UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Ervin Oro

Delta Updates for ESTCube-2 Onboard Computer Software

Bachelor's Thesis (9 ECTS)

Supervisor: Indrek Sünter, MSc
Supervisor: Helle Hein, PhD

Tartu 2018

Delta Updates for ESTCube-2 Onboard Computer Software

Abstract:

During the ESTCube-2 mission, loading of new software onto the satellite is planned, in order to introduce new features, test and compare novel software solutions, and resolve potential software or hardware issues. Main difficulties are slow uplink, limited on-board processing power, execution of software from flash memory, and frequent updates due to the experimental nature of the mission. Different methods for updating on-board software of embedded systems have been developed and used, but they are not without limitations. In this thesis a novel method for updating ESTCube-2 on-board software is designed and implemented. It compiles and uploads only new or updated functions independently of each other, and stores uploaded binaries without modifications in the first available flash area. This minimizes uplink usage (in the case of ESTCube-1, an update changed only 2% of the code on average), provides native execution speeds, and removes the need to erase flash memory for every change. This way it is possible to load new code even without rebooting the on-board computer. The results of this work can be also used to select updating method for other similar systems in addition to ESTCube-2. By reducing limits on software updates, flexibility of the space system is increased. Software flexibility has measurable value and can be more cost-effective than other options for increasing system's flexibility (hardware ones, for example).

Keywords:
ESTCube-2, delta updates, in-orbit software updates, embedded software updates

CERCS: P175 Informatics, systems theory

Delta uuendused ESTCube-2 pardaarvuti tarkvarale

Lühikokkuvõte:

ESTCube-2 missiooni jooksul plaanitakse laadida satelliidi pardale uut tarkvara, et lisada funktsionaalsust, testida ja võrrelda uudseid tarkvaralahendusi ning lahendada potentsiaalseid tarkvara või raudvara probleeme. Peamisteks keerukust lisavateks faktoriteks on aeglane üleslaadimiskiirus, tagasihoidlik arvutusvõimsus pardal, tarkvara jooksutamine välkmälust ning sage uuendamisvajadus missiooni eksperimentaalse loomuse tõttu. Varasemalt on välja töötatud ja kasutatud erinevaid lahendusi manussüsteemide pardatarkvara uuendamiseks, kuid kõigil neil esinevad omad puudujäägid. Käesoleva töö raames töötati välja ja realiseeriti uudne meetod ESTCube-2 satelliidi pardatarkvara uuendamiseks. Selle järgi kompileeritakse ja laetakse üles teineteisest sõltumatult ainult uued või uuenenud funktsioonid ning salvestatakse üles laetud binaarid muudatusteta välkmälu esimesse vabasse piirkonda. Väljatöötatud meetod vähendab üleslaadimist vajavat andmemahtu (ESTCube-1 korral muutis uuendus keskmiselt vaid 2% koodist) ning välistab vajaduse välkmälu kustutamiseks iga uuenduse käigus, võimaldades uusi funktsioone lisada ka ilma manussüsteemi taaskäivitamata. Käesoleva töö järeldusi saab kasutada lisaks ESTCube-2 satelliidile ka teistele sarnastele süsteemidele tarkvara uuendamise meetodi valimisel. Uuendamist raskendavate piirangute vähenemisest tuleneval tarkvara paindlikkuse kasvul on mõõdetav väärtus. See väärtus saab olla suurem, kui oleks alternatiivsetesse (näiteks raudvaralistesse) lahendustesse panustamisel saadav väärtus.

Võtmesõnad:
ESTCube-2, delta uuendused, tarkvara uuendamine orbiidil, manussüsteemi tarkvarauuendused

