UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Physics

Computer Engineering

Indrek Sünter

SOFTWARE FOR THE ESTCUBE-1 COMMAND AND DATA HANDLING SYSTEM

Supervisors: M.Sc. Kārlis Zālīte
Ph.D. Mart Noorma

Tartu 2014

Contents

1 Acronyms and abbreviations

Acronym / abbreviation	Meaning
ADC	Analogue to Digital Converter
ADCS	Attitude Determination and Control System
API	Application Programming Interface
ARM	Advanced RISC machine
CDHS	Command and Data Handling System
CDMS	Command and Data Management System
COM	Communication system
CRC	Cyclic Redundancy Check
CW	Continuous Wave
DMA	Direct Memory Access
EM	Satellite Engineering Model
EPS	Electrical Power System
FM	Satellite Flight Model
FPU	Floating Point Unit
FRAM	Ferroelectric Random Access Memory
GPIO	General Purpose Input / Output
I^2C	Inter-integrated Circuit
ICP	ESTCube-1 Internal Communications Protocol
LEO	Low-Earth Orbit
MCS	Mission Control System
MSb	Most Significant bit
OS	Operating System
OSI	Open Systems Interconnection
PL	Payload
PWM	Pulse Width Modulation
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory

Acronym / abbreviation	Meaning
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory
RTC	Real-Time Clock
SAR	Synthetic Aperture Radar
SBRF	Satellite Body Reference Frame
SEU	Single Event Upset
SPI	Serial Peripheral Interface
SRAM	Static Random Access Memory
SWIFT	Software Implemented Fault Tolerance
TLE	Two-line orbital element set
TM	Satellite Table Model
UART	Universal Asynchronous Receiver / Transmitter
UDP	User Datagram Protocol
XML	Extensible Markup Language

2 Introduction

The main mission of the first Estonian student satellite, ESTCube-1, is to test the reeling out of one $10\ \mathrm{m}$ long tether with the help of centrifugal force and to measure the effect of upper atmospheric plasma acting on the tether. The experiment is performed in polar Low Earth Orbit (LEO) [1]. The main experiment requires at least the following: Attitude Determination and Control System (ADCS) to spin up the satellite to $1\ \mathrm{rps}$ [2], on-board logic for controlling satellite payload (PL) and on-board storage for experiment data. The secondary mission is to take a photo of Estonia from space. This requires the on-board Attitude Determination and Control System (ADCS) to point the on-board camera towards Estonia for taking photos.

On board of ESTCube-1, Command and Data Handling System (CDHS) coordinates the experiment and stores mission data. CDHS runs attitude determination and control algorithms, CDHS controls satellite payload (PL) and ADCS hardware, distributes received telecommands, executes scheduled commands or scripts and stores measurements in on-board memory for radio downlink.

It is possible to order or outsource a pre-designed satellite on-board computer. However, by developing satellite hardware and software ourselves, the educational value of the student satellite project is improved considerably. Moreover, a custom design that has been tailored to mission-specific requirements can be made more power-efficient than a generic design intended for supporting a variety of missions.

The goal of this work is to design and implement the on-board software for ESTCube-1 Command and Data Handling System.

At first, hard and soft requirements are collected from satellite subsystem teams and a preliminary design is drafted. During the design and implementation, requirements usually change, new requirements are added while some of the old requirements become irrelevant. This is especially the case for the first student project that builds the foundation and know-how for future projects. In this thesis,

the author presents only the final requirements and design that have withstood the test of time for more than a year.

Despite the increasing popularity of cubesat student satellite projects, there are only a few theses or articles that describe the architecture or design of cubesat on-board computer software. Hopefully this thesis helps to alleviate the issue.

This thesis is organized as follows: In Section 3, a subset of ESTCube-1 Command and Data Handling (CDHS) on-board software requirements as well as hardware limitations are described. The next section lists the work of other nanosatellite teams on the software design of their on-board computers with similar requirements. In Section 5, an overview of the ESTCube-1 CDHS software design is presented. The following sections describe how CDHS hardware and software have performed during the year of operation in-orbit and based on the experience, what could be improved in the next versions of CDHS firmware. In the appendix, the procedure of CDHS software development, testing is briefly described, followed by a short overview of the software used for both testing as well as for operating the satellite in-orbit. The appendix also cointains a list of files and directories on the USB memory stick that is accompanied with the thesis. The file system contents of several CDHS memory devices are also listed in the appendix. The appendix ends with a few code listings that are referenced in the thesis.

3 Requirements

Since the initial phases of the ESTCube-1 project, CDHS software was planned to do the following:

1. Schedule and execute commands.
2. Gather and store housekeeping and mission data.
3. Run attitude determination and control algorithms, which are developed by ADCS.
4. Control ADCS and PL hardware.
5. Support firmware upgrades.

Data on the health and performance of satellite systems is considered housekeeping data. Sensor measurements, camera images and calculation results based on sensor input are considered mission data.

During the design and development of ESTCube-1 hardware and software, more detailed requirements emerged. Some of the more detailed requirements have been grouped into the following subsections.

3.1 Error handling

On start-up, CDHS must store resets and their types (power-on, watchdog reset, hard fault, stack overflow or malloc failure) in the error log, which could later be checked by satellite operators. Any other errors and warnings such as communication errors with sensors and other subsystems must be logged and satellite must recover. Regardless of the error, CDHS firmware must not freeze.

In order to avoid a long list of inter-subsystem communication errors or communication errors with peripheral devices, it must be possible to disable the logging of these errors when they are not needed.

In case of a hard-fault, stack overflow or malloc failure, the system must log the error and reboot.

3.2 ICP

CDHS must comply with the ESTCube-1 satellite internal communications protocol (ICP) standard. Refer to [3, page 228] for a detailed description of ICP.

ICP calls must be thread-safe, that is, it must be possible to send ICP packets from different tasks running simultaneously on CDHS.

In case of intermittent stability issues with EPS, it must be possible to disable communication with EPS temporarily.

3.3 Operating system

Attitude determination and control algorithms need latencies to be deterministic so that measurements could be extrapolated for the time that torque would be applied. The accuracy of determining the latency should be in the order of $10\ \mathrm{ms}$.

The operating system (OS) must support multi-tasking and must provide mutexes, semaphores and queues for synchronizing operations running in parallel. OS resource usage must be minimal, for CDHS only has $96\ \mathrm{KB}$ of RAM and $256\ \mathrm{KB}$ of flash memory per firmware image.

3.4 Command scheduler

A command scheduler is needed for handling commands from a queue and allowing for commands to be scheduled at specific times. It must be possible to schedule a set of commands so that it will be repeated with a configurable period of execution.

In command structure, command header must pose a minimal overhead to command parameters. Command header must have a command identifier field and a field for the length of command parameters. In order for CDHS to send a response to an enqueued or scheduled command, command header must contain a field that contains the identifier of the subsystem that issued the command.

CDHS must be able to schedule the execution of specific commands at specific date and time.

3.5 Flash file system

On-board CDHS, there are three SPI flash memory devices, $16\ \mathrm{MB}$ each. Flash memory expects data to be written in pages (one page is up to $256\ \mathrm{B}$). Although the memory devices are NOR flash, it is not guaranteed that byte or word writes are safe. To ensure the safety as well as performance, data should be buffered into a buffer of $16\ \mathrm{B}$. Over-programming of already written data is also not guaranteed to be safe. In order to modify a byte, the whole $64\ \mathrm{KB}$ block should be erased. Each block can be erased only up to $100\ 000$ times and the erasing sequence of one block can take up to $3\ \mathrm{s}$, thus, the number of block erasures should be kept low. On erase, a block is filled with 0xFF values.

In order to cope with the aforementioned peculiarities, a file system is needed that would hide the details from higher level program code.

Flash is known to be sensitive to radiation. Conventional flash memories start failing at doses of $10-20\ \mathrm{kRad(Si)}$ due to the charge pump degradation, whereas Ferroelectric Random Access Memory (FRAM) devices can tolerate $280\ \mathrm{kRad(Si)}$ without errors. [4, p. 155] In order to avoid the corruption of the file system, metadata must be stored in FRAM and only file contents can be stored in flash.

A minimum of two types of files would be needed: image and journal. An image file only has value as a whole. The old image can be erased and replaced with a new image. When an image file becomes full, an error must be logged. A journal

file is written one entry at a time and once the file becomes full, it wraps around the end and oldest entries are replaced with new ones. All files should support random access to minimize read and write times as well as to make the file system easier to test and debug. For journal files, it must be guaranteed that when a flash block is erased, the integrity of journal entries in other blocks is not sacrificed.

For simplicity, it can be assumed that rarely are files deleted and files are of constant length. File lengths are can be aligned to full blocks (each file is a multiple of 64 KB) and there are always less than 256 files. Numerical file names are preferred for minimizing the overhead of on-board string manipulation. Directories, links are not needed.

It should be possible to detect and record bad bytes and bad blocks. A bad byte is a byte that contains bits, which cannot be set to their correct values. A bad block is a block that contains more bad bytes than the configuration allows for. The file system should be able to fix bad bytes and skip bad blocks on writing and reading.

3.6 FRAM file system

CDHS has five SPI FRAM devices, 256 KB each. Serial memory devices cannot be addressed directly and the manual handling of addresses and lengths becomes cumbersome as the number of separate regions increases. In order to hide the details from the high level program code, an FRAM file system is needed.

FRAM file system must support the storage of files of arbitrary length. Requirements for file types are the same as for flash file system (image and journal files). There are always less than 256 files. Directories, links are not needed and file names should be numerical.

In case of a corrupt FRAM file system, the error must be reported and system files must be restored.

3.7 I^{2}C, SPI and UART drivers

CDHS has two I^{2}C ports for ADCS gyroscopic sensors and magnetometers as well as two SPI ports for Analogue to Digital Converters (ADCs) on ADCS and PL boards. A third SPI port is only used for memory devices and real-time clock on-board CDHS. [3]

I^{2}C gyroscopic sensors occasionally block the I^{2}C bus, which disables communication with I^{2}C magnetometers and a backup I^{2}C FRAM on-board CDHS. In order to circumvent this issue, I^{2}C driver must automatically recover from bus and peripheral issues caused by I^{2}C devices.

ADCs, CDHS RTC, flash memories and FRAM memories all have different SPI clock requirements. Flash memories support up to $104\ \mathrm{MHz}$, FRAM $40\ \mathrm{MHz}$, whereas for ADCs, the maximum is $10\ \mathrm{MHz}$ and RTC only supports clock rates up to $4\ \mathrm{MHz}$. SPI driver should be able to adjust SPI clock frequency on the fly, in order to minimize memory access times. Direct Memory Access (DMA) should be used for reading from or writing to memory devices via SPI.

I^{2}C, SPI and UART drivers must support asynchronous operation. I^{2}C, SPI drivers should also support blocking mode of operation. Direct calls to I^{2}C, SPI and UART peripherals and hardware must only be performed in driver daemons.

3.8 Time management

CDHS does not have an uninterruptible power supply for RTC. CDHS must update its time via EPS.

In order to synchronize ADCS attitude and raw measurements with CAM photos, time difference between CDHS and CAM must be less than $100\ \mathrm{ms}$. Latencies in the communications protocol between subsystems are not deterministic to the desired accuracy. A separate line is needed to synchronize CDHS and EPS times.

3.9 ADCS algorithms

According to ADCS simulations, the minimum iteration frequency that allows for spin-up to 1 rps is 2.5 Hz [5]. This means that one iteration of sensor measurements, pre-processing, sensor models, Kalman filter and attitude controller must take less than 400 ms to execute. While running ADCS algorithms, CDHS must be able to store sensor measurements, attitude, controller output, run a script in the background and respond to commands issued by satellite operators.

Simplified perturbation model SGP4, used by ADCS to calculate satellite orbital state vectors, relies on double-precision floating point arithmetic calculations. Since CDHS microcontroller does not have a hardware floating-point unit (FPU), it has to perform floating-point arithmetic calculations in software.

