SpaceTeamSat1
Preliminary Design Document
Communication and On-board Computer Software

02.04.2023

Version 1.1

Revision History

Revision	Date	Author(s)	Description
0.0	24.05.2022	Patrick Kappl	Document creation
1.0	12.02.2023	Patrick Kappl	Version sent to the reviewers
1.1	02.04.2023	Patrick Kappl	Improved citations

Authors

Contribution	Name	Email
Main author	Patrick Kappl	patrick.kappl@spaceteam.at
Flowcharts	Jerome Hue	jerome.hue@spaceteam.at
Requirements	David Wagner	david.wagner@spaceteam.at

Contents

Abbreviatios and Acronyms

ACK acknowledge 17

COBC communication and on-board computer 1–9, 11, 13, 15–21

CS checksum 4–6, 18

EDU education module 1–3, 7, 10–18, 20, 21

EPS electric power system 1, 11, 12

FEC forward error correction 9

FW firmware 1, 4–9, 17–19

GS ground station 1, 4, 9, 10, 17, 20

HAL hardware abstraction layer 1

HW hardware 1

MCU microcontroller unit 1, 4, 17, 18

RF radio frequency 1, 3, 7–11, 17, 21

Rodos Real-time Onboard Dependable Operating System 1, 7–9, 11

RTOS real-time operating system 1

RX reception 2, 3, 8

STS1 SpaceTeamSat 1 3, 7–10, 21

SW software 1–4, 7

TX transmission 2, 3, 8, 10

1 Introduction

This document gives an overview of the tasks and functionalities of the software (SW) of the communication and on-board computer (COBC). For the most part firmware (FW) and COBC SW are used synonymously in this document. Technically, however, we define that the FW does not contain the bootloader, while the COBC SW does.

Our real-time operating system (RTOS) of choice is called Rodos (Real-time Onboard Dependable Operating System) [1]. It is developed at the University of Würzburg and, among other things, used for satellites [2]. 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. Rodos also provides a hardware abstraction layer (HAL) for our chosen microcontroller unit (MCU), the STM32F411RE [3].

A detailed description of the hardware (HW) of the COBC can be found in [4]. For the purpose of the SW it is sufficient to know that the MCU can communicate with the education module (EDU), the radio frequency (RF) modules and the sensors on the electric power system (EPS). It has also access to persistent memory modules which are logically divided into the so-called persistent state, telemetry memory, COBC file system, primary FW partition, secondary FW partition 1 and secondary FW partition 2. For this high-level view on the COBC SW the exact mapping between these logical memories and the physical ones is only important in two cases: First, the primary FW partition contains the code that is actually executed by the MCU and therefore has to be on its internal flash memory. If space allows it the secondary partitions will be there as well. Second, the COBC file system is based on an external flash memory. Most file system libraries for embedded applications are only implemented for a single memory technology and flash is chosen because it allows for the largest memory capacity.

When it comes to error handling, we use a very simple approach. We defined that the COBC SW is working properly as long as it can communicate with the ground station (GS). If that is not the case the COBC gets reset. Similarly, if errors occur when communicating with the EDU or the sensors on the EPS, the COBC resets them and tries again later.

Chapter 2 lists the requirements that lay the foundation for the functionality of the COBC SW. Chapter 3 describes the bootloader which implements a mechanism of self-repair for the FW and also allows updating it while in orbit. The high-level tasks of the COBC and the related Rodos threads are defined in chapter 4. A flowchart is provided for most threads. Chapter 5 lists all commands that can be sent from the GS to the CubeSat. Chapter 6 provides more information on the COBC file system and chapter 7 describes the protocols used for communicating with the GS and the EDU.

2 Requirements

The following subsystem requirement is deduced from STS1 System Architecture Requirement 3.1 [5]: The CubeSat shall automatically send out a beacon at least once every 30 s.

3.1.COBC_SW.1 The COBC SW shall automatically send out a beacon every 30 s, with a tolerance of ±2 s.

Fulfilled: In sections 4.4 and 4.5

The following subsystem requirement is deduced from STS1 System Architecture Requirement 4.1. [5]: The CubeSat shall always be able to receive a command in an acceptable amount of send time.

4.1.COBC_SW.1 There shall be a mechanism in place in the SW of the COBC that disables the transmission (TX) path and activates the reception (RX) path for at least two continuous seconds every 60 s.

Fulfilled: In section 4.4

The following subsystem requirement is deduced from STS1 System Architecture Requirement 7.1. [5]: The CubeSat shall be able to deactivate all systems not necessary for communication once a command is received to do so.

7.1.COBC_SW.1 There shall exist a command to deactivate all the communication parts which are necessary solely for TX.

Fulfilled: In section 5.1

7.1.COBC_SW.2 There shall exist a command to activate all the communication parts which are necessary for TX.

Fulfilled: In section 5.1

7.1.COBC_SW.3 The COBC SW shall have a command handling in place to deactivate all the communication parts which are necessary solely for TX.

Note: This deactivation shall survive a reboot of the system.

Fulfilled: In section 4.2

The following subsystem requirement is deduced from STS1 System Architecture Requirement 7.2. [5]: The CubeSat should have a central processing unit (here: COBC) which deactivates the EDU, if it is not used in the next couple of minutes.

7.2.COBC_SW.1 The COBC SW shall deactivate the EDU if there is no program scheduled on it for the next minute.

Fulfilled: In section 4.7 and fig. 4.4

7.2.COBC_SW.2 The COBC SW shall activate the EDU in time for it to be done booting when the next program shall be executed.

Fulfilled: In section 4.7 and fig. 4.4

The following subsystem requirement is deduced from STS1 System Architecture Requirement 8.1. [5]: The CubeSat shall be able to receive one full program code at least once every 24 h.

8.1.COBC_SW.1 The COBC shall be able to RX with a data rate of 9600 Bd.

Note: It is required to receive 1 MiB within the average uplink time per day, which is 30 min. With a safety factor of 2 for protocol overhead, bad line of sight, bad weather conditions, etc. this leads to a required data rate of approx. 9300 Bd.

Fulfilled: In sections 4.4 and 7.1

The following subsystem requirement is deduced from STS1 System Architecture Requirement 8.6. [5]: SpaceTeamSat 1 (STS1) shall be able to download the entire generated data of one student code at least once every 48 h.