CERCS: P175 Informaatika, süsteemiteooria

Contents

1 Introduction **6**

1.1 ESTCube-2 6

1.2 Outcomes 8

2 Use cases **9**

2.1 ESTCube-1 experience 9

2.2 Use cases during ESTCube-2 mission 10

2.2.1 ESTCube-2 hardware 10

2.2.2 Types of updates 10

3 Related work **12**

3.1 Full system image replacement 12

3.2 Virtual machines and script interpreters 13

3.3 Binary differences 13

3.4 Dynamically loaded modules 14

4 Design approach **16**

4.1 Requirements 16

4.2 Proposed solution 17

4.2.1 On-board storage of application software 18

4.2.2 Linking 19

4.2.3 Compiling application functions 20

4.2.4 Global variables 21

5 Testing **22**

5.1 Implementation details 22

5.2 Function discovery and preparation 23

5.3 Running application functions 23

5.3.1 Fully self-contained function 23

5.3.2 Calling firmware functions 24

5.4 Performance 24

5.5 Section .rodata 25

6 Conclusion **27**

References **30**

Appendix A Acronyms **31**

Appendix B Indirect function calls **32**

Appendix C Boot sequence **33**

Appendix D A short library function **34**

Appendix E Licence **35**

1 Introduction

Main goal of this thesis is to design and implement a method of updating software more suitable for ESTCube-2 than existing alternatives. Proposed solution focuses on enabling delta updates (avoiding upload of unchanged code as much as possible) and reducing the amount of on-board processing required. Additionally, neither updatable software's performance nor abilities should be limited.

During the ESTCube-2 mission, loading of new software onto the satellite is planned, in order to introduce new features, test and compare novel software solutions, and resolve potential software or hardware issues. The value of increased software flexibility is measurable and can be higher than, for example, that of hardware flexibility (Nilchiani 2009). For updating software on ESTCube-2, main difficulties are slow uplink (9600 bits per second), limited on-board processing power (Ehrpais et al. 2016), execution of software from flash memory (Haljaste 2017) (limitations of which are detailed in Section 2.2.1), and frequent updates due to the experimental nature of the mission.

All existing solutions for updating software have limitations (described in Section 3) when applied to the on-board computer (OBC) of ESTCube-2. The topic is relevant right now, since ESTCube-2 is currently in development phase. Even though this thesis focuses on the use cases of ESTCube-2, the topic has wider importance - several existing solutions have been designed with Internet of Things or wireless sensor networks (Dunkels et al. 2006; Han et al. 2005) in mind. Those systems have several common aspects with nanosatellites. The method proposed in this thesis has not been previously described in literature.

Scope of this work is limited to the GNU toolchain, standard FreeRTOS distribution, and on-board software written in C.

1.1 ESTCube-2

ESTCube-2 is an experimental three-unit CubeSat. The mission is planned to serve as an in-orbit demonstration of the platform, which could then be employed on future missions. (Iakubivskyi et al. 2016)

Iakubivskyi et al. (2016) list among the systems to be tested on ESTCube-2

- tether module for plasma brake deorbiting (previous versions of which have flown on the satellites ESTCube-1 and Aalto-1),
- Earth observation camera system (which is based on the European Student Earth Orbiter camera),
- high speed C-band downlink system,
- a novel miniaturized (up to 0.6 units) satellite bus,
- cold gas propulsion module by NanoSpace,
- thin film protective coating experiment by Captain Corrosion OÜ.

ESTCube-2 is developed mostly by the students of University of Tartu, but others from all over the world have contributed as well (Ehrpais et al. 2016). Also among the main objectives for ESTCube-2 is to educate a new generation of space engineers, and to promote space technologies in general (Iakubivskyi et al. 2016).

The OBC of the satellite is tasked with running attitude and orbit control system (AOCS) algorithms, controlling payload experiments and star tracker, and handling telemetry and telecommands. The most important requirements for AOCS are set by the plasma brake and Earth observation payloads. The former needs sufficient angular momentum for the centrifugal deployment of the tether, and the latter requires accurate pointing. While some methods, like the use of magnetic torquers, build upon the heritage of the successful ESTCube-1 mission, others, like reaction wheels by Hyperion Technologies and in-house developed star tracker, are completely new for the team. Due to the large amount of experimental software, it is planned that it should be possible to perform firmware updates on all of the microcontroller units (MCUs) after the launch of the satellite. This enables the team to correct any unexpected problems that the satellite may encounter. (Ehrpais et al. 2016)

1.2 Outcomes

The method proposed in this thesis considers a function to be an atomic unit. Functions would be compiled separately and only new or updated functions would be uploaded. This way less data needs to be uploaded while preserving the native performance and abilities of the software. Primary difficulty with this approach lies in how to add new or updated functions incrementally without the need for on-board modifications. This is desirable since flash memory does not support deleting or changing any data short of an entire sector, and on-board modifications to binaries increase the risk of radiation-induced errors.