It must be possible for satellite operators to enable or disable attitude determination and to select an attitude controller. It must also be possible to configure the iteration frequency for each controller.

Magnetometers must not be measured while the magnetic torquers are active.

3.10 Telemetry storage

CDHS must be able to gather and store housekeeping telemetry from CAM, COM, EPS, ADCS and CDHS itself at a configurable time period.

It must be possible to direct the output of any on-board commands to any files in CDHS memory devices. CDHS must also support configurable logging of specific commands and responses from other subsystems.

Although there is a requirement that CDHS must be able to store CAM images, it is not of high priority because CAM is able to store four photos in RAM. The detailed requirements and design of image transfer between CAM, CDHS and storage in CDHS memory are still to be determined. A simple ICP forwarding of CAM image packets has been enough so far.

3.11 Telemetry buffering

In the initial phases of the project, there was a safety requirement that the satellite must not send any packets, unless it receives a request. This helped to avoid the risk of a Single Event Upset (SEU) causing the satellite to continuously send packets until either the on-board radio transceiver burns down or until the batteries are drained.

However, because each down-link packet had to be requested separately, the efficiency of data down-link was limited. Also, COM is only able to buffer four packets at maximum. A feature was requested, which would allow for CDHS to send the next packet right after COM has finished transmitting the previous one. Mission Control System (MCS) suggested adding unique identifiers for down-link packets as well as the possibility to request for specific packets to be re-sent from the satellite.

3.12 Beacon

Based on the periodically gathered telemetry of different subsystems, CDHS must be able to send packet beacon at a configurable time period. In addition, it must be possible to configure CDHS to compile telemetry data and send it to EPS for transmission as EPS normal mode beacon. [6]

3.13 Hardware control

ADCS magnetometers and gyroscopic sensors are connected to CDHS via I^2C. Sun sensor ADC's are connected to CDHS via SPI. CDHS must be able to configure these devices and perform measurements. CDHS must be able to control magnetic torquers, which are directly driven by EPS.

Before the mission, tether reel and tether end-mass must be unlocked. Tether reel and end-mass locks are directly connected to EPS. Tether reel motor controller

provides two control pins: motor enable, motor direction. Before the launch of ESTCube-1, the motor direction pin was disabled on the reel motor controller for safety reasons. Initially there was a feedback pin from the motor controller to CDHS, which allowed for CDHS to estimate the length of the reeled out tether. Due to software issues on the reel motor controller, the feedback line was also disabled before the launch. However, satellite operators must be able to specify the amount of tether to be reeled out.

For voltage and current measurements on the payload high-voltage supply, CDHS must be able to configure the ADC that is connected to CDHS via SPI and perform measurements. The high-voltage supply provides five control lines. Two lines for switching the polarity of the output. Two control lines for toggling electron emitters. One control line is reserved for switching ground. CDHS must be able to control these lines on command. CDHS must guarantee that both polarity control lines are never pulled high at the same time.

3.14 Firmware upgrade

Due to a tight schedule in hardware assembly and testing, most of the on-board software development and testing had to be postponed for a time after the launch. Consequently, in-orbit firmware upgrades became the highest priority in software development. In order for ESTCube-1 to be able to perform the experiment, it must be possible to upgrade CDHS firmware in-orbit. To improve fault tolerance, CDHS must have at least one fallback firmware image that is selected when boot-up to the other firmware fails.

During the firmware upload procedure, it must be possible to request for a pagemap that indicates the status of firmware pages. Pagemap would be used to re-send firmware pages that the satellite has not received.

Before writing an uploaded firmware image to microcontroller flash, the image must be verified against its checksum.

3.15 Low-level access

In case of a system upset in orbit, it should be possible to collect information on issue. It must be possible to read raw memory regions from external devices connected to CDHS as well as from microcontroller internal memory and microcontroller registers.

It should also be possible for satellite operators to fix the issues either by manually configuring microcontroller pins, overwriting memory or changing the contents of microcontroller registers.

3.16 Configurability

For ESTCube-1, there are 11 communication passes each day with a period of about 1.5 hours. The duration of each communication pass varies from about 3 to 13 minutes. Taking communication passes and packet loss into account, each in-orbit firmware upgrade takes about 1.5 days to complete. Thus, it is desired to have the possibility to make smaller changes in the behaviour of CDHS without resorting to a firmware upgrade each time.

Configurability must not come at the cost of fault tolerance. All critical configuration variables must be checked at run-time. There must be a safe fallback configuration that is applied in case of problems.

CDHS must offer in-orbit configurability for ADCS parameters such as orbital parameters, inertial matrix, controller gains, etc.

4 Related work

Based on the requirements for ESTCube-1 CDHS software, software design solutions being used on other satellites are listed here.

Only a few published articles and theses were found on the software architecture and design of cubesat on-board computers.

4.1 UWE-2

The UWE pico satellite platform is based on an H8 microprocessor from Hitachi. The on-board computer of UWE-2 has $8\ \mathrm{MB}$ of RAM and a flash memory of $4\ \mathrm{MB}$ for permanent storage. In UWE-2 software, system initialization, error handling, inter-module communication and OS calls are performed by a central module in uCLinux user mode. [7]

4.2 SwissCube-1

SwissCube-1 Command and Data Management System (CDMS) runs eCos on an ARM7TDMI processor. The Command and Data Management system software is stored in a non-volatile memory and allows for firmware upgrades during the mission. Firmware upgrade is performed by first erasing the microcontroller flash memory and then uploading the new firmware image part by part. [8]

4.3 NUTS

The NUTS cubesat on-board computer (OBC) runs FreeRTOS on an AVR32UC3 and uses Cubesat Space Protocol (CSP) for satellite internal communication. NUTS is scheduled to be launched by the end of 2014. The NUTS OBC has $2\ \mathrm{MB}$ of RAM, $512\ \mathrm{KB}$ of microcontroller flash and $2\ \mathrm{GB}$ of storage space on NAND flash memory. [9]

The usage of YAFFS2 on NUTS OBC as the flash file system has been proposed [10].

4.4 STRaND-1

The STRaND-1 cubesat uses an off-the-shelf GomSpace A712 "Nanomind" on-board computer (OBC) with 2 MB of RAM, 8 MB of flash memory and runs FreeRTOS. GomSpace-supplied library provides drivers for the I^2C bus, magnetometers and magnetic torquers. STRaND-1 Attitude Determination and Control System (ADCS) software is run on the OBC. [11]

STRaND-1 OBC supports modular firmware upgrades in-orbit, which are performed over a custom protocol Strandatoga that has been derived from Saratoga. Saratoga is a light-weight transport protocol based on the User Datagram Protocol (UDP) [12]. On-board software modules can be replaced, augmented or removed without the need for ceasing any on-board operations. However, only one of the microcontroller flash memory banks is used for storing a full bootable firmware image. The OBC runs a central task, which manages software modules, their timing and provides an Application Programming Interface (API) for accessing hardware and memory management. [11]

In order to perform mission tasks, the OBC uses the scheduling of telecommands. On STRaND-1 OBC, all internal variables can be accessed and modified live, which allows for resolving unforeseen issues in-orbit. [13]

5 Software design

CDHS software can be classified into layers, based on dependencies between software components. Refer to Figure 1 on page 22 for details. On the figure, rectangles with solid lines represent software modules running on dedicated operating system tasks, while rectangles with dashed lines depict code that is called directly. Device drivers are highlighted in green and sensor pre-processing filters are highlighted in violet.

The standard peripheral library for STM32F1 offers C macros and functions for accessing microcontroller peripheral hardware. However, the peripheral library does not allow for the indexing of General Purpose Input Output (GPIO) pins, I^2C, SPI or UART ports, DMA channels. A minimalistic Hardware Abstraction Layer (HAL) has been written to improve the Application Programming Interface (API), as well as account for the differences in the hardware of STM32F1 and STM32F2.

As a Real-Time Operating System (RTOS), FreeRTOS 7.6.0 [14] is used. Based on configuration, CDHS task initializes all the other enabled devices and software modules. I^2C, SPI and UART driver daemons are started. Device drivers are initialized for ADCS gyroscopic sensors, magnetometers, FRAM and flash memories, Analogue to Digital Converters (ADCs) and Real-Time Clock (RTC).

After the initialization procedures, CDHS task is used for the following:

- Low-priority periodical operations:
  - gathering housekeeping data from all subsystems,
  - compiling telemetry for beacon transmission,
  - gathering system statistics,
  - synchronizing RTC with EPS,
  - checking system time against RTC,
  - performing user-defined operations.

- Scheduled time-consuming operations:
  - formatting a file system,
  - clearing file contents,
  - generating trigonometry look-up tables.

ICP task handles communication with other satellite subsystems and provides a foundation for the CDHS command scheduler, which in turn is the basis for scripting.

File systems access FRAM and flash device drivers, CDHS time management routines rely on the RTC device driver. ADCS sun sensors and gyroscopic sensors are periodically polled in separate tasks that perform temperature correction, averaging and conversion into Satellite Body Reference Frame (SBRF). Pre-processed measurements are then read by the ADCS task.

Magnetometer measurements must not be performed while the magnetic torquers are active. In order to simplify the synchronization, magnetometer measurements and pre-processing are called directly from the ADCS task.

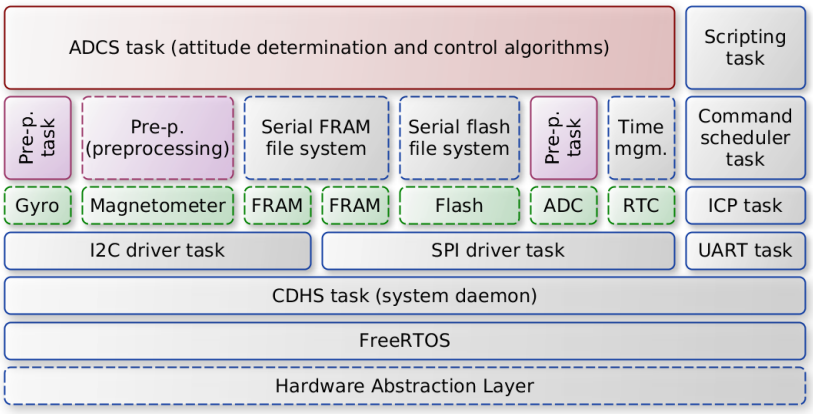

Figure 1: CDHS software layers.

Table 1: Comparison of real-time operating systems.

Name	Minimum Flash footprint	Minimum RAM footprint	License	Ported to
FreeRTOS	4 KB	0.5 KB	free, GPL	Cortex-M3[1]
Salvo	1 KB	0.05 KB	commercial	
ThreadX	2 KB	0.5 KB	commercial	Cortex-M3[1]
Keil RTX	4 KB	0.5 KB	commercial	
eCos	10..100 KB	Not known	free, GPL	Cortex-M3[1]

[1] Officially ported to Cortex-M3 and/or Cortex-M4.

5.1 Operating system

See Table 1 for a few of the real-time operating systems considered in the design phase. As the operating system of CDHS, the free open-source real-time operating system FreeRTOS 7.6.0 was chosen, mainly because of its low flash and RAM footprints and low performance overhead. A minimal configuration of FreeRTOS only needs about 4 KB of flash and 0.5 KB of RAM. On the CDHS, FreeRTOS takes about 11 KB of flash and 64 KB of RAM. Most of the RAM is reserved for the heap of dynamic memory management. [3]

With a proper configuration of priorities, latencies in FreeRTOS can be made deterministic in the order of 10 ms or less.

With barely any modifications, FreeRTOS allows for putting the microcontroller to sleep every tick (in our configuration, one tick equals 1 ms) after a task execution round has been finished. This reduced average current consumption by $10...20$ mA. See Listing 4 on page 64 for source code.

5.2 Error handling

A minimalistic exception handling system was developed in C, with support for multitasking. For each task with an exception handling scope, a fixed-length exception stack is allocated. In the exception handling scope, Duff's device [15] and `setjmp`, `longjmp` functions are used in C macros. See Listing 5 on page 65 for an example.