8.6.COBC_SW.1 The COBC shall be able to TX with a data rate of 9600 Bd.

Note: It is required to send 2 MiB within the average downlink time per two days, which is 60 min. With a safety factor of 2 for protocol overhead, bad line of sight, bad weather conditions, etc. this leads to a required data rate of approx. 9300 Bd.

Fulfilled: In section 7.1

The following subsystem requirement is deduced from STS1 System Architecture Requirement 10.5. [5]: The EDU shall be controlled by commands from the COBC.

10.5.COBC_SW.1 The COBC SW shall be able to activate the EDU.

Fulfilled: In section 4.7 and fig. 4.4

10.5.COBC_SW.2 The COBC SW shall be able to deactivate the EDU.

Fulfilled: In section 4.7 and fig. 4.4

10.5.COBC_SW.3 The COBC SW shall be able to communicate with the EDU.

Note: The EDU shall not interfere with the operation of the COBC.

Note: The COBC shall be able to send the EDU a student program and get the result of a student program from the EDU.

Fulfilled: In section 4.7 and figs. 4.6 to 4.8

The following subsystem requirement is deduced from STS1 System Architecture Requirement 11.3. [5]: The CubeSat shall comply with the SW and RF related Fly Your Satellite requirements [6] where it makes sense.

11.3.COBC_SW.1 The on-board SW shall implement a configurable countdown timer that triggers a recovery routine to power the command and control chain and resets to a known working configuration.

Fulfilled: In section 4.4

11.3.COBC_SW.2 The command and control recovery routine shall only be triggered if the ground station does not reset the timer by telecommand.

Fulfilled: In section 4.4

3 Bootloader

As already mentioned in chapter 1 the bootloader, while obviously being SW that is run on the COBC, is not part of the FW. It is responsible for flashing the right FW onto the MCU. We can neither update nor repair the bootloader in case it gets corrupted, e.g., by radiation. Therefore, its size must be as small as possible to reduce the risk of damaging it.

To ensure a reliable operation of the CubeSat and prevent the FW from being corrupted over time we employ a mechanism of self-repair. Each FW image always includes a checksum (CS). Additionally, a copy of the active image which runs from the primary FW partition, is stored in one of the two secondary ones. In regular intervals (30 min) the checksums for both these FW images are recalculated and compared to the old checksums that were 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 in the primary and secondary partitions, and overwrites 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 15 min later. Figure 3.1 shows the flowchart of the thread that does this FW consistency check while fig. 3.2 shows the flowchart of the bootloader.

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 another secondary FW partition where the new image can be written to. Additionally, there are two variables: `backupSecondaryFwPartition` and `activeSecondaryFwPartition`. They store which secondary partition contains the active and backup FW respectively. Both of these variables are in a persistent memory. To update the COBC FW one has to go through the following procedure which is also illustrated in fig. 3.3:

1. Upload a new FW image to the CubeSat, store it in the secondary partition that is not currently active and check its integrity.
2. Set `backupSecondaryFwPartition` to `activeSecondaryFwPartition` (should already be the case but better safe than sorry).
3. Set `activeSecondaryFwPartition` to the partition with the new image.
4. Reset the COBC causing the bootloader to flash the new FW to the primary partition on the internal flash of the MCU.
5. After verifying that the new FW works as intended, set `backupSecondaryFwPartition` to `activeSecondaryFwPartition`.

More details on the exact commands that are sent to the CubeSat can be found in section 5.3.

To prevent an update with a faulty FW from rendering the COBC useless, a backup mechanism is implemented. The variable `nResetsSinceRf` is a reset counter incremented by the bootloader and stored in a persistent memory. As the name suggests it is reset every time a command from the GS is

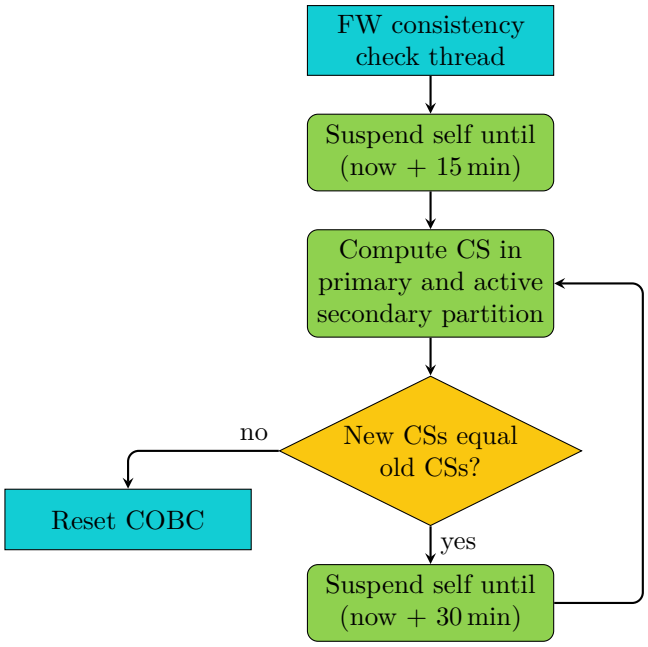

Figure 3.1: Flowchart of the thread, performing the FW consistency check. If one of the CSs differ, the COBC is reset. The bootloader (see fig. 3.2) then also checks the FW images and repairs the faulty one.

successfully received. This event also resets the external watchdog. Therefore, if the new FW cannot communicate with the ground the COBC is reset frequently. If too many of those resets happen, the bootloader switches to the backup FW. A flowchart of this shown in fig. 3.2.

It is important to note here that the FW integrity check does not look at the backup FW. Since the backup FW is only stored once there would be no way to repair it anyway. 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.