Taking into account the scope of this thesis as well, main subject of research is how the GNU C compiler generates machine code, and what are the options to alter the process to suit specific needs. Modifying the compiler itself is outside of the scope of this thesis, but in addition to compiler configuration flags, pre- and post-processing are utilized to produce separately packed functions. These packed functions would be stored sequentially into on-board flash memory as they arrive and without modifications, with the rule being that the latest appearing version of any function is considered to be in effect. This way the need to delete flash memory for each update is eliminated, making it possible to add new functions even without rebooting the embedded system. These function packages have been prepared on the ground in such a way that an on-board custom dynamic linker can connect them to each other without the need to modify received binaries. Performance, applicability and limitations of the proposed solution are analyzed.

The rest of this thesis is organized as follows. Section 2 describes how the system will be used based on ESTCube-1 experience and planned ESTCube-2 hardware. Section 3 describes identified four categories of previous approaches to the problem of embedded software updates. Their strengths and weaknesses are listed. Section 4 describes detailed requirements and the solution that is proposed in this thesis for fulfilling those requirements. Section 5 includes results of testing the solution. Appendix A lists all used abbreviations. Appendix B shows an experiment that justifies the chosen structure for the function table. Appendix C includes the bootloading procedure needed to make the proposed solution work. Appendix D demonstrates an example of a very short function from an AOCS library which is, however, still in development.

2 Use cases

Use cases for the ESTCube-2 OBC software updating system (predictions on how the system will need to be used later on) are derived from ESTCube-1 command and data handling system (CDHS) experience. Differences between the two missions are taken into account where applicable.

2.1 ESTCube-1 experience

ESTCube-1 CDHS received 20 distinct firmware updates during its mission, with 19 of them being successful (Sünter et al. 2016). Slavinskis et al. (2015) list among functionalities added to the CDHS "power saving mode, variety of data logging functions, high time-resolution functions for sensor measurements, experiment-related functions, additional preprocessing of attitude measurements, as well as attitude determination and control algorithms." Some of the changes were introduced in response to unforeseen circumstances.

Size of the changes in corresponding source codes was analyzed. On average, 2.09% (σ=3.43%) of the code lines were added, removed or edited during a firmware update. The largest update changed 14.71% of the code, while the smallest update only modified 0.02% of all the lines. Updates with largest code changes were so due to addition of large portions of new code. Overall, size of the source code for CDHS software (including all libraries) also gradually increased through the mission (from 63 390 lines of code to 78 868, a 24.42% increase). However, no update consisted of just new code being added, always some previously existing code was changed or removed as well.

As it was on ESTCube-1 (a classic nonsegmented firmware), a minor code modification could result in a binary difference of almost 99%, which caused the need to upload an entire firmware image again every time. Size of a typical CDHS firmware image after compilation was 250 kibibytes. Those images had a Shannon entropy of about 0.6 (on the scale of $0-1$), resulting in a theoretical maximum compression ratio of 40%. (Sünter et al. 2016)

2.2 Use cases during ESTCube-2 mission

The hardware, on which the software will be updated, significantly affects the choice of updating method. This choice is also dependent on the properties of the updates themselves, which can differ notably due to different reasons causing the need for those updates.

2.2.1 ESTCube-2 hardware

ESTCube-2 OBC will be centered around an STM32F767IIT6 MCU, which has 2 mebibytes[1]of internal flash and 516 kibibytes[1] of internal static random-access memory (SRAM). For the data of running programs the OBC has 2 mebibytes[1] of magnetoresistive random-access memory. For external configuration tables, error logs, on-board statistics, and other data without strict latency restrictions, it has 3×512 kibibytes[1] of ferroelectric random-access memory. Mass storage for firmware versions, measurements, and payload data is provided by 2×32 mebibytes[1] of external flash. (Haljaste 2017)

At least critical software components must be stored in the internal flash, so that it would be possible to disable all external device drivers in safe mode. This is desirable since having more code enabled increases the probability of any faults occurring. However, flash memory consists of sectors, which can be up to 256 kibibytes[1] in size (STMicroelectronics 2018). In order to edit any data already written to the memory, an entire sector must be erased and rewritten (ibid.).