Error logging has two levels of storage to reliably deliver the list of errors to satellite operators. Errors during boot-up as well as errors reported from interrupt handlers are stored in a RAM section that retains its contents till the next CDHS power cycle. When CDHS is in fall-back mode, error logging to FRAM is disabled or FRAM devices are disabled, then all errors are stored in the RAM section. After the initialization of external FRAM memories and file systems, the RAM section is periodically checked for new errors that could be appended to a circular error log file in the system FRAM. [3]

Errors are stored in FRAM as entries with a $4\ \mathrm{B}$ timestamp, $2\ \mathrm{B}$ exception index and a $2\ \mathrm{B}$ module index. Timestamp indicates the moment that the error was logged. Module index specifies the software module where the exception was caught, exception index identifies the error. An $800\ \mathrm{B}$ file allows for the storage of 100 errors. The error log is circular so that the oldest entries are replaced with the most recent errors. In order to save RAM, only $50\ \mathrm{B}$ has been reserved for errors in the RAM section. By storing errors in a circular buffer without a timestamp nor module index, there is room for 25 most recent errors at maximum. Once the logging storage in system FRAM has been initialized, timestamps and default module indexes are assigned to all the errors.

Errors are divided into groups, such as ICP, I^2C, SPI, command scheduler, etc. The logging of individual groups can be enabled or disabled separately. For on-board statistics, errors from different groups are counted separately. To avoid cases when the error log would become full of the same error, consecutive repetitive errors are logged only once.

5.3 ICP

A cross-platform implementation of the satellite internal communications protocol (ICP) is used in CDHS software. ICP provides Open Systems Interconnection (OSI) model data link and network layers for a mesh topology of UART connections between ESTCube-1 subsystems [3]. See Figure 2 for the ICP topology.

ICP routing table has been configured to send packets directly to the destination, unless there is a problem with one of the links. In case packets are not successfully delivered to COM, the packets are routed through EPS. However, in order to maintain ground communications with CDHS in case of an issue with packet delivery to EPS, CDHS does not forward the packets through COM.

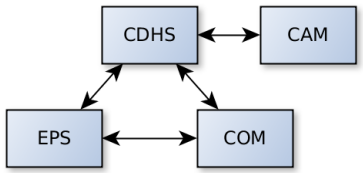

Figure 2: ICP topology on ESTCube-1.

Direct calls to the ICP library have been grouped together into a single FreeRTOS task, to avoid racing conditions. ICP task has a FreeRTOS queue for its operations. This allows for triggering ICP update or packet-send calls from interrupts or any other tasks.

In order to communicate with CDHS hardware via ICP, a USB serial adapter would be plugged to a personal computer (PC) that is running ICPTerminal software. Refer to Section 11.2 for more information on ICPTerminal.

CDHS counts ICP errors with EPS. If the number of consecutive errors reaches a configurable threshold, CDHS temporarily disables ICP communication with EPS for a configurable amount of time (the default is 1 s). With a startup flag, CDHS can be configured to avoid any ICP communication with EPS, until the configuration is changed.

Table 2: ICP endpoint indexes.

Index	Endpoint	Meaning
0	EPS	Electrical Power System
1	COM	Communication System
2	CDHS	Command and Data Handling System
3	ADCS[1]	Attitude Determination and Control System
4	PL[1]	Payload
5	CAM	Camera
6	GS	Ground Station
7	PC[2]	Personal Computer, attached to CDHS
8	PC2[2]	Personal Computer, attached to EPS

[1] Endpoint reserved but not used on ESTCube-1.

[2] Used for debugging purposes only.

5.4 Command structure

In order to minimize the overhead of command metadata, commands are handled in binary form. The same structure is used for both commands that are sent to a subsystem and replies that the subsystem sends back. Command structure consists of a 4 B header and command arguments. The first two bytes of the header are used for identifying the command and the structure of its arguments. The third byte specifies the subsystem that expects the response from the command. The last byte of the command header provides the length of command arguments. See Table 3 on page 27 for the command structure, Most Significant bit (MSb) first. On the ground, binary commands are compiled and responses are interpreted with the help of ICPTerminal.

5.5 Command scheduler

Command scheduler handles commands and allows for the execution of commands at a specific date and time. CDHS command scheduler runs on a separate

Table 3: Satellite command structure.

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7
Byte 0	Command identifier (most significant byte)							
Byte 1	Command identifier (least significant byte)							
Byte 2	Source[1]				Block index[2]			
Byte 3	Length of arguments, in bytes							
...	Command arguments[3]							

[1] ICP endpoint index of the source subsystem. See Table 2 on page 26 for a list of ICP endpoints.

[2] Reserved for command block index, which allowed to group several commands into a single executable block. Not used in the latest design.

[3] Command arguments of variable length. Structure is determined by `Command identifier`.

FreeRTOS task.

Command scheduler supports arrays of commands, which it splits into individual commands that are then pushed into the queue. Command arguments may contain commands, which easily allows for self-repeating commands as well as commands that are executed if specific conditions are met.

For commands scheduled to be executed at a specific time, command scheduler has a pool of FreeRTOS timers that are assigned to commands. On timer overflow, the commands are pushed into the command scheduler queue for execution. The command scheduler task is suspended while the command queue is empty and wakes up when at least one command has been added to the queue. When the command scheduler queue becomes full, an error is logged and further commands are dropped until the previous commands have been handled.

5.6 File systems

Application Programming Interface (API) was designed to be similar for all file systems. File system calls interface a generalized structure with pointers to flash or FRAM file system functions. FRAM file system provides functionality that flash file system does not and vice versa. Unsupported functionality is indicated with function pointers with a value of `NULL`. Function pointers are assigned on mounting.

In the design of both flash and FRAM file systems, support for random access to files and file metadata as well as simplicity of implementation and testing were prioritized. In both file systems, file descriptor index coincides with the numerical file name (file index).

The metadata of a file system starts with a magic word that is used to identify the file system type and version during mounting.

5.6.1 Flash file system

Metadata of the flash file system is fixed to its maximum size, to allow for it to be stored as a file in an FRAM file system. In both the FRAM and flash file systems, files are of fixed length and their maximum size must be known at the time they are created. All files in the flash file system are aligned to flash block boundaries.

See Figure 3 on page 29 for details on the metadata structure of the flash file system. The `device type` field in the device descriptor in flash file system header specifies whether the flash memory device has a parallel or serial interface. In case of a parallel interface, the device address field contains the base address of the memory, whereas in case of an SPI interface, the field contains SPI device index.

The device descriptor also specifies the page, block and write buffer lengths in bytes. A block is the minimum erasable unit, a page is the maximum programmable unit and all writes are aligned to page boundaries. Write buffer is configured for

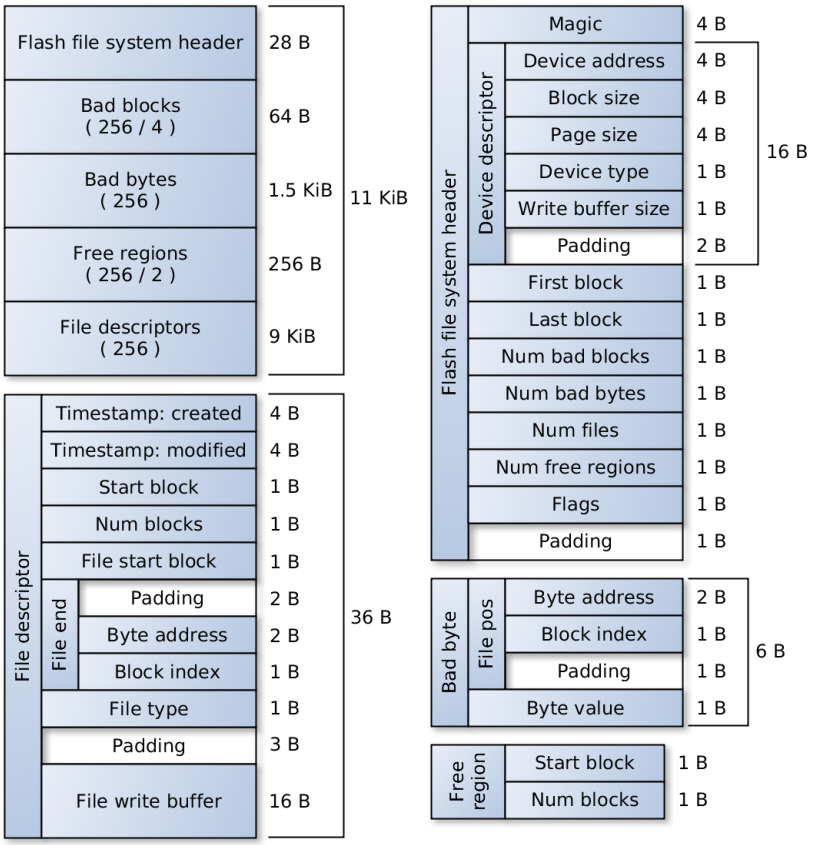

Figure 3: Metadata structure of the flash file system version 3 with the default configuration.

the minimum programmable unit, in case it is 1 B, a write buffer is not needed. On CDHS, SPI flash memory devices are configured with 64 KB blocks, 256 B pages and a write buffer of 16 B.

The `first block` and `last block` fields in the file system header specify the region of the flash memory that is managed by the file system.

The `Num bad blocks` field indicates the number of detected blocks with a mismatch between the data written and read back. The `Num bad bytes` field indicates the number of individual bytes for which, there is a mismatch in the written value and the value read back.

Bad blocks are marked by storing their indexes in the file system metadata, following the file system header. The next region in the file system metadata is reserved for bad bytes. Each bad byte contains the expected byte value and its position in file.

The bad block ratio is configurable at build time, the default value of 4 allows up to $1/4$ of flash blocks to be marked as bad. The maximum number of bad bytes is fixed to 256.

The `Num free regions` field in the file system header indicates the number of continuous free regions in the flash memory. Unused flash blocks are merged into continuous free regions. By maintaining the table of continuous free regions in an external non-volatile memory, microcontroller RAM footprint is reduced without the cost of recalculating the regions each time a file is created or deleted. In the case of maximum fragmentation of file descriptors, half of the flash blocks would be occupied by files and half of the blocks would be marked as free. Thus, for a flash with 256 blocks, the number of continuous free regions can be up to $256/2 = 128$.

The `flags` field in flash file system header is used for configuring the detection and recovery of bad blocks and bad bytes.

At the end of file system metadata, space has been reserved for 256 file descriptors. File descriptor has two optional timestamp fields, the time when the file was created and the time of the last modification (write or seek operations). The `Start block` and `Num blocks` fields define the total capacity of the file. The `file start block` and `file end` fields allow for the file to grow up to the capacity as well as provide support for circular files. `File type` field indicates whether the file is circular or not. At the end of each file descriptor, a write buffer is allocated. Write buffer groups short write calls and aligns them to minimum programmable units before writing to flash.

Once a block becomes full so that the next journal entry would not fit at the end of the block, then the entry is aligned to the beginning of the next block. This way,

journal integrity is not lost when a block is erased and the start block index of the circular file is iterated. On the other hand, without storing additional metadata for each block, each written journal entry must be read back as a whole so that read calls would be aligned the same way. This simplification was not a problem with the initial file system requirements. Later when the requirement of storing the output of any configurable set of commands was introduced, the simplification became an issue.

Padding bytes have been automatically added by the toolchain, to ensure data alignment.

5.6.2 FRAM file system

The FRAM file system metadata consists of a header, followed by a region of file contents, free space and ends with an array of file descriptors. As new files are created, the regions of file contents and file descriptors grow towards each other, until there is no free space left.

Following the magic word, FRAM file system header stores an address that points at the end of the contents of the last allocated file (`Last file address`). With files being deleted and added, the file contents region may become fragmented. Since the amount of free space is checked each time a new file is created, the `Last file address` field saves the read-out of all file descriptors. In order to minimize file access times, file descriptors are random accessible by file index. File contents, however, are allocated in the order of file creation. See Figure 4 on page 32 for details on the structure of the FRAM file system metadata.