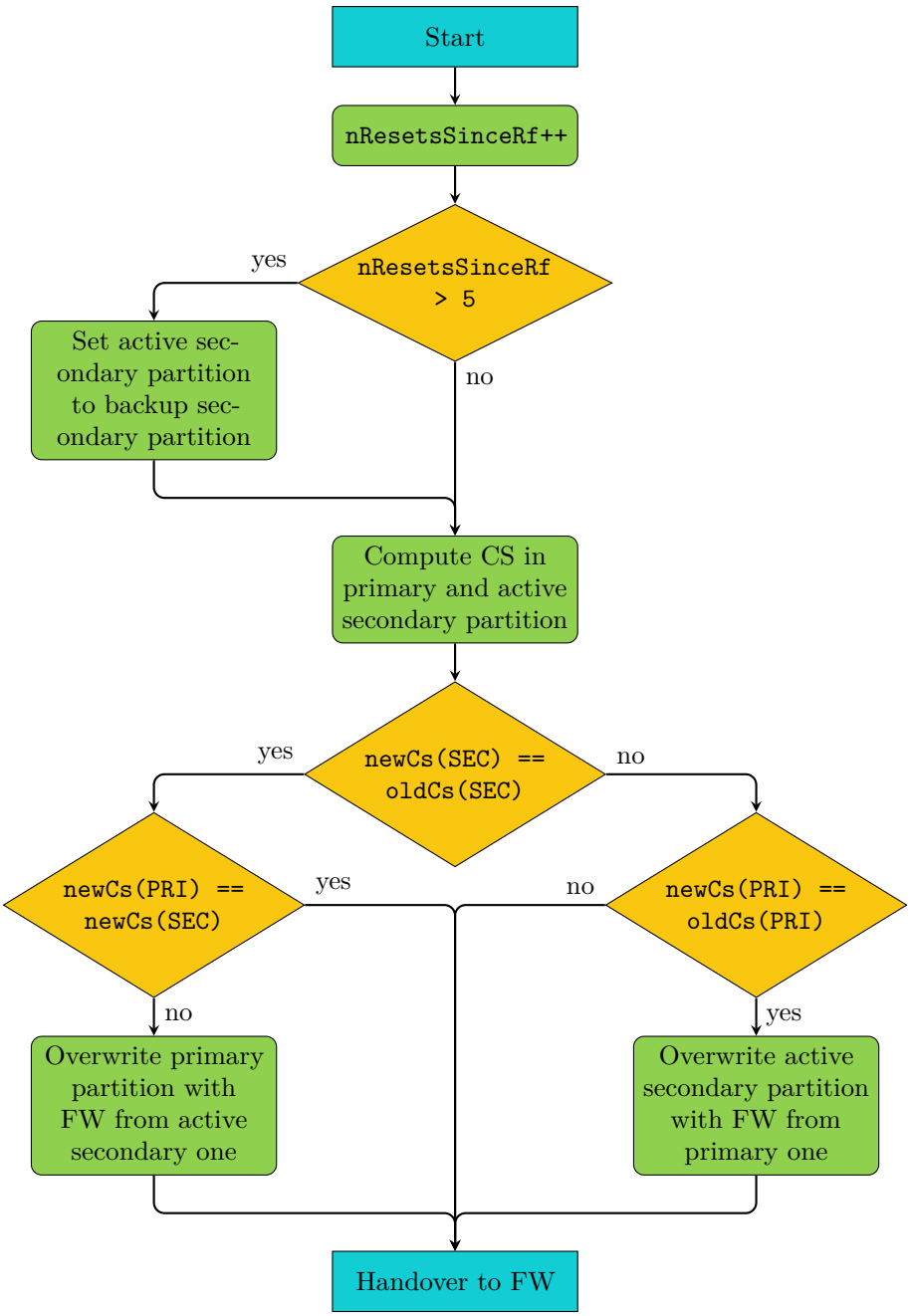

Figure 3.2: Flowchart of the bootloader of the COBC. PRI and SEC are the primary and active secondary partition respectively.

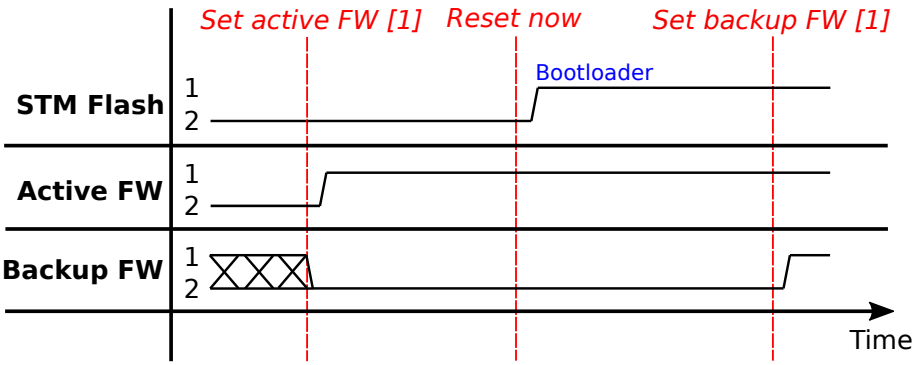

Figure 3.3: Timing diagram for the FW update procedure.

4 Tasks

All tasks of the COBC SW except for the FW consistency check are described here. There are one or more threads to fulfill each task. Most threads are described with the help of flowcharts. First, however, we explain how time works on board STS1 since this is of general relevance and importance.

4.1 Time estimation

There are two different concepts of time for the COBC. We call them the "Rodos time" and the "real time". The Rodos time is the time since the last reset and is used by Rodos to schedule the threads, do time loops and delays, etc. The real time is an estimate for the Unix time on board our CubeSat and is only used for scheduling programs on the EDU. Every time an RF command is received the COBC computes the difference between the Unix timestamp of that command and the current Rodos time. This difference is than stored as the so-called "time offset" in the persistent state. It allows to easily compute the current real time from the current Rodos time. After a reset of the COBC, however, the Rodos time is also reset. This basically invalidates the time offset. To compensate for that not only the time offset but also the real time is stored in the persistent state. This is done every 10 s by the aptly named save real time thread. During initialization a new time offset is computed based on this real time and the current Rodos time. This mechanism ensures that after a reset, the real time is only off by at most 10 s plus the time the COBC was powered down.

4.2 Initialization

Figure 4.1 shows the flowchart for the initialization procedure of the COBC. The persistent state contains the following data:

- `nResetsSinceRf`
- `activeSecondaryFwPartition`
- `backupSecondaryFwPartition`
- `txIsOn`
- `antennasShouldBeDeployed`
- Real time (Unix time estimated from last received RF command)
- Time offset (= real time − Rodos time)
- Various parameters
- EDU program queue
- EDU program status and history

The first three variables are required for the bootloader (see chapter 3). The time related variables are explained in section 4.1. The rest is described in the following sections.