2.2.2 Types of updates

Due to the limited number of suitable launches (caused mostly by limited funding possibilities), it might happen that the satellite has to be delivered on an unexpectedly accelerated schedule, so that some software functionalities are not completed or sufficiently tested before the launch. In such case, significant amounts of new code would need to be added to the firmware with an update. Additionally, some previous code would need to be modified to make use of those new functionalities. However, largest

[1]Cited sources claim those numbers to be in decimal units (kilo- and megabytes), but are assumed to do so by mistake.

parts of the firmware by size - operating system (OS), hardware abstraction layer (HAL) and drivers - must definitely be finalized before the launch in order to enable successful satellite operation. This means that the size of even the largest update caused by this reason would stay under about 20%.

Several software components are experimental. In order to assess the properties of novel solutions, they need to be compared with existing methods, which could mean the need to deploy alternative algorithms for some period of time. However, swapping out a software component could not cause changes larger than introducing a new feature.

The testing of novel software solutions also entails the need for iterative improvements, as the perfect setup is unlikely to be achieved on the first try. While most of this should be possible by only changing configuration values separate from the firmware, it might happen that some unforeseen change does require code rewriting. Updates for aforementioned reasons would affect only a small number of components and would alter significantly less than 1% of the firmware.

When bugs are discovered in on-board software, a fix needs to be developed and deployed. Such update would mostly consist of changes to the existing code, and can be expected to change considerably less than 1% of the firmware. Changes would also be limited to a single component or functionality. When a bug is discovered, it may be necessary to deploy the fix without delays, to minimize the probability of any mission critical risks materializing.

Unforeseen issues are also possible with the satellite hardware, due to the use of commercial off-the-shelf components, extreme miniaturization, and the educational nature of the development process. Some of such issues could be circumvented with software workarounds. These can range from simple code changes to the addition of completely new functionality.

Lastly, the main mission of ESTCube-2 is to conduct experiments, like unreeling the plasma brake tether. During experiments it might be useful to rapidly deploy new small subroutines that handle aspects of the experiment that were not predicted beforehand.

3 Related work

The problem of software updates on embedded systems has seen many different solutions so far. A brief overview of existing methods is given in this section.

3.1 Full system image replacement

For example, on ESTCube-1, one way to update on-board code was to recompile the entire code-base, upload it to the satellite, and after rebooting the MCU this new code would be active (Sünter et al. 2016). This method benefits from a simple design. Additionally, by replacing the entire system image, any compatibility issues are eliminated and next to none on-board processing is required. Support for full image replacement is often implemented as a fallback even on systems that support other more advanced update mechanisms as well (Garrido et al. 1998; Greco and Snyder 2005; Tarbe 2013).

Additionally, several missions have had the ability to keep several such system images stored at any time, allowing switching between them in case of problems. In some cases, like MINISAT01, one of the images was read-only and only the other one could be modified (Garrido et al. 1998). However, in some other cases, like ESTCube-1 (Tarbe 2013) and The Mars Exploration Rovers (Greco and Snyder 2005), it has been possible to update both firmware images independently. The former poses significant drawbacks: in the case of MINISAT01, a complete firmware update took 2 full days, and for that duration the original launch firmware version had to be used (Garrido et al. 1998).

However, this kind of simple design also poses significant drawbacks. Most importantly, it requires large amounts of uplink, even when the change was minor. For example, on ESTCube-1, taking into account the size of the firmware, uplink speed and orbital parameters, uninterrupted firmware update would take about 1.5 days to complete (Sünter 2014), even though on average an update changed only about 2% of the code. In the case of the MINISAT01, a full firmware update took 2 complete days, while a partial update could be done in few hours (Garrido et al. 1998). In the case of the Mars Exploration Rovers, replacement of the entire firmware image required uplinking about 8 megabytes of data, while a delta update that they completed required uplinking of approximately only 2 megabytes (Greco and Snyder 2005).

3.2 Virtual machines and script interpreters

Another method that is commonly used when new code needs to be uploaded frequently and in small parts, is to utilize virtualization techniques on board. For example, interpreted scripting language pawn script was used on ESTCube-1 (Sünter et al. 2016) and will be used on TTÜ100 satellite (Aasaväli 2017). With this approach some parts of the code are written in native code and some as scripts. Since scripts interact with rest of the system only through predefined application programming interfaces (APIs), they can be modified without influencing any other parts of the system. Thanks to the abstraction layer that the script interpreter provides, script files can be moved and rearranged without difficulties (Riemersma 2017).