FRAM device descriptor contains device type, device index and volume length in bytes. FRAM file system header ends with the number of files and file system flags.

File descriptor starts with optional timestamps. The `Address of contents` points at the base address of file contents. `Total capacity` specifies the maximum length of the file contents and `Current length` is the number of bytes that has

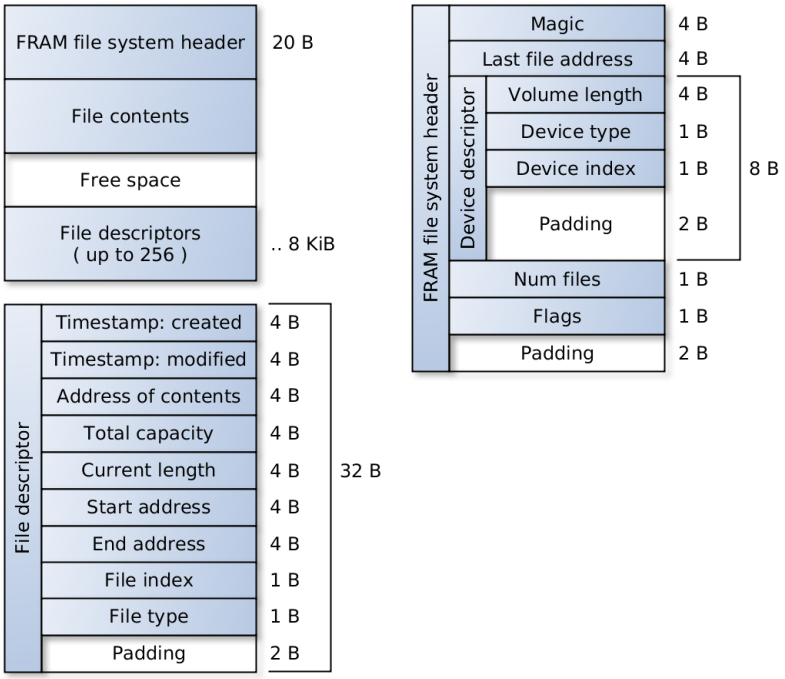

Figure 4: Metadata structure of FRAM file system version 3 with the default configuration.

been written into the file contents. `Start address` marks the beginning and `End address` marks the end of the file for circular journal files. `File type` indicates whether the file is of image or circular journal type.

The separate `File index` field is a remnant from previous versions of the FRAM file system metadata.

Padding bytes have been added automatically by the toolchain.

5.7 I^{2}C, SPI and UART drivers

Both I^{2}C and SPI drivers have a transaction queue and a daemon task that processes transactions and operates the corresponding peripherals.

Transaction structure contains identifiers for target port, device, transaction status

fields and lists of pointers to write and read buffers with their lengths.

For each I^2C transaction, first the contents of write buffers are sent to the target device and then responses are stored in the read buffers. With this design, only one transaction is needed to read one register or a continuous array of registers from an I^2C device.

I^2C driver has been designed on hardware interrupts. I^2C interrupt handlers have been optimized for performance.

For each SPI transaction, first the target device is selected, then the contents of write buffer are sent to the device while reading to the read buffers at the same time. In case the write buffers are empty, zero bytes are sent until the read buffers are full. Once the transaction has been finished, the target device is de-selected.

Due to the hardware design of CDHS, external memory devices are connected via SPI. In order to minimize memory access times, SPI driver uses DMA.

SPI and I^2C transactions have configurable timeouts. Each SPI and I^2C port has a dedicated FreeRTOS timer for aborting the active transaction on timeout. In addition to FreeRTOS timers, there are secondary timeout checks in the driver daemons.

UART drivers are not transactional, because the number of received bytes is not known. There is a daemon, which processes messages that are to be sent. An interrupt is used to transmit a message byte-by-byte and to receive individual bytes. To relay the received bytes, UART driver simply offers a callback function to higher software layers. On CDHS, the callback function is used to fill ICP input buffer. Incoming bytes are checked for ICP frame delimiters. Once a potential ICP frame has been received or the ICP input buffer is full, then CDHS wakes the ICP task for an update.

5.8 Time management

On ESTCube-1, the central time management is performed by EPS, due to hardware design. EPS is the only subsystem with access to a non-interruptible power supply. EPS RTC is synchronized with the ground during satellite operations. CDHS initiates a time synchronization procedure with EPS about 20 seconds after boot-up. The delay of 20 seconds is to provide enough time for the satellite operator to react, in case the communication between CDHS and EPS causes a reset to one of the involved subsystems. At the initialization of time synchronization, CDHS sends a command to EPS over ICP. EPS responds with a future timestamp and triggers a pin when the previously provided timestamp has been reached. CDHS sets its time when the pin is triggered and schedules a re-configuration of its SPI RTC. See Figure 5 for a timeline of this procedure.

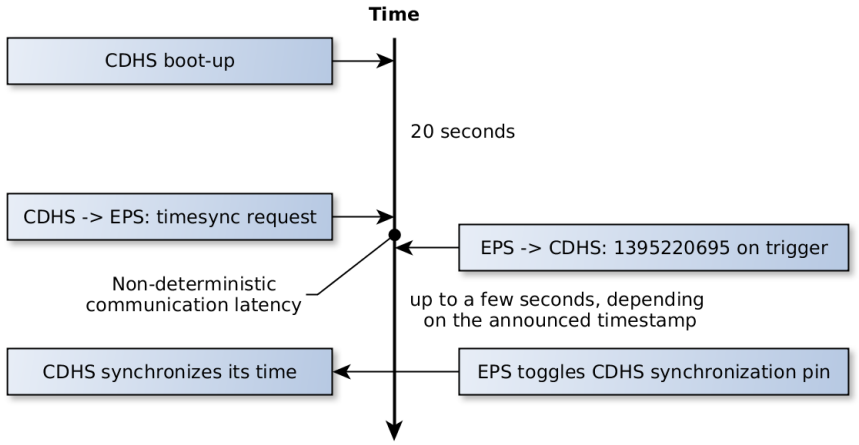

Figure 5: Synchronization of CDHS time from EPS.

On CDHS, SPI RTC is configured to toggle an interrupt pin at the start of each second. CDHS synchronizes its timestamp on this interrupt. This helps to compensate for possible drifts in the microcontroller clock rate.

5.9 ADCS algorithms

ADCS algorithms are called from a dedicated task with a configurable priority and iteration frequency.

In order to save time in the ADCS task, separate tasks have been created for the periodical measurement and pre-processing of gyroscopic sensors and sun sensor ADC's. The results are collected by the ADCS task in a thread-safe way. Only the measurement and pre-processing of magnetometers is called directly from the ADCS task because this makes it easy to guarantee that magnetometers are never measured while the magnetic torquers are active.

The timing of operations in the ADCS task has been organized in such a way that magnetic torquers would be active while the attitude determination and control algorithms are calculating magnetorquer parameters for the next iteration. See Figure 6 on page 36 for an illustration of the timing of the ADCS task at an iteration frequency of $2.5\ \mathrm{Hz}$. Based on attitude controller output from the previous iteration, a command with magnetic torquer Pulse Width Modulation (PWM) values, direction of currents and a timeout is sent to EPS at the start of each ADCS iteration. The timeout parameter is used for automatically deactivating the magnetic torquers on EPS. Due to a possible non-deterministic communication latency between CDHS and EPS, CDHS has a configurable delay ($20\ \mathrm{ms}$ by default) before it notifies EPS to activate the torquers by toggling a pin. During the delay, CDHS performs magnetometer measurements and pre-processing. See Figure 7 on page 36 for a timeline.

At the time of writing, EPS still limits magnetic torquer timeouts to $255\ \mathrm{ms}$ so that at an iteration frequency of $2.5\ \mathrm{Hz}$, magnetic torquers are active for only 63% of the iteration. With higher iteration frequencies, the percentage is improved. However, the more often magnetic torquer control commands are sent to EPS, the greater the risk of EPS encountering a sporadic reset. An EPS reset also causes a CDHS power cycle, which stops attitude control. With an upcoming revision of EPS firmware, the maximum magnetic torquer timeout value shall be increased.

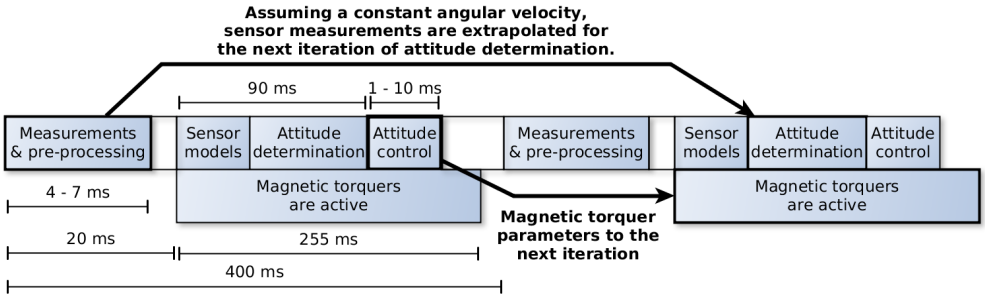

Figure 6: Timing of the ADCS task at 2.5 Hz iteration frequency.

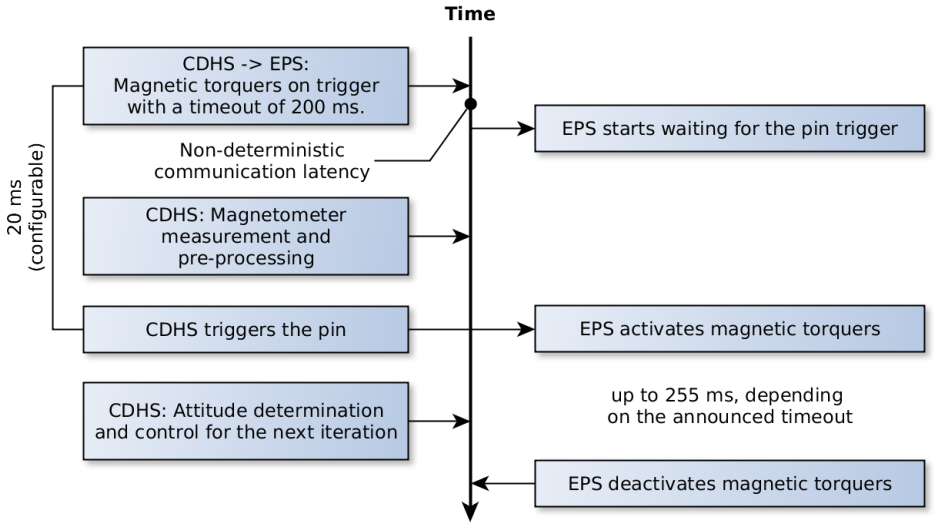

Figure 7: Timeline of magnetic torquer control.

In-orbit measurements of attitude determination and spin-up controller with the CDHS microcontroller running at $32\ \mathrm{MHz}$ indicate a calculation time of roughly $110\ \mathrm{ms}$ per iteration. This would allow for an iteration frequency of at least $8\ \mathrm{Hz}$.

5.10 Telemetry logging

Command scheduler support for higher-order commands is used to enable the logging of command responses. CDHS provides commands that store the output of the commands supplied as an argument. The target memory device and file

can either be configured for all subsequent logging calls or for each logging call separately. CDHS also has commands for configuring a set of filters, which allow for specific commands or response packets from other subsystems to be logged.

Regular housekeeping data is logged into a file where the structure of each entry is fixed, thus allowing for random-access by time. The support for storing the responses of any commands is provided at the expense of losing random-access to individual entries because the lengths of journal entries differ. For the lack of a better name, the journal files for logging the output of commands are called "Special" journals.

In order to be able to reliably read the contents of a journal in flash file system, the length of the next journal entry must be known. For special journals, this is not the case. To avoid issues with misaligned file reads, special journals are down-linked as raw memory regions instead of files because alignment does not apply to raw memory reads.