The power-on self-test has yet to be defined and might be left out entirely if we do not come up with useful tests – and reactions! – that we can perform here.

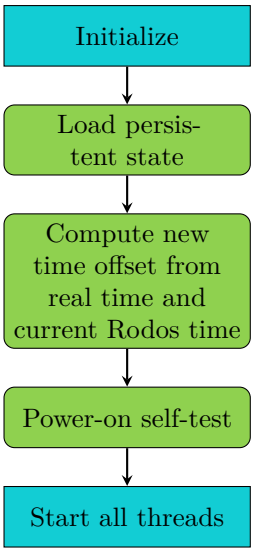

Figure 4.1: Flowchart of the initialization procedure of the COBC.

4.3 Antenna deployment

Figure 4.2 shows the flowchart of the antenna deployment thread. It is assumed that the solar cells' power is enough to successfully deploy the antennas, 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 `antennasShouldBeDeployed` is stored in persistent memory and initially set to `true`. Only via a command (see chapter 5) can it be set to `false`.

4.4 RF communication

This subsection describes the RF communication between STS1 and the ground from the perspective of the COBC. The basic concept consists of a high- and a low-priority TX queue, where other threads can add entries containing readily packaged data as well as an RF communication thread. So far only beacons have a high priority, the rest has a low priority. The communication thread is executed every 10 ms. First, it checks the high-priority TX queue, then it listens for incoming messages and only then sends other data packets waiting in the low-priority TX queue. This means that sending a beacon stops ongoing RX operations and partially received data packets are lost. 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 for 5 s in receive-only mode. This is important since it allows us to send commands from the ground even during a large data downlink. For the rest of the time the RF module is in RX mode by default and only if nothing is received, data form the low-priority queue can 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.

NB: Since the TX queues are not stored in a persistent memory, they 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 TX queues contain packaged data, i.e., the data is already integrated into the protocol frame according to our RF communication protocol (see section 7.1) with

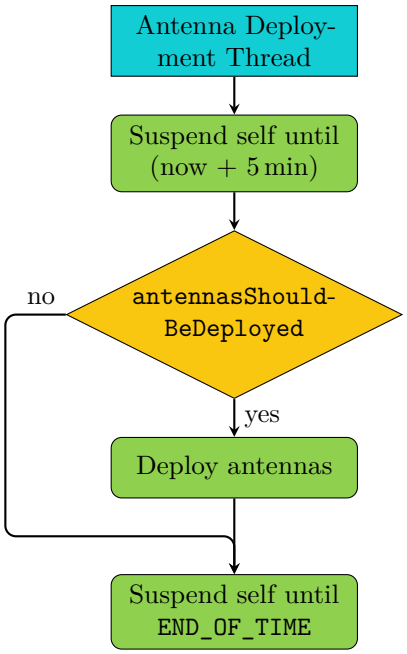

Figure 4.2: Flowchart of the thread that handles the antenna deployment mechanism.

a header, 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 forward error correction (FEC) – and parsed. If that was successful, the external watchdog timer is reset. Only after all that we can determine what to do with the packet. If it is part of a data upload it is written to one of the external memories. If it is a command, an entry in the command queue is added (see section 4.5). 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. Upon successfully receiving a command, the difference between that timestamp and the Rodos time (= time since reset) is computed and stored in a persistent memory. The time offset allows to easily estimate the real time on board the CubeSat. See section 4.1 for more details on the topic of time.

Nice to have: adaptable baud rate for the RF module. The absolute lower limit is 1200 Bd, a realistic expectation is 9600 Bd, more baud rate would be nice.

4.5 Command handling

Apart from collecting, storing, and sending telemetry data; trying to deploy the antennas of the COBC; as well as checking the FW integrity, STS1 does nothing on its own without receiving a command. As mentioned in section 4.4, 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. Chapter 5 gives a list of all available commands with short descriptions. It would also be possible to add priorities to the commands, but that has not been discussed so far. Note that the command queue is 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.

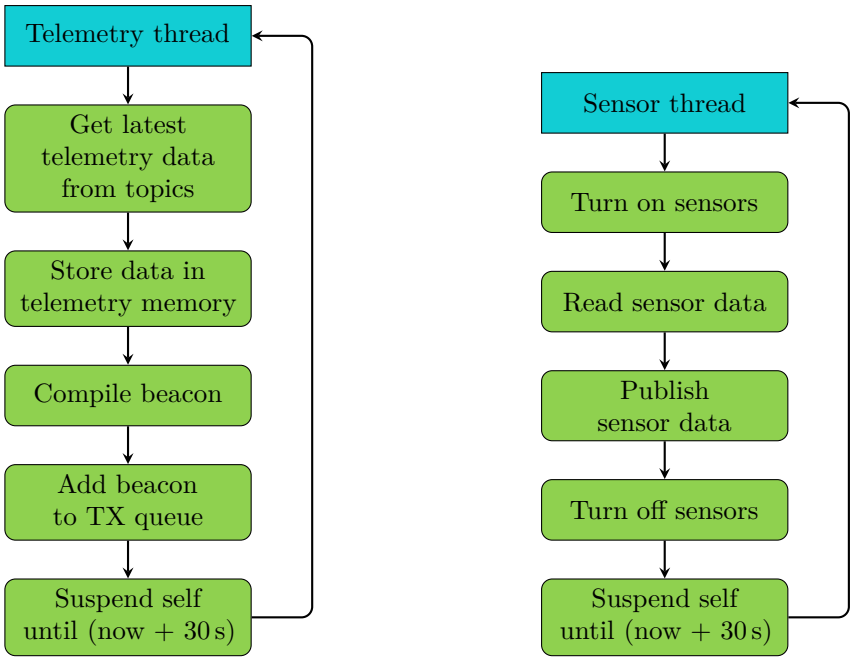

Figure 4.3: Flowchart of the telemetry thread (left) and the thread that collects and publishes sensor data for the telemetry/beacon (right).