As an alternative to scripting languages, virtual machines have also been used to achieve similar results. For example, Simon et al. (2006) describe a Java virtual machine for wireless sensor networks. Custom bytecodes specifically tailored for embedded systems have been explored as well (Levis and Culler 2002).

However, not all of the on-board software can be rewritten this way. Interpreted code can be significantly slower than compiled code. For example, Simon et al. (2006) measured compiled C code to be 10 times faster that equivalent java byte-code running in their interpreter (however, this benchmark can not be considered conclusive). Additionally, virtualized software has limited access to other system resources through predefined APIs.

3.3 Binary differences

In order to overcome the large uplink requirements of full image replacement while keeping the benefits of native code (most notably maximum execution speed and unlimited inter-component communication), some authors have experimented with calculating a delta between two system images and using it to recreate the new firmware image on board.

One of the simplest binary difference based approaches for delta software updates was implemented on MINISAT01. Their entire firmware image consisted of 32 parts, each of which could be updated separately. While updating the whole firmware took two full days, updating a single part could be done in just few hours. However, this approach had a serious limitation: old and updated code had to be exactly binary compatible, no lengths could be changed. Therefore this kind of approach is only useful for updating values of constants, since code updates typically also change the length of compiled binary. (Garrido et al. 1998)

To overcome this limitation, MINISAT01 also featured third method of updating on-board software. With this method, new or updated functions were located at the end of the firmware, while old versions were kept on their original locations. Then any calls to modified functions were changed in on-board software. This was possible, since modifying function call with new location does not change the length of the call, and MINISAT01 stored its firmware in electrically erasable programmable read-only memory. (ibid.)

A more complex approach to binary difference based software updates was taken on the Mars Exploration Rovers. They calculated differences between new and old firmware images that could start and end at arbitrary locations and change the size of the image as well. However, to overcome the limitations of flash memory (described in Section 2.2.1), they had to build the updated image in volatile memory and then store the new generated image in non-volatile memory. This meant that they had to reserve a full day to assemble, validate and store the updated software. During that time no other activities could be carried out. They also acknowledge that “The single biggest improvement to the Mars Explorer Rovers’ flight software modification process would be to reduce the amount of time necessary to stand down from nominal surface operations.” (Greco and Snyder 2005)

3.4 Dynamically loaded modules

Already for some time general-purpose computer systems have supported loading software components independently of each other by utilizing virtual memory (Kilburn et al. 1962). This way each component can be statically linked beforehand without conflicts. Embedded systems that have a memory management unit (MMU), have similarly used server-side pre-linking of software (Shen and Chiang 2010).

A very similar problem to that of embedded systems without an MMU has been solved on general-purpose computers for shared libraries. While on embedded systems modifications to compiled binaries are undesirable due to the limited computational power and the use of flash memory, on general-purpose computers modifications to binaries would force several versions of the same library to be kept in memory. Neither can be statically linked in order to avoid conflicts between different modules. The solution is to compile software as position independent code (PIC) by introducing a level of indirection between symbols and actual memory addresses. Several different approaches to such indirection have been explored (Levine 1999, Chapter 8). Hybrid solutions of pre-linking and pre-locating combined with dynamic re-location have also been explored (Dong et al. 2009).

The table that maps symbols to addresses can be part of the OS memory or part of each module. Example of the former can be found in the SOS operating system described by Han et al. (2005), where each module has to register functions into a jump table. The latter approach has been implemented, for example, in Contiki OS, that makes use of modified executable and linkable format (ELF) to hold code alongside its global offset table (GOT) (Dunkels et al. 2006). Unfortunately, dynamic linking of ELF files is still not widely supported on embedded operating systems (Xinyu et al. 2017). Among others like it, the standard distribution of FreeRTOS (the operating system used on ESTCube-2 OBC) does not support loading modules dynamically (Barry 2005). Xinyu et al. (2017) describe a method for including the code relocation functionality into the relocatable code itself as one possible solution.

4 Design approach

As shown, no existing solution fully covers the described use cases, and this mandates further research. This section defines requirements for a system that would be more suitable for ESTCube-2 than any existing alternatives. Possible design for meeting those requirements is also proposed.