Each special journal entry starts with a timestamp of when the entry was logged. Following the 4 B timestamp, each entry has a response header that follows the CDHS command structure format (See Section 5.4) and response arguments. The structure of response arguments is determined by both the `identifier` and `length of arguments` fields in the response header. See Figure 8 on page 38 for an example.

5.11 Telemetry buffering

In order to provide support for down-link packets with unique identifiers as well as for re-sending specific packets on request, a telemetry buffer module was designed and implemented. Each packet targeted to Ground Station (GS) is stored as an entry in the telemetry buffer, which is a circular journal file. Each entry contains a 3 B sequence counter as a unique identifier, packet source subsystem index, packet type, packet length and a unix timestamp. The first packet is sent to COM after being stored in the buffer. The next packet is sent when a ready-to-send packet is

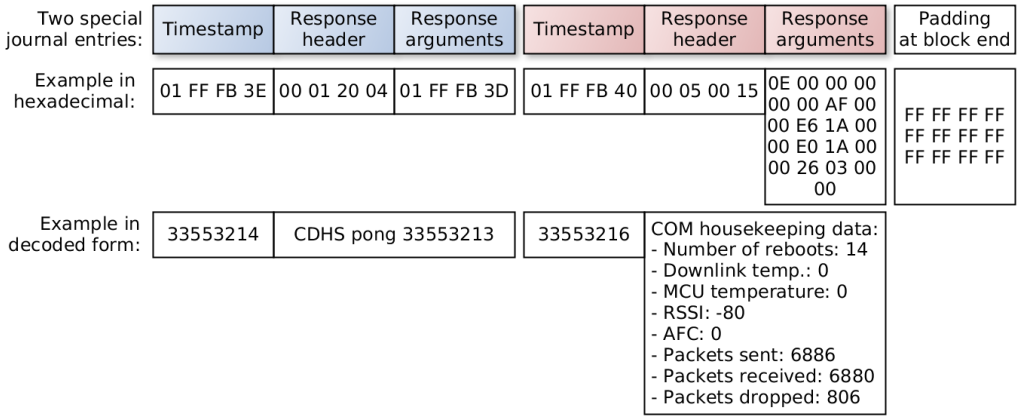

Figure 8: An example structure of a special journal with two entries: a response to a CDHS ping command (12 B) and a housekeeping response from COM (29 B).

received from COM. In case COM does not indicate that it is waiting for the next packet to send to the ground, the next packet is sent after a configurable timeout has been reached. With each sent packet, the read address of the telemetry buffer is iterated.

At first, a 1 MB flash journal file was used for telemetry buffer contents. Telemetry buffer header was stored in a file in the system FRAM. Due to frequent write and erase calls, the buffer file was moved to FRAM and shrunk to 100 KB.

Commands were implemented to request for the status of the telemetry buffer and to request for a list of packets to be re-sent. Telemetry buffering can be enabled on request.

Contrary to our expectations, telemetry buffering does not improve the efficiency of our data down-link. Although it allows for several down-link requests to be grouped into a single packet, the timing of transmitted packets is flawed. Not always does COM send a ready-to-send packet after it has finished transmitting the previous one, which often causes timeout delays between packets. Moreover, reducing the timeout only caused COM packet buffer to become full more often.

The feature of requesting for the re-transmission of packets has not been used, since ICPTerminal extensions have been designed to automatically re-send the

requests, for which the responses are missing. In order to monitor time variant parameters, the use of on-board logging is preferred.

Due to intermittent timing issues, the telemetry buffer read address occasionally leaps to the start of the buffer, so that CDHS starts re-sending old packets and has to be stopped manually. It can be stopped either by disabling telemetry buffering, resetting the buffer or performing a CDHS reboot.

5.12 Beacon

CDHS can be configured to periodically collect telemetry from CAM, COM, EPS, ADCS and itself. It can then be configured to use the telemetry for sending it as packet beacon or EPS normal mode beacon, which is Continuous Wave (CW) beacon.

When beacon is enabled, CDHS asks EPS for their beacon status. EPS beacon status indicates if EPS is running in debug or normal mode, as well as provides the time since the last beacon transmission. In debug mode, EPS periodically transmits their own beacon. With the beacon status, CDHS avoids interrupting EPS while another beacon is being transmitted. CDHS also checks EPS guardian state and determines the period at which it should request for an EPS normal mode beacon. All of the aforementioned steps can be disabled with CDHS configuration.

Since intermittent stability issues with EPS were detected in-orbit when beacon was enabled for more than three days, telemetry gathering and beacon transmission are now disabled by default.

5.13 Hardware control

ADCS magnetometers, gyroscopic sensors and sun sensor ADC's are controlled over I^2C or SPI with dedicated device drivers, which have been written by ADCS.

Magnetic torquers are controlled by sending EPS an ICP packet and triggering a pin. Similar to the time synchronization procedure in Section 5.8, a pin trigger is used to alleviate the issue of non-deterministic delays in ICP communication.

The ADC of payload high-voltage supply is controlled over SPI with the ADC122S device driver. Payload locks are controlled by sending specific ICP packets to EPS. Reel motor controller and high-voltage supply control lines are controlled via CDHS pins. Software checks are used to make sure that both high-voltage supply polarity control lines are not pulled high at the same time. To mitigate the problem of missing feedback from the tether reel motor controller, CDHS assumes a constant reeling speed and provides a command to reel for a specified number of milliseconds. With a reeling time of more than $100\ \mathrm{ms}$, the ratio between reeling time and the amount of tether reeled out was seen to be almost linear.

Payload hardware control has not been tested in orbit yet. This will be performed once the satellite has been reliably spun up to $1\ \mathrm{rps}$.

5.14 Firmware upgrade

CDHS firmware is uploaded to a $256\ \mathrm{KB}$ SPI FRAM, a $128\ \mathrm{B}$ page at a time. CDHS maintains a pagemap, where each bit marks a page. High bit values indicate pages that have been successfully received and low bit values indicate missed pages.

Firmware pages would be transmitted from the ground while periodically requesting firmware pagemap for confirmation. Missing pages are automatically re-sent to the satellite. Once the whole firmware image has been uploaded, a CRC-32 checksum is calculated and compared against the checksum in the firmware image header. CRC-32 was chosen because the STM32F1 microcontroller has a built-in peripheral for calculating CRC-32 checksum.

After a successful verification of the integrity of the firmware image in the FRAM, commands are scheduled for the CDHS bootloader to copy the image into flash. A

soft-reboot is performed, during which, the bootloader performs the scheduled operations and logs its progress into the microcontroller internal flash. Both CDHS microcontrollers have two firmware image slots. In case the boot-up sequence to one of the firmware images would fail, the bootloader would try to boot to the other firmware image. In case both firmware images have been corrupted, CDHS microcontrollers would be switched via EPS. [16, 3]

Both CDHS microcontrollers have $768\ \mathrm{KB}$ of flash memory. Two $300\ \mathrm{KB}$ slots have been reserved for firmware images. In case of $256\ \mathrm{KB}$ firmware images, this allows for $44\ \mathrm{KB}$ of firmware-specific configuration as well as $168\ \mathrm{KB}$ for global configuration and look-up tables. Although $256\ \mathrm{KB}$ was thought to be more than enough, our release builds have already reached $230\ \mathrm{KB}$. See Figure 9 on page 42 for the distribution of memory addresses for firmware images. For STM32F1 microcontrollers, the interrupt vector table must be aligned to a $512\ \mathrm{B}$ boundary. Because of this, a padding section was added after each firmware image header.

Firmware image header consists of three $32\ \mathrm{bit}$ words:

1. Firmware image size in Bytes, without header and padding.
2. Firmware version in hexadecimal format.
3. CRC-32 checksum of the firmware image.

Due to linker issues with generating position independent code for bare-metal Cortex-M3, firmware for each slot has to be separately linked and uploaded to the satellite.

5.15 Low-level access

CDHS provides a command for reading microcontroller memory, registers or the contents of any external device that is connected via I^2C or SPI. Requirements for the on-board file systems were set so that files could be read either by using file access commands or by reading the raw memory contents at a specific address.

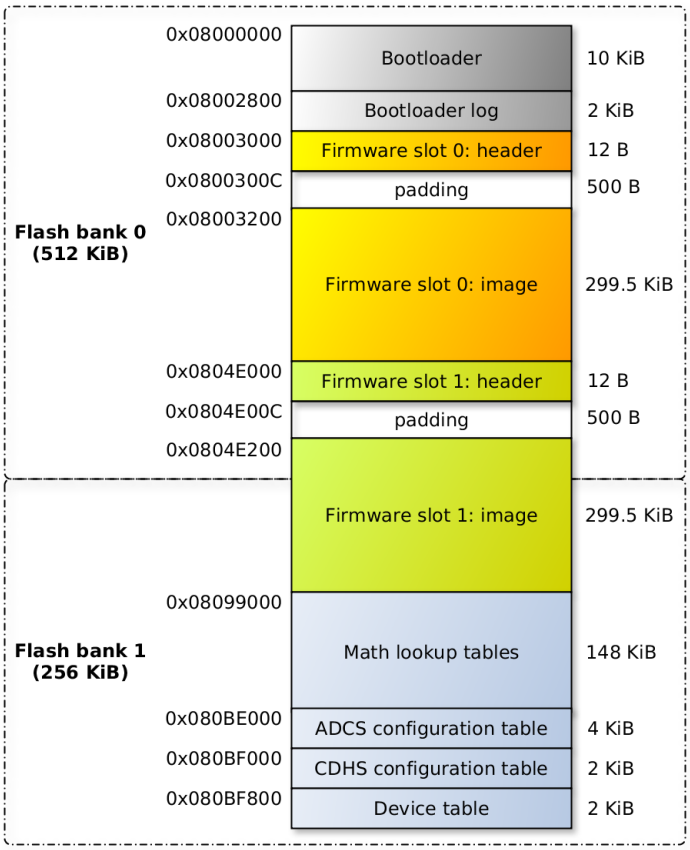

Figure 9: Distribution of microcontroller flash memory on CDHS.

A command for calculating the CRC-32 checksum of on-board memory, registers or external memory contents can be used to check for changes in large memory regions without the need for downloading them.

In addition, CDHS has a command for reading microcontroller pin values. There are commands for requesting microcontroller peripheral frequencies, amount of free heap left, for measuring the execution time of any other commands or for recording the status of FreeRTOS tasks.

CDHS also allows for sending raw ICP packets to other subsystems, or to emulate forwarded packets. For example, CDHS can be used to send a request to EPS so that EPS would think the packet came from the Ground Station (GS) and would send the response to the ground.

Once the execution of CDHS debug commands has been enabled, low-level manipulation commands can be used. These allow for setting microcontroller pin values, writing to internal memory regions, registers or any external device connected via I^2C or SPI.

A python script is used to parse linker map files of built firmware images for the indexes of dynamic configuration table entries as well as for the memory addresses of global variables. The script produces a python module for ICPTerminal, which allows for accessing on-board variables by name.

5.16 Configuration tables

In the flash memory of both CDHS microcontrollers, one $2\ \mathrm{KB}$ block has been reserved for a table of CDHS configuration variables. An example of CDHS configuration variables: microcontroller clock frequency, start-up flags, loading sequence of external memory devices, task priorities, task stack margins, queue lengths, etc. Two $2\ \mathrm{KB}$ blocks have been reserved for ADCS configuration variables such as Two-Line Element set (TLE), gain for satellite de-tumbling controller or Kalman filter covariances for magnetometers and gyroscopic sensors, etc. See Figure 9 on page 42 for the distribution of memory addresses for configuration tables.

On boot-up, CDHS configuration table is loaded into RAM from flash. In RAM, configuration variables are checked against limits before use. Configuration variables in RAM can be updated from the ground and written back to flash to make changes non-volatile.