4.6 Telemetry

The CubeSat's telemetry allows to discover its state from the ground. A telemetry packet is broadcast by STS1 automatically every 30 s. This automatic broadcast is called the beacon. Its occurrence can only be suppressed, by completely deactivating outgoing CubeSat traffic via the corresponding command Disable CubeSat TX. Compiling, storing and sending the telemetry data is handled by its own thread (see fig. 4.3 for its flowchart). Since the beacon is added to the high-priority queue, it is sent immediately and ongoing transmissions or receptions are aborted. For more details on the RF communication see section 4.4.

In order to be able to fetch telemetry data that has not been received by the GS, it is saved persistently on the CubeSat in the telemetry memory. The telemetry data packets can be requested from the ground station via the command Send telemetry records X–Y (see section 5.1). The telemetry memory is a ring buffer and should be large enough to store all telemetry data of the last week. The required memory for this is estimated to be about 6 Mibit.

The telemetry/beacon data contains the following:

- `nResetsSinceRf`
- `activeSecondaryFwPartition`
- `backupSecondaryFwPartition`
- `fwChecksumsAreOk`
- `eduShouldBePowered`
- `eduHeartBeats`
- Index of current EDU program queue entry
- Program ID of current EDU program queue entry
- Program ID of latest status and history entry
- Status of latest status and history entry
- `newResultIsAvailable`
- Number of EDU communication errors since last beacon
- Reset counter
- Last reset reason

- Rodos time (= time since reset)
- Real time (= Rodos time + time offset)
- `antennasShouldBeDeployed`
- Sensor data
- RF baud rate
- Number of correctable errors in uplink
- Number of uncorrectable errors in uplink
- Number of bad RF packets
- Number of good RF packets
- Last received command ID

The sensor data consists of

- Battery voltages
- Battery temperatures
- COBC temperature
- CubeSat bus voltage

It is collected and published in its own thread. See fig. 4.3 for the corresponding flowchart. Since the sensors are read with the same low frequency that the beacon is sent, i.e., every 30 s, they are turned off between reads. This also means that if some error occurs when communicating with a sensor it gets reset anyway before the next read, so no special error handling is necessary.

4.7 EDU management

Broadly speaking, EDU management means deciding when to power the EDU, transferring the programs to it, telling it when to execute which program and retrieving the results. This work is distributed among six threads. In descending order of their thread priority they are the EDU heartbeat, power management, communication error, program queue, archive, and listener thread. All of them have an initial delay of 15 s. This ensures that there is enough time to send a beacon and save a new real time to the persistent state before the EDU management starts (see section 4.1 for details related to time). With that, even if the EDU is damaged in such a way that operating it resets the COBC, we still get beacons and a progressing real time. After this general remark we take a closer at what each EDU thread does.

The EDU heartbeat thread (priority 6) checks if the EDU is alive by polling the heartbeat line and publishes the result as the `eduHeartBeats` topic. To reliably detect the heartbeat, which has a frequency of 10 Hz, the thread is executed with 50 Hz.

The EDU power management thread (priority 5) turns the EDU on or off depending on a “power good” signal from the EPS and whether the EDU is or will be busy in the near future. Figure 4.4 shows its flowchart with the detailed logic. The delay is the time span until the next program shall be executed on the EDU. The check “delay $<$ 1 min” prevents the EDU from frequently being turned off for only a short amount of time. Since the Raspberry Pi on the EDU consumes much more power when booting than when being idle this might also save energy.

The EDU communication error thread is suspended by default. A communication error is defined as either the EDU violating the COBC–EDU protocol (see section 7.2) or not responding within a certain, yet-to-be-defined amount of time. Whenever such an error occurs, the error thread is resumed causing it to reset the EDU and wait in a busy loop until a heartbeat is detected. Due to the high priority of the error thread this busy wait effectively pauses the communication until the EDU is back online. After that the lower priority thread can resume where it left off and the command that caused the error is sent again. See fig. 4.5 for the flowchart of the EDU communication error thread.

The EDU program queue thread is responsible for scheduling the execution of programs on the EDU. To this end it processes the “EDU program queue” and keeps information about previous and current

