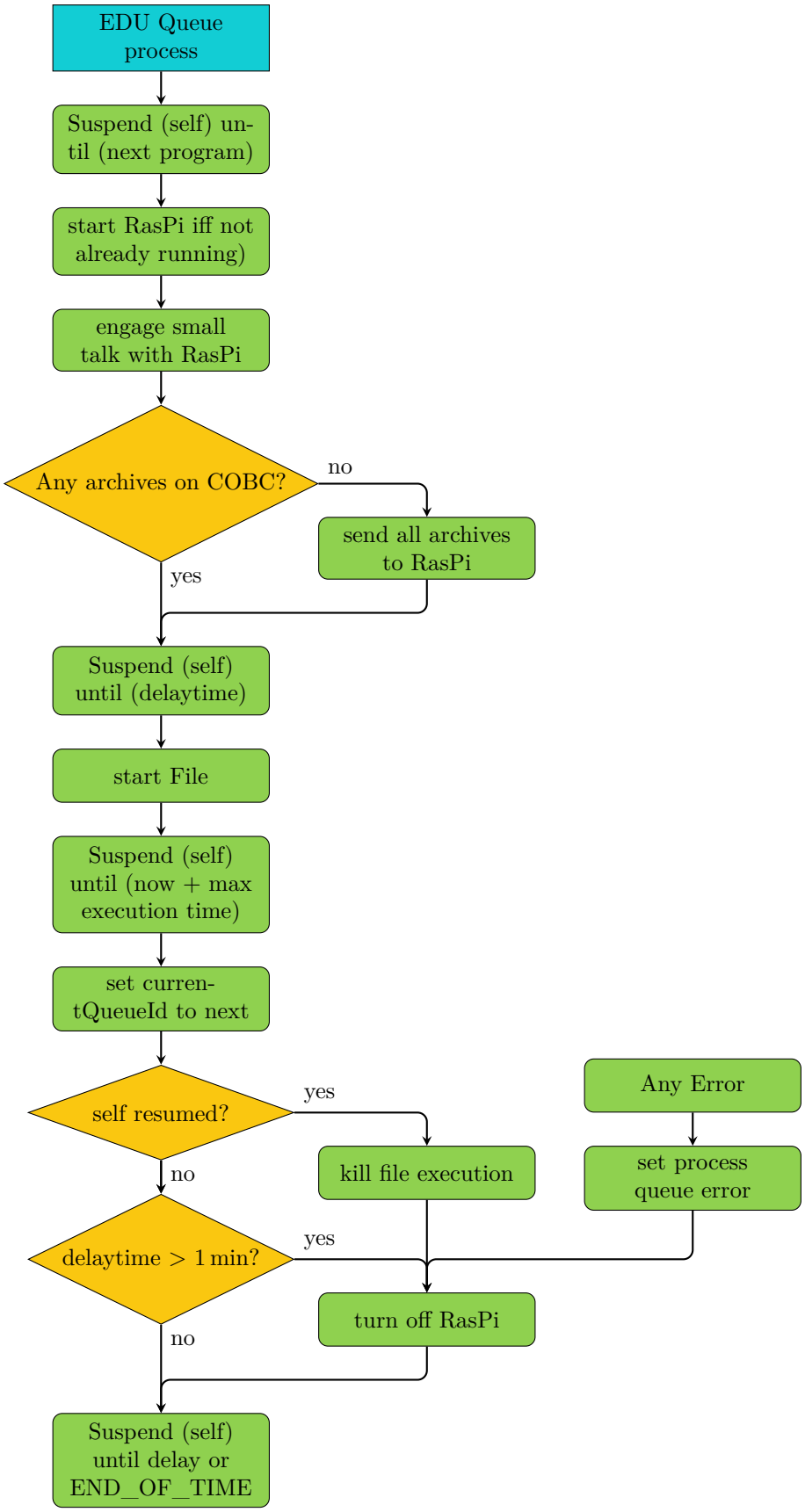

Figure 4.6: Dataflow for EDU queue handling.

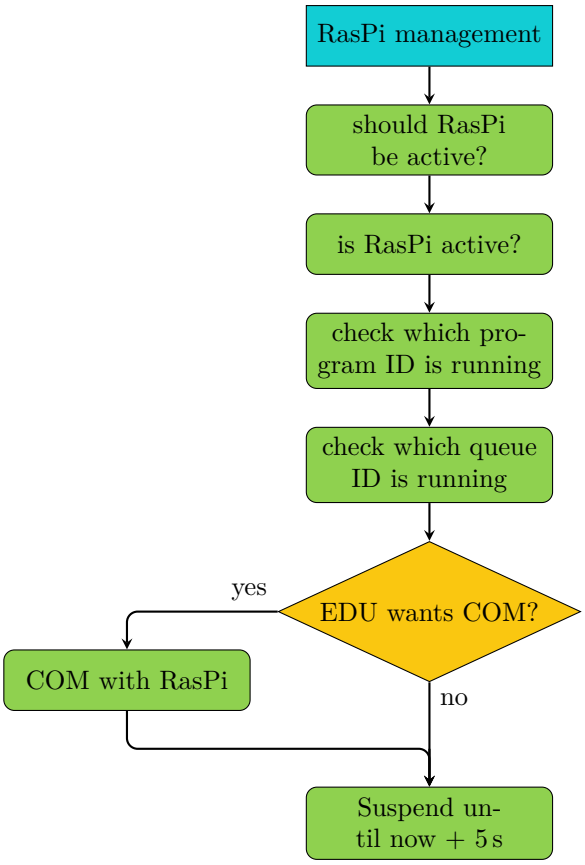

Figure 4.7: Dataflow for managing the Raspberry Pi.

4.2.3 Execute Python Files

The command to execute a Python file contains not only the program ID but also the queue ID. The queue ID can be used when executing the same Python file multiple times to differentiate between the result files. The background routine running on the RaspberryPi then executes the Python file mentioned in the global variable. In case there was a reset of STS1 the same Python file is executed with the same queue ID, this could leads to an already existing results file for this queue ID. It is in the responsibility of the students to handle already existing result files. As soon as the Python file is finished the RaspberryPi sets a GPIO pin (Update Pin) to signal the COBC that it has information to send to it.

The COBC also has the possibility to stop the execution of the currently running Python file. Once the command is received the background routine terminates the currently running Python file.

A short side note, when STS1 is started there are already some executable python scripts on it. This allows us to execute scripts without prior upload.

4.2.4 Return Result Files

The moment the RaspberryPi receives the command to return the result file of a specific Python file with a specific queue ID, the background routine on the RaspberryPi checks if it has the mentioned result file. If the RaspberryPi has the file, it sends the file to the COBC.

Bibliography

[1] ESA: Concurrent Design Facility. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Concurrent_Design_Facility (visited on 28/05/2021).

[2] miro: Dataflow Game. URL: https://miro.com/app/board/o9J_lJEU6tk=/ (visited on 28/05/2021).

[3] COSMOS Software by Ball Aerospace. URL: https://cosmosrb.com/ (visited on 27/09/2020).

[4] SatNOGS – Open Source global network of satellite ground-stations. URL: https://satnogs.org/ (visited on 18/01/2021).

[5] Space Data System Standards. URL: https://public.ccsds.org/Publications/BlueBooks.aspx (visited on 26/07/2021).

[6] Compute Module 3. URL: https://www.raspberrypi.org/products/compute-module-3/ (visited on 04/08/2021).

[7] Rodos (Wikipedia). URL: https://en.wikipedia.org/wiki/Rodos_(operating_system) (visited on 30/05/2021).