In case at least 6 CDHS resets have occurred so that CDHS has not sent any responses back to the ground, fall-back configuration is used. In fall-back configuration, all I^2C and SPI devices are disabled, only the most basic on-board functionality is left enabled, allowing for operators to manually initialize devices and resolve possible issues.

Configuration variables can be accessed either by their indexes or by their memory addresses. CDHS configuration variables have been grouped into a single structure, thanks to which, variable indexes can be kept constant across firmware versions. Since ADCS configuration variables are scattered throughout several libraries, variable indexes change as the linker changes the ordering of the variables within the region.

5.17 On-board scripting

Scripting allows for flexibility at a minimal cost of microcontroller code memory.

Pawn [17], formerly known as Small, scripting language was chosen for its minimal flash, RAM requirements and support for embedded systems. See Table 4 on page 45 for a comparison of a few scripting languages that have been used on embedded systems with low resources. Customized versions of Pawn are used for scripting in game industry [1], which offers active communities for discussions related to the Pawn language.

With all the necessary wrappers for floating point arithmetic calculations, string operations, timers, binary structures, command interface and logging, the customized Pawn uses about $18\ \mathrm{KB}$ of flash on CDHS. The stack size is configurable for each Pawn script and Pawn features overlays, which allow for the execution of scripts larger than the amount of RAM that is available on the system. On CDHS, scripts are typically run with 256 words of stack and $2\ \mathrm{KB}$ overlays, which totals to a RAM footprint of about $4\ \mathrm{KB}$. The length of a Pawn word (also called a Pawn cell) is configured as $32\ \mathrm{bits}$.

[1] According to Wikipedia, Pawn is used in the San Andreas Multiplayer mod, Half-Life mod, AMX Mod X and Source Engine based SourceMod as well as in other projects.

Table 4: Comparison of scripting languages.

Name	Minimum Flash footprint[1]	Minimum RAM footprint[1]	License	Ported to
Pawn	10 KB	2 KB	free, Apache 2	ARM7[2]
Squirrel	100 KB	100 KB	free, MIT	
eLua	128 KB	32 KB	free, MIT	Cortex-M3[3]
PyMite	64 KB	8 KB	GPL 2	

[1] The presented footprints are rough estimates. Only the footprints of Pawn were measured on CDHS.

[2] The ARM7 port is not usable for Cortex-M3. However, Pawn also has a less optimized version that is cross-platform.

[3] Officially ported to Cortex-M3, which is used on CDHS.

5.18 Optimizations

In order to reduce the flash footprint of the firmware image, rarely used functionality was removed and software modules were simplified. Standard functions for memory and string operations were rewritten for minimizing flash footprint. See Listing 1 for a list of compiler flags and Listing 2 for a list of linker flags used for the release build of CDHS firmware images.

Listing 1: Compiler flags

```
-Os -fshort-enums -ffunction-sections -fdata-sections
-fno-reorder-blocks -fno-reorder-functions -fno-strict-aliasing
-fno-peephole2 -fno-delete-null-pointer-checks
```

Listing 2: Linker flags

```
-nostartfiles -nodefaultlibs -nostdlib -Xlinker --gc-sections
```

In order to improve the performance of sine and cosine functions, CDHS generates a 148 KB look-up table. See Figure 9 on page 42 for an overview of CDHS microcontroller flash memory layout.

6 In-orbit performance

ESTCube-1 has been operational in-orbit for slightly more than a year. In order to save battery life, CDHS microcontroller is down-clocked to 32 MHz and configured to sleep on idle. By the time of writing, there have been about 12 successful in-orbit firmware upgrades to CDHS.

A list of issues encountered so far:

- Most likely an effect of ionizing radiaton:
    1. One of the firmware image slots is damaged on one of the two microcontrollers (microcontroller A).
    2. On its own, CDHS has entered fallback mode twice.
- Deviation in memory access times:
    1. False positive bad bytes and bad blocks in the flash file system.
- The cause is unknown:
    1. A few anomalous bytes in on-board logs. Bytes `2C 20 04` instead of `6C 20 7C`.
    2. Occasional loss of on-board statistics and script files.

The transmission of packet beacon as well as EPS normal mode beacon have been rarely used due to intermittent stability issues with EPS.

The feature for logging housekeeping telemetry has been rarely used, as it does not provide enough information for attitude determination and control tests. Housekeeping log is inefficient for sampling a few parameters at a higher frequency. The "special" on-board logging functionality was introduced to resolve the issue.

The "special" on-board logging as well as file-transfer have been used for storing raw sensor measurements over several orbits, for recording the current consumption of satellite subsystems, monitoring satellite attitude, etc.

Commands have been scheduled for being executed at specific times to take camera images of the North and South poles, United States, Africa and several other regions.

Attitude determination and several attitude control algorithms have been successfully run in-orbit:

1. Attitude determination with camera images for verification.
2. Detumbling to minimize changes in the magnetic field measurements.
3. Pointing to direct the on-board camera towards a target in-orbit or on the ground.
4. Spin-up to increase angular velocity around satellite Z axis.
5. Spin-up to increase angular velocity around a pre-configured axis.

The on-board scripting functionality has been used for the following:

1. Transmitting telemetry packets of gratitude to radio amateurs for their help in receiving satellite Continuous Wave (CW) beacon and packet telemetry.
2. Testing magnetic torquers and measuring their effect with magnetometers, gyroscopic sensors and sun sensors.
3. Satellite demagnetisation with magnetic torquers.
4. Running pointing controller and taking camera images when the pointing error drops below a configured threshold.

7 Further improvements

While avoiding the logging of consecutive repetitive errors helps to use the error log more efficiently, it becomes a problem when the timestamps of repetitive errors are of interest. During ADCS tests, there is a tendency for errors to repeat in

cycles. Error logging could be improved to detect and filter out repetitive cycles. Instead of storing absolute timestamps for all error log entries, relative timestamps against a single absolute timestamp could be used.

Although the requirements for flash file system had been set in such a way that both read and write access times would be minimal, fast read access has not proven necessary. By sacrificing random access on flash file reads, the file system could be designed upon a tree structure. This would make the detection and recovery of corrupt bytes easier and more reliable.

Flash file system write and erase times are a limiting factor in CDHS data logging. On a group of flash memory devices and FRAMs, something similar to Redundant Array of Independent Disks 0 (RAID 0) could be implemented to speed up file system write calls. However, this would increase the risk of losing data due to a malfunctioning memory device. The size of the bad block table in flash file system metadata can be reduced from $64\ \mathrm{B}$ to $8\ \mathrm{B}$ by storing a bitmap instead of a list of indexes.

The speed of Pawn script abstract machine can be improved by implementing an assembler optimized version for Cortex-M3 microcontrollers.

With a more recent version of FreeRTOS, microcontroller sleep on tickless idle can be implemented. This might reduce CDHS current consumption further. By implementing dynamic runtime clocking of CDHS microcontroller and memory devices, the average current consumption could be decreased even more.

Currently CDHS only allows satellite operators to log telemetry as command responses. However, often a single command contains too much overhead or only a few parameters are needed from many different commands. Could design and implement a system that would allow requests for configurable sets of parameters. This would increase the efficiency of both data logging and telemetry downloads.

8 Conclusion

Requirements for the on-board software of ESTCube-1 Command and Data Handling System (CDHS) were gathered. Accounting for the requirements, the on-board software for CDHS was designed and implemented. The on-board software contains FreeRTOS drivers for data buses and on-board devices, error handling, command scheduler, telemetry logging, file systems for serial FRAM and flash memory devices. Several of the developed modules are also being used on the camera system of ESTCube-1.

Tests have been performed on the software on two hardware models on the ground as well as on the satellite in orbit. With the exception of a few, the issues encountered during in-orbit testing have been successfully reproduced on the lab clone of the satellite, resolved, after which, the updated CDHS firmware images have been successfully uploaded to the satellite in orbit.

On-board CDHS, several in-orbit tests have been carried out on attitude determination and control software. The pre-processing algorithms for sensor measurements have been verified, the output of on-board attitude determination has been compared to photos from the on-board camera. Detumbling, pointing and two spin-up controllers have been run successfully on CDHS in orbit. With the pointing controller and with the help of CDHS on-board scripting, the secondary mission of taking a photo of Estonia from space has been fulfilled.

ESTCube-1 has been in orbit for slightly more than a year now and all the systems are still operational.

9 Kokkuvõte

Antud töö raames sai loetletud ESTCube-1 Käsu- ja Andmehaldussüsteemile ehk pardaarvutile esitatud nõuded. Vastavalt nõuetele sai arendatud pardaarvuti tarkvara, mis sisaldab FreeRTOS ajureid andmesiinide ja pardaseadmete jaoks, veahaldust, käsuhaldurit, moodulit telemeetria salvestamiseks ning failisüsteeme jadaliidesega ferroelektriliste muutmälude ja välkmälude jaoks. Mitmed arendatud tarkvaramoodulitest on leidnud kasutust ka ESTCube-1 kaamerasüsteemi pardal.

Arendatud tarkvaral on sooritatud teste kahel satelliidi maapealsel mudelil ning orbiidil lendaval satelliidil. Mõningate eranditega on orbiidil täheldatud probleemid edukalt reprodutseeritud maapealsetel mudelitel, ning uus parandustega versioon tarkvarast on edukalt orbiidil olevale satelliidile laetud.

Orbiidil on pardaarvutil edukalt katsetatud satelliidi orientatsiooni määramise ja juhtimise tarkvara. Andurite mõõdiste eeltöötluse algoritmid on orbiidil testitud ning satelliidi orientatsiooni määramise algoritmi väljundit on võrreldud pardakaamera piltidega. Kasutades satelliidi osutamise algoritmi koos pardaarvuti skriptidega, on täidetud ka osa ESTCube-1 missioonist - pildistada Eestit kosmosest.

Seni on ESTCube-1 olnud orbiidil veidi üle aasta ning kõik satelliidi süsteemid on endiselt töökorras.

10 Acknowledgements

I would like to thank

- Kaspars Laizāns for the design and assembly of the final version of CDHS hardware so that I could focus on the software,
- Henri Kuuste for reviewing my code and providing advice on software architecture,
- Martin Valgur for developing ICP and implementing the core of ICPTerminal,
- Erik Kulu for operating the satellite $24/7$ and performing in-orbit testing
- and everyone else who have been working on the ESTCube-1 project.

I would like to thank both Kārlis Zālīte from the Tartu Observatory Synthetic Aperture Radar (SAR) team as well as Mart Noorma for supervising this thesis.

References

[1] A. Slavinskis, U. Kvell, M. Pajusalu, H. Kuuste, I. Sünter, E. Ilbis, T. Eenmäe, K. Laizāns, A. Vahter, E. Eilonen, J. Kalde, P. Liias, A. Sisask, L. Kimmel, V. Allik, S. Lätt, and M. Noorma, "Estcube-1 nanosatellite for electric solar wind sail demonstration in low earth orbit," in "64^{th} International Astronautical Congress," (2013).

[2] A. Slavinskis, U. Kvell, E. Kulu, I. Sünter, H. Kuuste, S. Lätt, K. Voormansik, and M. Noorma, "High spin rate magnetic controller for nanosatellites," Acta Astronautica (2014).

[3] K. Laizāns, I. Sünter, K. Zālīte, H. Kuuste, M. Valgur, K. Tarbe, V. Allik, G. Olentšenko, P. Laes, S. Lätt, and M. Noorma, "The design of fault tolerant command and data handling subsystem for estcube-1," Proceedings of the Estonian Academy of Sciences pp. 222–231 (2014).

[4] N. Wrachien, "Advanced memories to overcome the flash memory weaknesses: a radiation viewpoint reliability study," Ph.D. thesis, University of Padova (2010).

[5] A. Slavinskis, E. Kulu, J. Viru, R. Valner, H. Ehrpais, T. Uiboupin, M. Järve, E. Soolo, J. Envall, T. Scheffller, I. Sünter, H. Kuuste, U. Kvell, J. Kalde, K. Laizāns, E. Ilbis, T. Eenmäe, R. Vendt, K. Voormansik, I. Ansko, V. Allik, S. Lätt, and M. Noorma, "Attitude determination and control for centrifugal tether deployment on the estcube-1 nanosatellite," Proceedings of the Estonian Academy of Sciences pp. 242–249 (2014).