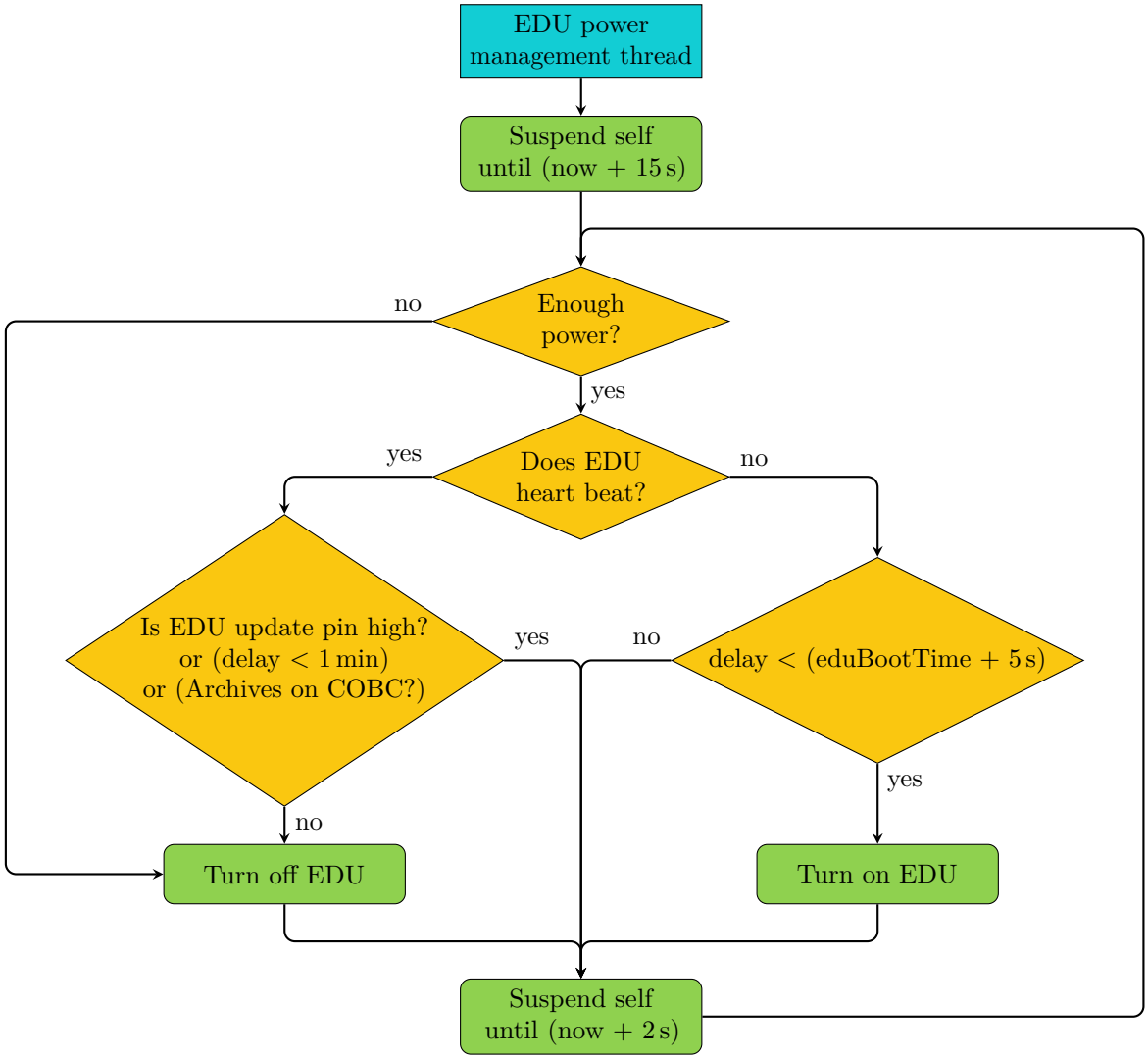

Figure 4.4: Flowchart of the EDU power management thread (priority 5). Whether there is enough power for the EDU or not is signalled via a pin connected to the EPS. The delay is the time span until the next program shall be executed on the EDU.

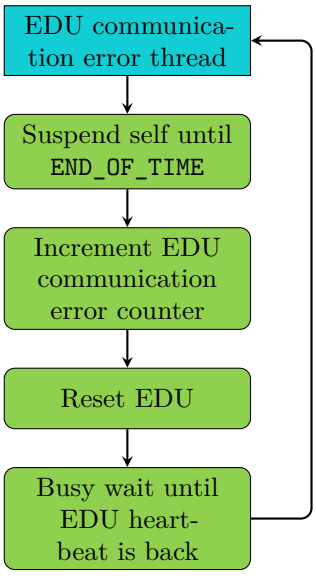

Figure 4.5: Flowchart of the EDU communication error thread (priority 4).

programs in the so-called "EDU program status and history". Both of them are part of the persistent state (see section 4.2). The queue entries consist of a program ID, queue ID (for uniquely identifying different runs of the same program), start time, and max. execution time. To mark the end of the queue a dummy entry is used where the start time is set to `END_OF_TIME`, the largest representable time. The status and history entries are organized as a ring buffer and consist of a program ID, queue ID, and status. There are eight statuses:

1. Program running on EDU
2. Program could not be started
3. Program finished successfully
4. Program finished with an error
5. Result file fully transferred to COBC
6. Result file sent to ground
7. Result file transfer acknowledged from ground
8. Result file deleted from COBC file system

Figure 4.6 shows the flowchart of the EDU program queue thread. Two seconds before a program shall start, the current real time of the COBC is sent to the EDU to update its system time. This makes evaluating result and log files on the ground easier. Then, once the start time is reached the command Execute Program is sent, a status and history entry is created and after waiting for a bit more than the max. execution time, the COBC goes to the next queue entry. If, however, the start time is more than 15 min in the past, program execution is skipped and the COBC immediately goes to the next entry. This mechanism prevents endless operate-EDU-reset loops which could occur since the EDU program queue is saved persistently. It also skips through old queue entries that could not be processed because the power was too low. Of course this should not happen since the "intelligence on the ground" should correctly plan the EDU program queues with the power and energy budget in mind.

EDU programs are sent to the COBC in form of an archive which, in addition to a Python script called `main.py`, can contain any auxiliary files and data necessary for the execution of the program. As long as such archives are in the COBC file system and the EDU is powered, the EDU archive thread tries to send them over. If a full archive is successfully received and stored by the EDU it is automatically deleted from the COBC file system. Figure 4.7 shows the corresponding flowchart.

The EDU listener thread periodically checks the EDU update pin to see if the EDU has information for the COBC. This mechanism is necessary because the COBC is the master in the communication so the EDU cannot interrupt it and directly initiate a data transfer. If the update pin is high the COBC sends a Get Status command to find out what information the EDU wants to send. It notifies the COBC that either a program has finished (un-)successfully or that the EDU wants to send a result file. In the first case the corresponding status and history entry is updated and the EDU program queue thread is resumed. In the second case the COBC requests the result file, stores it in the COBC file system, and then updates the corresponding status and history entry. All of this is visualized in the flowchart shown in fig. 4.8.

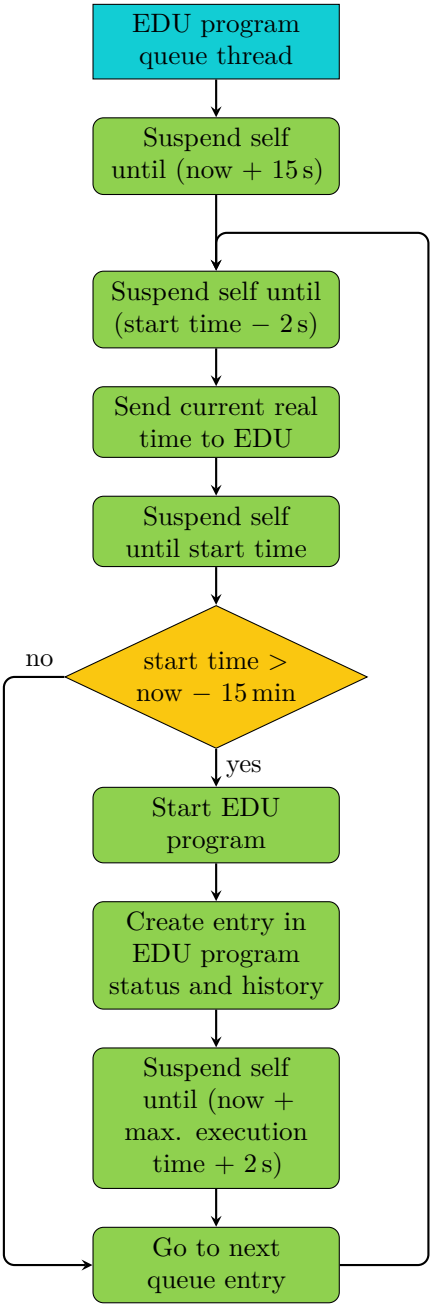

Figure 4.6: Flowchart of the EDU program queue thread (priority 3). Start time is the time at which the next program in the queue shall be executed.