4.1 Requirements

Software updating system that would be more suitable for ESTCube-2 OBC than any existing alternatives, must:

1. allow addition of functions without the need to re-transmit significant portions of already existing code, so that it could also be used for small one-time subroutines.
2. allow changing a subset of existing software without the need to re-transmit significant portions of unchanged code, so that upload bandwidth and update application time would be minimized.
3. not require on-board modifications of uploaded code. This makes it possible to store code directly in internal flash if external memory modules are unavailable.
4. make it possible to rearrange functions on board. This enables defragmenting the internal flash.
5. not depend on OS features that are unavailable in a standard distribution of Free-RTOS, since this is the OS planned for ESTCube-2 OBC.
6. not significantly reduce code performance compared to a system that uses full image replacement.
7. support full inter-component communication. This enables any unforeseen updates as well, which a predefined API could limit.

Not all of the identified requirements are mission critical for ESTCube-2, but fulfilling them increases the amount of available time and opportunities to carry out mission activities. Therefore they are all basis for the proposed solution.

4.2 Proposed solution

The proposed solution divides on-board software into two parts: firmware (startup script, HAL, OS, and critical device drivers) and application software (everything else) (Figure 1). Firmware would be updatable by means of full image replacement, similar to how updates worked on ESTCube-1, including the use of multiple versions. This is proposed since firmware is expected to be updated rarely, if ever, and is too critical to use experimental updating methods with it.

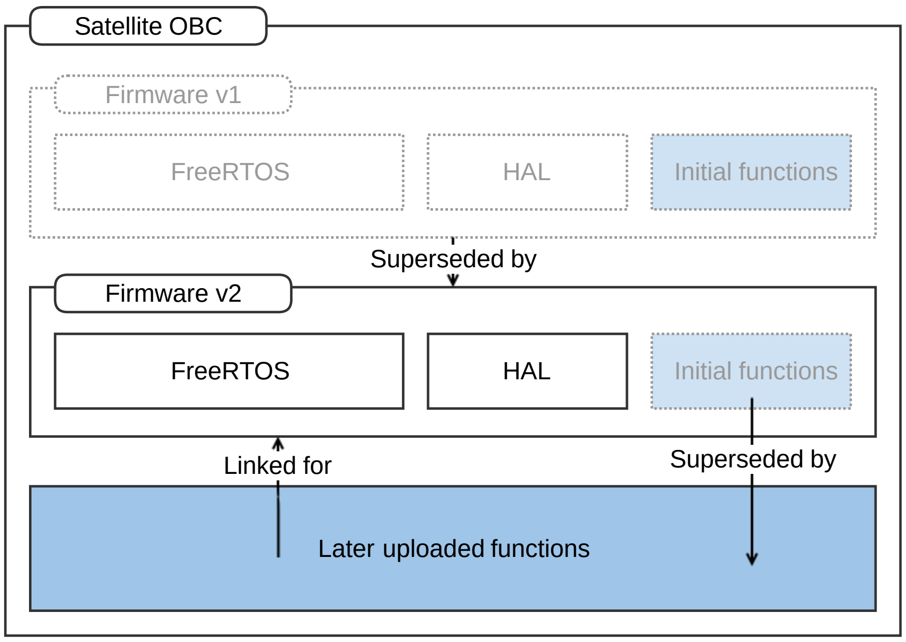

Figure 1. Proposed on-board software organization. Blue denotes application software.

Application software, on the other hand, would be divided into parts, each containing a single function. Each application function would be combined with a short header (Table 1) to form a package. Each function could then be independently updated, added or removed.

The headers (Table 1) will have nop-instructions appended to them to pad them to 12 bytes in length. Some instructions will generate a fault or will execute slower if addresses are not word-aligned (STMicroelectronics 2017, Section 3.3.5), therefore a multiple of 4 bytes was chosen for the header length.

Table 1. Application function's header.

Length (bytes)	Field	Format	
4	Cyclic redundancy check	CRC-32	
1	Type	`0xff`	nothing
		`0xaa`	function
		`0x55`	global
		`0x00`	disabled
2	Function ID	uint16	
1	Version no	uint8	
2	Length of body	uint16	
2	SRAM offset (global variables only)	uint16	

Function version dependencies would be managed by a system of snapshots similar to git (Chacon and Straub 2018, Chapter 1.3): after new functionality has been developed and tested, current version of all functions is recorded. Mission control system will then determine, which of them differ when compared to satellite's internal storage, and upload those.