[6] T. Ilves, "Estcube-1 electrical power system operation software," Master's thesis, university of tartu (2013).

[7] M. Schmidt and K. Schilling, "An extensible on-board data handling software platform for pico satellites," Acta Astronautica (2008).

[8] L. L. Moulin, "Swisscube integration, launch and operational activities," Master's thesis, HES-SO Valais-Wallis (2009).

[9] D. E. Holmstrøm, "Software and software architecture for a student satellite," Tech. rep., Norwegian University of Science and Technology (NTNU) (2012).

[10] K. A. Ødegaard, "Error detection and correction for low-cost nano satellites," Master's thesis, Norwegian University of Science and Technology (NTNU) (2013).

[11] C. P. Bridges, S. Kenyon, P. Shaw, E. Simons, L. Visagie, T. Theodorou, B. Yeomans, J. Parsons, V. Lappas, C. Underwood, S. Jason, D. Mellor, N. Navarathinam, P. Wellstead, A. Schofield, R. Linehan, J. Barrera-Ars, B. Dyer, D. Liddle, and M. N. Sweeting, "A baptism of fire: The strand-1 nanosatellite," in "27^{th} Annual AIAA/USU Conference on Small Satellites," (2013).

[12] L. Wood, W. M. Eddy, W. Ivancic, J. McKim, and C. Jackson, "Saratoga: a delay-tolerant networking convergence layer with efficient link utilization," in "Delay Tolerant Networking session, Third International Workshop on Satellite and Space Communications," (2007).

[13] S. Kenyon, C. P. Bridges, D. Liddle, B. Dyer, J. Parsons, M. Pollard, D. Feltham, R. Taylor, D. Mellor, A. Schofield, R. Linehan, R. Long, J. Fernandez, H. Kadhem, P. Davies, J. Gebbie, N. Holt, P. Shaw, L. Visagie, T. Theodorou, V. Lappas, and C. Underwood, "Strand-1: Use of a $500 smartphone as the central avionics of a nanosatellite," in "62^{nd} International Astronautical Congress," (2011).

[14] R. Barry, "Freertos," (2014). `http://www.freertos.org/RTOS.html`.

[15] T. Duff, "Description of duff's device," (1988). `http://www.lysator.liu.se/c/duffs-device.html`.

[16] K. Tarbe, “Bootloader for estcube-1 command and data handling system and camera module,” Bachelor thesis, University of Tartu (2013).

[17] T. Riemersma, “Pawn - an embedded scripting language,” (2014). http://www.compuphase.com/pawn/pawn.htm.

11 Appendix

11.1 Development and testing

The software development and testing of CDHS software is performed in three or four stages, depending on the complexity of the performed software modifications.

11.1.1 Simulation

Larger software modules, such as ICP, Flash and FRAM file systems, attitude determination and control algorithms are simulated before they are integrated into the CDHS firmware. The software modules are written to be platform independent so that they can be first developed and tested on a personal computer and then used on the real hardware with a minimum of testing.

In order to develop and simulate attitude determination and control algorithms, the ADCS team uses Matlab with Simulink. In order to develop and simulate ICP, custom applications have been written for running unit tests and simulating the communication between several subsystems. A custom application has also been developed for simulating Flash and FRAM memory devices and running tests on the file systems.

At this stage, unit tests, functionality tests are run and specific failure scenarios are tested.

11.1.2 Firmware integration

Software developed and tested on a personal computer is integrated with the rest of CDHS firmware. Eclipse CDT is used to write the code, Mentor Graphics CodeSourcery Lite toolchain is used to compile and link the firmware image. Firmware image header and padding are added with a custom python script. See Section 5.14 for a description of the firmware image header structure.

Two methods are used for updating the firmware image on CDHS engineering model. Occasionally, the firmware is uploaded through an USB serial connection with ICPTerminal. See Section 5.14 for a detailed procedure. Most often, the CDHS engineering model (EM) is manually reset into the built-in bootloader of STM32F1 and the firmware is uploaded through an USB serial connection with an application that follows the STM32 bootloader protocol. Although with both methods, a firmware upgrade takes a few minutes, the latter method is quicker.

Tests on the CDHS engineering model are performed manually by sending requests and analysing replies with ICPTerminal. See Figure 11 on page 57 for a screenshot of ICPTerminal in action. See Figure 10 on page 56 for an image of the CDHS engineering model with an additional USB serial converter attached.

Hardware in the loop tests were performed for testing the integration of attitude determination and control algorithms. Input data was taken from the simulation environment and fed to the engineering model. Calculations were performed onboard and results were compared with those from simulations.

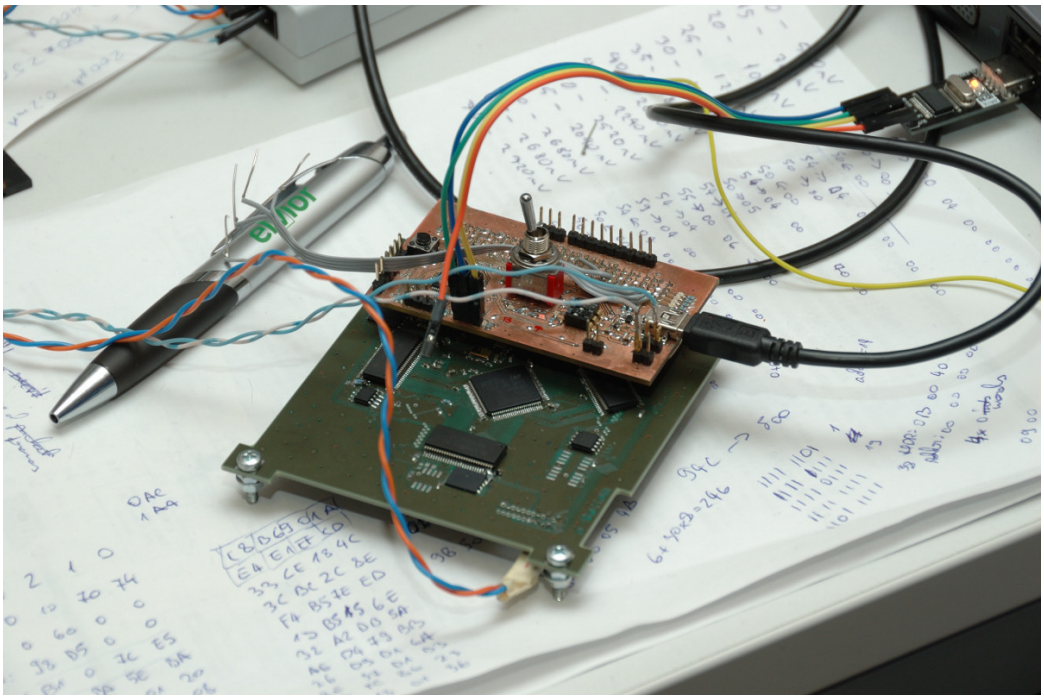

Figure 10: Testing on CDHS engineering model.

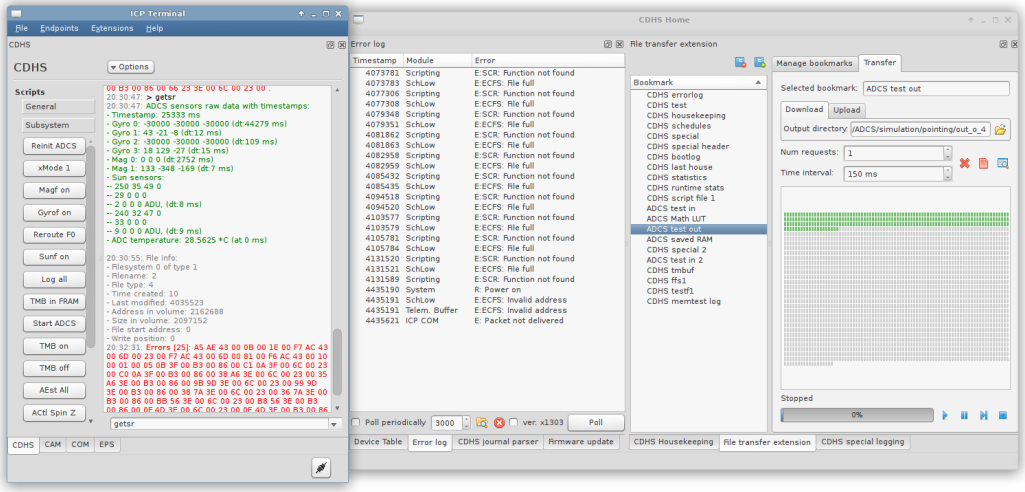

Figure 11: ICPTerminal connected to the CDHS and ADCS engineering model.

11.1.3 Subsystem integration

Firmware tested on-board an engineering model is uploaded to the lab clone of the satellite (also called table model or TM), where inter-subsystem functionality and performance are tested. See Figure 12 on page 58 for an image of the table model, connected to a solar panel simulator and a programmable power supply. Magnetic torquers are taped onto a plastic structure. A CDHS firmware upgrade to the table model takes about 1 h.

11.1.4 Qualification testing

Final functionality tests as well as software qualification tests are performed by satellite operators on ESTCube-1 that is in orbit (also called flight model or FM). See Figure 13 on page 59 for an image of the flight model at a cleanroom in Kourou in French Guiana. A CDHS firmware upgrade takes at least a day, depending on the up-link packet loss.

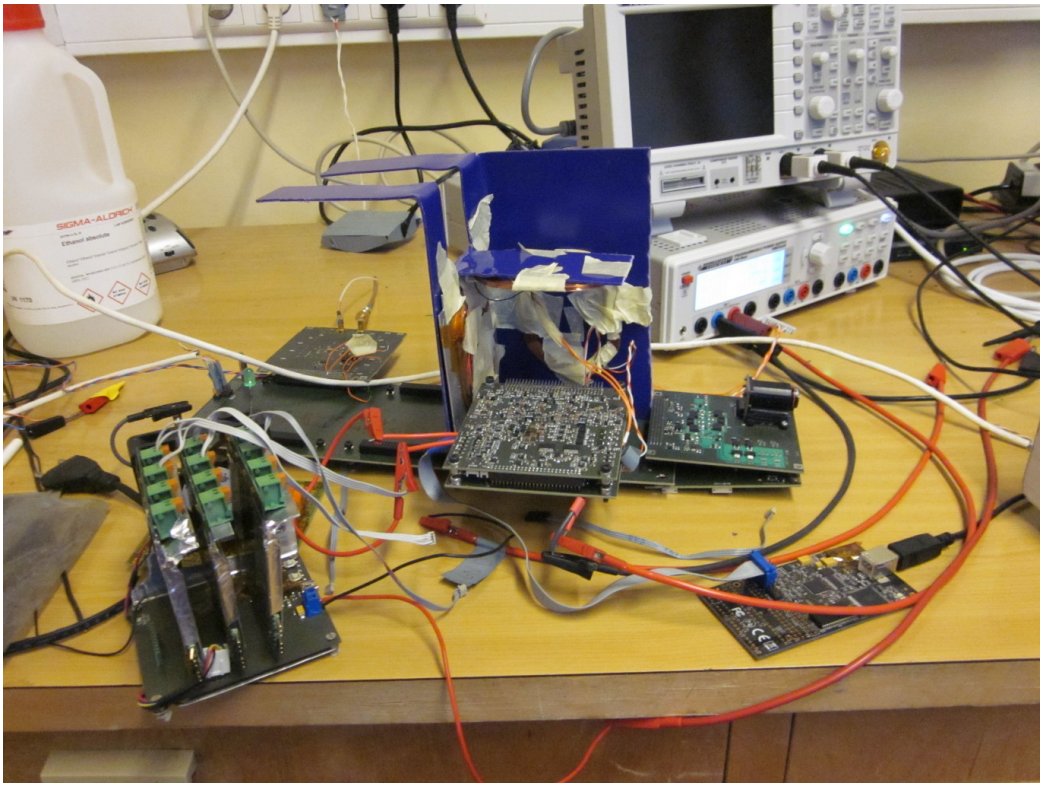

Figure 12: Testing on the table model.