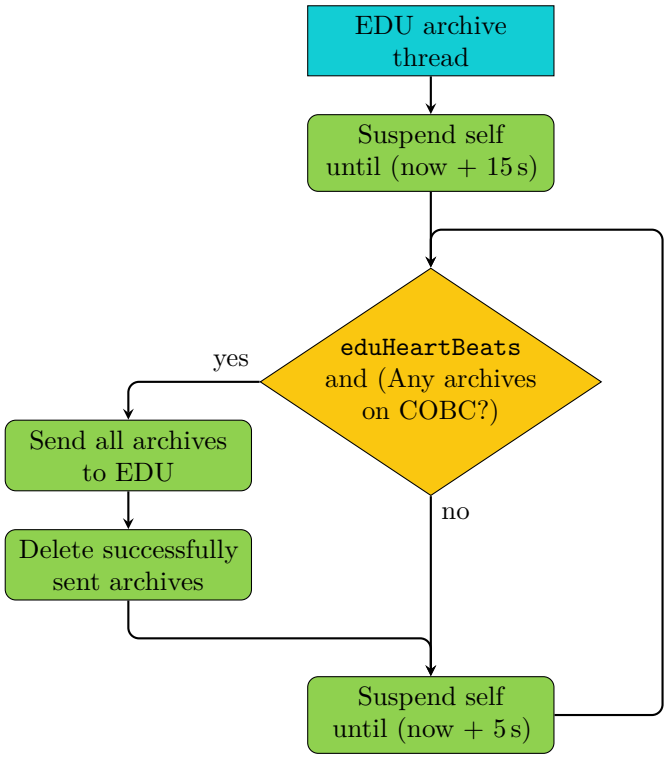

Figure 4.7: Flowchart of the EDU archive thread (priority 2). The archives contain the Python program and any additional files or data necessary for the execution of the program.

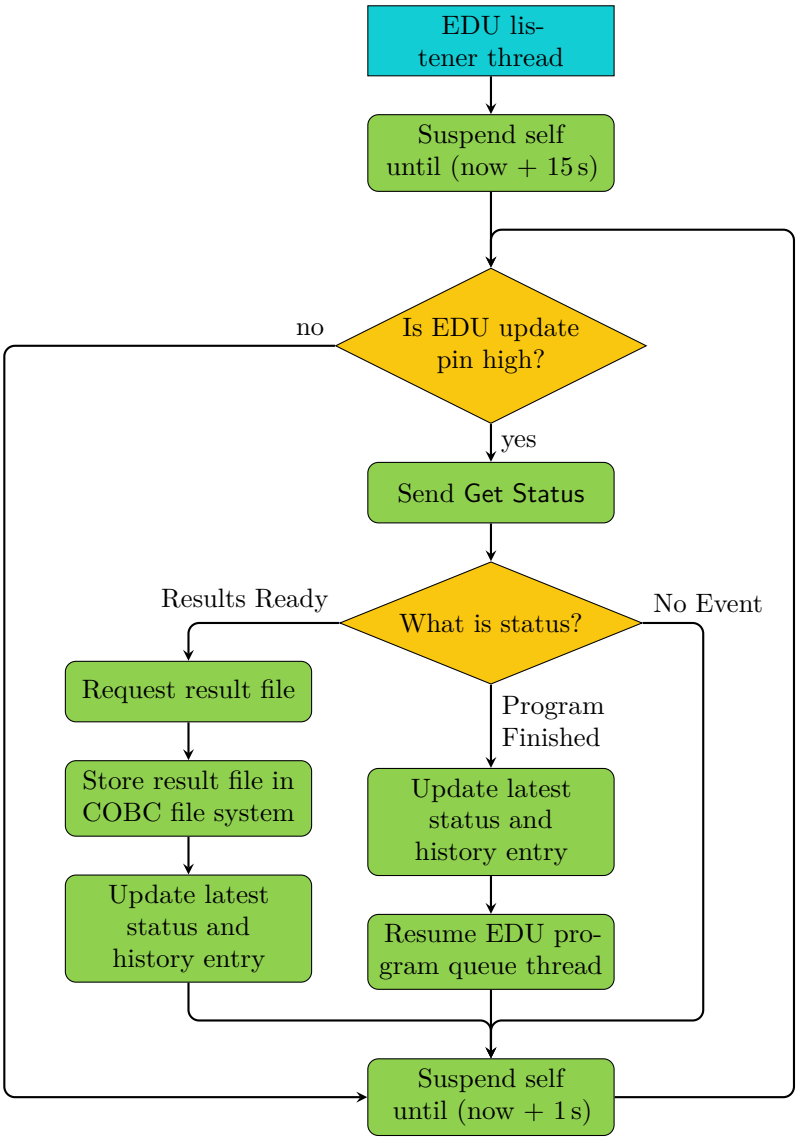

Figure 4.8: Flowchart of the EDU listener thread (priority 1).

5 GS Commands

In the following all commands that can be sent from the GS to the CubeSat are listed and briefly described. Note that justifications of the commands are not explicitly given here.

5.1 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 sets `antennasShouldBeDeployed` to `False` and answers with an acknowledge (ACK).

- Send telemetry records X–Y

  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 parameter

  Allows setting/changing 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.

- Get parameter

  Allows reading 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 MCU. The CubeSat cannot send an answer. However, the reset can be recognized by analyzing the reset counter contained in the beacon.

5.2 EDU Commands

- Upload EDU archive

  Uploads an archive, ultimately intended for the EDU module, and stores it in the COBC file system. The archive contains a Python program and any additional files or data necessary for the execution of the program. First, the corresponding files are selected in the UI. Then they are compressed in an archive, and a CS is calculated. Moreover, the generated package receives a program ID. The CubeSat answers whether the archive was successfully written to the COBC file system or not.