11.2 ICPTerminal

ICPTerminal is a modular Python application that Martin Valgur has developed specifically for ICP communication with hardware, following the CDHS command structure. ICPTerminal uses the same C library for ICP that also runs onboard satellite subsystems. By design, ICPTerminal easily allows for creating extension modules for parsing error logs, upgrading firmware, performing file transfer with the satellite, downloading and processing camera images, etc. On startup, ICPTerminal loads Extensible Markup Language (XML) files with ICP endpoints on the satellite, the descriptions of satellite command and reply packets and extension modules. For each subsystem, a separate terminal-dock frame is created with a communication log and a text box for entering commands in text form. See Figure 11 on page 57 for a screenshot of ICPTerminal.

Although ICPTerminal had been intended for testing purposes only, it is currently

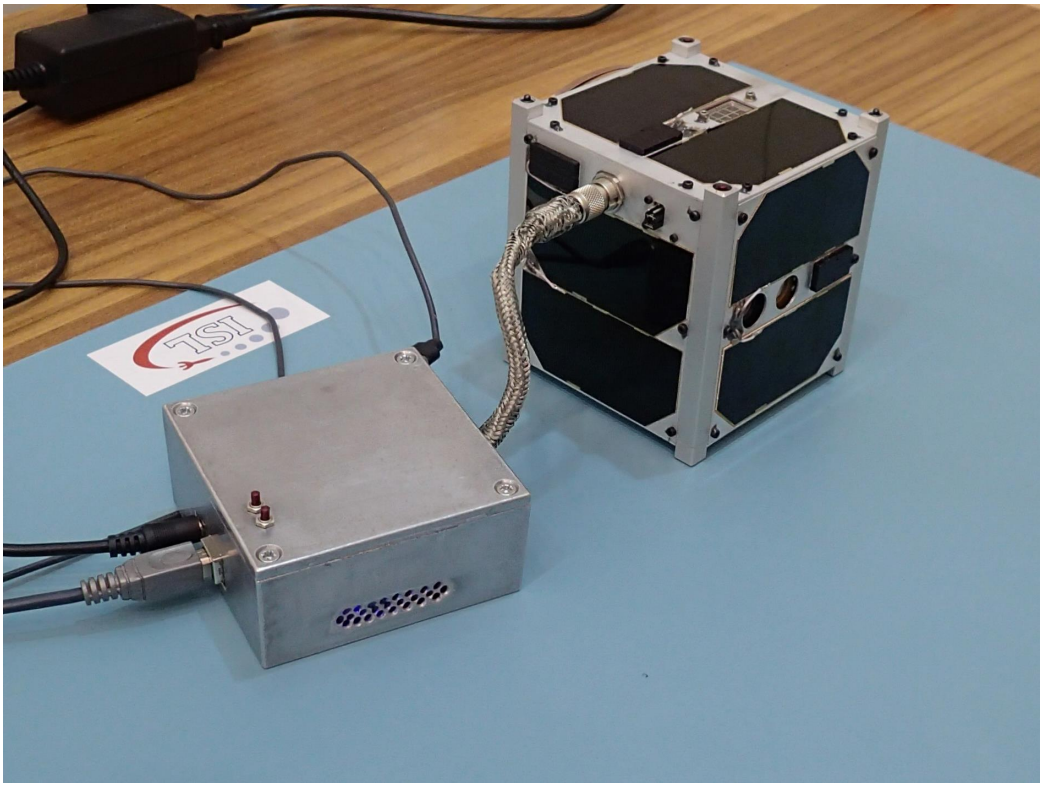

Figure 13: Firmware upgrade on ESTCube-1 flight model at a cleanroom in Kourou in French Guiana.

still being used for operating ESTCube-1.

For file transfer with ESTCube-1 CDHS, an extension module was written. The file transfer extension allows the operator to add so called bookmarks, which describe either on-board files, raw memory regions or registers. File bookmarks contain at least the following parameters: target file name (name of the file that is used to store the data), file system type, file system index, file index. Raw memory bookmarks specify target file name, device type, device index, memory address and length. When a bookmark is selected for downloading, an empty file is created for storing content and a pagemap file is created for marking successfully received pages. For missing pages, the extension re-sends the read commands. In the case of uploading files to the satellite, CDHS maintains a pagemap on-board. The pagemap would be downloaded and checked for missing pages, which would

then be re-sent to the satellite.

An ICPTerminal extension was written, which would parse down-linked binary files and convert them to Comma Separated Values (CSV) that could be imported to Matlab, GNumeric or any custom utilities.

An extension was written for executing state-machine scripts that allow for automating routine satellite operation procedures, synchronize satellite time, update TLE.

The core of ICPTerminal was modified to add support for higher order commands as well as for sending multiple commands per packet. See Listing 3 for an example ICPTerminal command that starts periodic logging of specific command responses and writes the output into an explicitly defined file. The command requests for attitude (`getattitude`), output of sensor measurements with pre-processed results (`getgf`) and last magnetic torquer parameters sent to EPS (`getlastcoils`). The output of the commands is written to a special journal with a file index of 4 in the first SPI flash memory (file system of type 1, file system index 0). The procedure is repeated once every second (1000 ms), until the operator stops it. This example relies heavily on the CDHS command scheduler support for higher order commands.

Listing 3: An ICPTerminal command to periodically log the responses of a few commands.

```
iloop 1000 {ilog2 1 0 4 {getattitude;getgf;getlastcoils}}
```

11.3 Content of the USB memory stick

Table 5: Contents of the accompanying USB memory stick.

M2014_Synter.pdf	A copy of the thesis
exp2014_Synter.pdf	Report on practical experiences in computer engineering. Describes the design and implementation of new features for ICPTerminal. The report is in Estonian only.
source_code	Source code of CDHS firmware

11.4 External memory contents

Table 6: Table of reserved files on the system FRAM (file system of type 0, index 1).

File name	File index	File type	File size	Description
test	0	4 (journal)	50 B	Reserved for file system read / write tests.
fwp-fram	1	2 (image)	300 B	Pagemap for temporary firmware storage.
fwp-slot0	2	2 (image)	300 B	Pagemap for firmware image slot 0.
fwp-slot1	3	2 (image)	300 B	Pagemap for firmware image slot 1.
fwp-bootldr	4	2 (image)	300 B	Pagemap for bootloader image.
error-log	5	2 (image)	800 B	Log of CDHS errors.
schedules	6	2 (image)	8 KB	Reserved for command scheduler. A list of commands executed on startup.
ffs0	7	2 (image)	15 KB	Metadata for flash file system 0.
ffs1	8	2 (image)	15 KB	Metadata for flash file system 1.
ffs2	9	2 (image)	15 KB	Metadata for flash file system 2.
special-hdr	10	2 (image)	256 B	Metadata for the special journal. A list of configurable filters for logging telemetry from other subsystems.
tmbuf-meta	11	2 (image)	64 B	Metadata for the telemetry buffer. Telemetry buffer state, packet identifier and address of the next packet to be read.
statistics	12	2 (image)	2 KB	Statistics file with file system read / write access times and errors grouped by software modules. Updated periodically on demand.
rt-stats	13	2 (image)	2 KB	Statistics on FreeRTOS tasks. Status, priority, microcontroller time, microcontroller percentage and amount of free stack for each task. Updated once on demand.
file-pmap	14	2 (image)	300 B	Pagemap for file uploads. Only one file at a time.

Table 7: Table of reserved files on the second SPI FRAM (file system of type 0, index 2).

File name	File index	File type	File size	Description
test	0	4 (journal)	50 B	Reserved for file system read / write tests.
tm-buffer	1	4 (journal)	100 KB	Buffer for telemetry packets to be downlinked via COM.

Table 8: Table of reserved files on the first SPI Flash (file system of type 1, index 0).

File name	File index	File type	File size	Description
test	0	4 (journal)	64 KB	Reserved for file system read / write tests.
housekeeping	1	4 (journal)	2 MB	Periodically gathered housekeeping data from several subsystems.
special	2	4 (journal)	2 MB	Special journal for logging any commands or responses. See Section 5.10 for a description of journal files of this type.
tm-buffer	3	4 (journal)	1 MB	Reserved for telemetry buffer. Deprecated due to latency issues with flash memory. FRAM should be preferred for buffering telemetry.

11.5 Code listings

Listing 4: FreeRTOS hook functions for having ESTCube-1 CDHS microcontroller sleep on each tick.

```
uint8_t g_request_lowpower = 0;

/**
 * An interrupt handler running at configTICK_RATE_HZ.
 */
void vApplicationTickHook( void ) {
  if (g_request_lowpower) {
    g_request_lowpower = 0;
    SCB->SCR = SCB_SCR_SLEEPONEXIT;
  // Avoid changing SCB->SCR too often .. it's slow.
  } else if ( ( SCB->SCR & (uint32_t) SCB_SCR_SLEEPONEXIT_Msk ) != 0 ) {
    SCB->SCR = 0;
  }
  g_ticks_since_sec++;
  // Synchronize hi-freq timer with ticks
  TIM2->CNT = 0;
}

void vApplicationIdleHook( void ) {
  uint8_t flags = get_startup_flags();

  // Go to sleep, in case sleeping is allowed
  if (( flags & STARTUP_SLEEP_ON_IDLE ) != 0) {
    /**
     * @note vApplicationIdleHook must not be blocking, so
     * one should not sleep, stop nor standby from here.
     */

    // Do it anyway, in case sleeping rough is allowed
    if (( flags & STARTUP_ROUGH_SLEEP_ON_IDLE ) != 0) {
      SCB->SCR = 0;
      __WFI();
    } else {
      g_request_lowpower = 1;
    }
  }
}
```

Listing 5: Example usage of exception handling.

```
uint32_t try_mount_ecffs( uint32_t i, uint32_t dev_type, uint32_t lmod ) {
  static uint32_t depth = 0;
  ecfs_file_info_t info;
  uint32_t retval = 0, try_again = 0;
  depth++;

  TRY {
    // Get the info of filesystem metadata file
    ecfs_fileinfo( g_rfs_int[ g_system_rfs_id ], FN_FFS1 + i, &info );
    // Mount it
    g_ffs_int[ i ] = ( ecffs_fs_interface_t * )
      ecfs_mount_from_device( &g_environment, FSDT_SPI_FRAM,
        g_rfs_int[ g_system_rfs_id ]->device, info.address_in_volume );
    // Getting here indicates success
    retval = 1;
  } CATCH( ECFS_EX_NO_FILESYSTEM ) {
    // Remember that we reformatted
    log_error( ec_time_isr(), DBG_EX_REFORMAT, lmod );

    if (depth < 2) {
      // Create a FLASH filesystem on FRAM
      ecffs_format( &g_environment, FSDT_SPI_FRAM,
        g_rfs_int[ g_system_rfs_id ]->device,
        info.address_in_volume, 0, 255, &g_flash_dev_desc[ i ], 0 );
      // Try to mount again later
      try_again = 1;
    }
  } FINALLY {
    if (THROWN) {   // An uncaught exception?
      log_error( ec_time_isr(), EXCEPTION, lmod );
      CONCEAL;    // Do not propagate exceptions any further.
      retval = 0;
    }
  } ETRY;

  // Shall we try again?
  if (try_again)
    retval = try_mount_ecffs( i, dev_type, lmod );

  depth = 0;
  return retval;
}
```

12 Non-exclusive license to reproduce thesis and make thesis public

I, Indrek Sünter (date of birth: 16.11.1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   Software for the ESTCube-1 command and data handling system,
   supervised by Kārlis Zālīte and Mart Noorma.

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.
   Tartu,
   May 26, 2014