- Update EDU queue

  Uploads a queue – completely overwriting the old one – which determines the execution order of the Python programs on the EDU. The generation of the queue shall be implemented in the UI. A queue entry consists of a program ID, which can directly be assigned to a program name, a queue ID, a start time and a 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 Unix/epoch timestamps. The CubeSat answers whether the queue was successfully updated or not.

- Send list of available results

  Requests a list of all available result files in the COBC file system. The CubeSat answers with the list.

- Send result

  Requests a certain result file, uniquely identified by the queue ID and program ID, stored in the COBC file system. The CubeSat answers with the results file.

- Delete file from COBC file system

  Deletes a certain file, denoted by a filename, stored in the COBC file system. The CubeSat answers with a confirmation after successfully deleting the file.

- (Send list of files in COBC file system)

  Requests a list of all files stored in the COBC file system. The CubeSat answers with the corresponding list. Note that currently it is not clear if this command is required.

- (Send list of programs on EDU)

  Requests a list of all Python files stored on the EDU module. This can be done be utilizing a Python script on the EDU itself. The CubeSat answers with the corresponding list. Note that currently it is not clear if this command is required.

5.3 FW Commands

There are a primary FW partition and two secondary ones. The primary partition is on the internal flash of the MCU and contains the code that is actually executed. The two variables `activeSecondaryFwPartition` and `backupSecondaryFwPartition` identify which of the two secondary partitions contains a copy of the active FW and which contains a backup FW. A more detailed explanation of the whole FW update process, which uses the following commands, is given in chapter 3.

- Upload FW

  Allows to upload a new COBC FW image. In our self-developed COSMOS-based UI the image and the relevant secondary FW partition 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 in the chosen FW partition or not.

- Set active FW

  This command contains the following information: the number of a secondary FW partition and a checksum. It leads to the following changes in the system: First, the given checksum is compared with the checksum in the given secondary FW partition. If they match, the `backupSecondaryFwPartition` gets set to the current `activeSecondaryFwPartition`. Afterwards, the `activeSecondaryFwPartition` gets set to the given secondary FW partition.

- Set backup FW

  Sets which of the two secondary FW partitions 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 secondary FW partition are equal, the backup FW is changed. The CubeSat answers with `backupSecondaryFwPartition`.

- Check FW integrity

  Checks the integrity of the FW image in the given FW partition. This is done by calculating the checksum of the image and comparing it with the checksum stored within the corresponding partition. The CubeSat answers whether the checksums match or not.

6 COBC File System

The COBC file system stores the archives uploaded from the GS as well as the result files send from the EDU on an external flash. The file system should have the following properties:

- Support for at least the following commands: add file, delete file, list files
- Wear leveling (otherwise the lifetime will be too low)
- Small enough memory overhead (MCU RAM and ROM, external memory)
- Preferably no dynamic memory allocations (we do not use a heap so far)
- Some measure to ensure data integrity would be nice
- Builtin support for checksums would be nice (ideally CRC32 since the STM32F411 has a HW unit for that)
- No copyleft license would be nice because Patrick Kappl does not like them.

Possible candidates that we evaluated according to the list above are:

- YAFFS
- EEFS
- littlefs
- JesFs

The winner of the evaluation is littlefs because it is the only flash file system that fulfills all items in our list – even the nice to have ones. It supports common POSIX file system commands, wear leveling, has low and bounded RAM and ROM usage, doesn't need a heap, has some form of built-in integrity check or checksum mechanism (we have to admit that we did not look into the details on that, yet) and uses the BSD-3-clause license.

7 Communication Protocols

7.1 RF Protocol

The RF protocol is not defined yet. Some things that we require or would like to have are listed in the following.

- Has checksum
- Has forward error correction
- Has multi packet support
- Invalid packets can be requested again until all packets are received successfully
- Has variable baud rate to test limits and get results/images faster
- Has low fallback baud rate of 1200 Bd
- …

7.2 COBC–EDU Protocol

The COBC–EDU protocol is described in detail in [7]. Here we just list the possible commands that the COBC can send to the EDU.

- Store Archive (program ID): Send an archive containing a student program with the given ID and possible additional data to the EDU.
- Execute Program (program ID, queue ID, timeout): Execute the program with the given ID. The queue ID is passed to the program as an argument. The timeout is the amount of time the program gets to execute.
- Stop Program: Stop the currently running program.
- Get Status: When the EDU sets its update pin it expects this command and answers with one of the following statuses: No Event, Program Finished (program ID, queue ID, exit code), Results Ready (program ID, queue ID)
- Return Result: Request the next available result.
- Update Time (timestamp): Set the EDU's system time. The timestamp is the real time of the COBC. See section 4.1 for more information on how time works on board STS1.

Bibliography

[1] Sergio Montenegro and Frank Dannemann. "RODOS - real time kernel design for dependability". In: *DASIA 2009 - DAta Systems in Aerospace* 669 (2009), p. 66.

[2] Wikipedia contributors. *Rodos (operating system) — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=Rodos_(operating_system)&oldid=1085095216 (visited on 12/06/2022).

[3] *STM32F411xC STM32F411xE*. Datasheet. Rev. 7. STMicroelectronics. 2017. URL: https://www.st.com/resource/en/datasheet/stm32f411re.pdf.

[4] Raphael Behrle et al. *SpaceTeamSat1 Preliminary Design Document: Communication and Onboard Computer Hardware*. Version 0.2. Apr. 2023. DOI: 10.5281/zenodo.7792683. URL: https://doi.org/10.5281/zenodo.7792683.

[5] Raphael Böckle et al. *SpaceTeamSat1 Preliminary Design Document: System Architecture*. Version 4.2. Apr. 2023. DOI: 10.5281/zenodo.7791608. URL: https://doi.org/10.5281/zenodo.7791608.

[6] *ESA: Fly Your Satellite! programme*. URL: https://www.esa.int/Education/CubeSats_-_Fly_Your_Satellite/Fly_Your_Satellite!_programme (visited on 24/09/2022).

[7] Stefan Galavics, Florian Guggi and David Wagner. "SpaceTeamSat1 Preliminary Design Document: Educational Unit". unpublished. DOI: 10.5281/zenodo.7770145. URL: https://doi.org/10.5281/zenodo.7770145.