4.2.1 On-board storage of application software

Main focus is on storing application functions in internal flash, since it is the most restrictive storage. Additionally, in safe mode, this is the sole non-volatile storage available (Section 2.2.1). Storage of some functions in other memories, as well as moving functions between different memories and addresses, would also be possible.

While flash memory does not support deleting or changing any data short of an entire sector, new data can be appended. Functions would therefore be written to flash sequentially as they arrive. Additionally, flash allows the addition of zeros into data. This way functions could be marked as disabled by replacing some predefined area of function header with zeros ('Type' in Table 1), without the need to delete the sector. In order to change a function, a new version would simply be appended to the storage. If several versions of a function are found in memory, the last one will be considered correct.

4.2.2 Linking

Compiled code contains many symbol references: function calls, global variables, etc. For software to run, all of them must be replaced with absolute memory addresses. In order to satisfy all requirements (especially about no modifications to uploaded code), a combination of static pre-linking and custom dynamic linker would be used.

Firmware would be compiled and linked to a fixed memory address first - meaning that absolute memory addresses of all functions and global variables in the firmware would be fixed. This way calls to all firmware functions from other firmware functions, as well as from any later added application functions, can be simply linked on the ground (Figure 2).

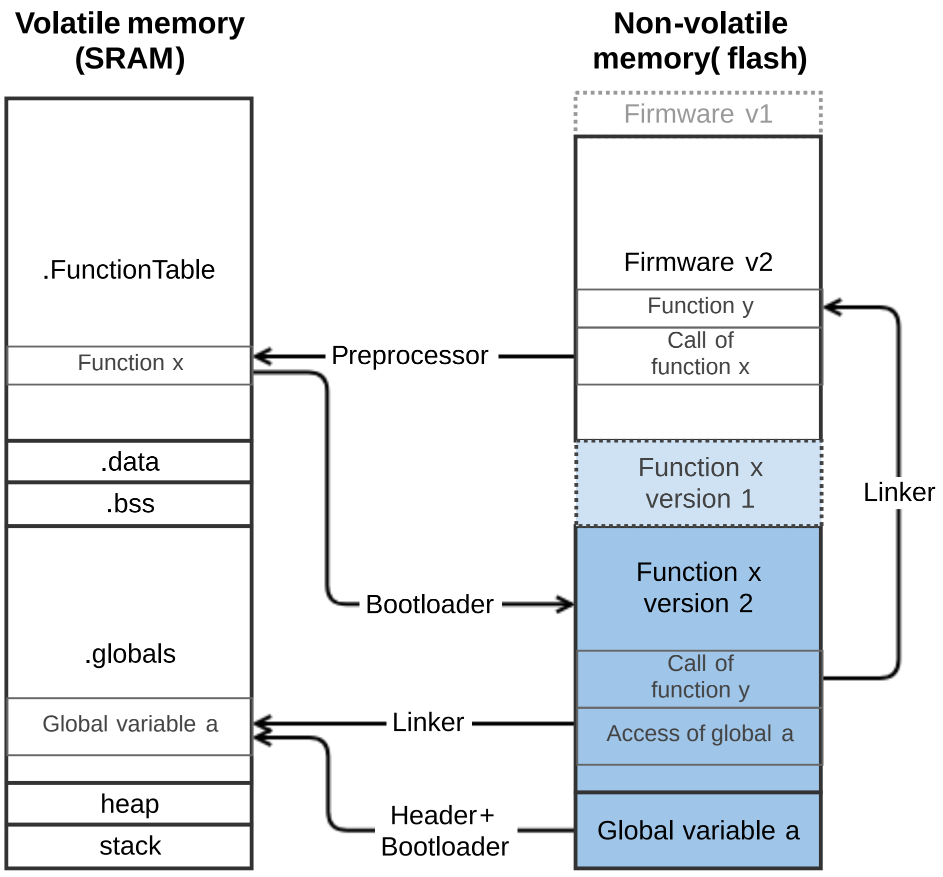

Figure 2. References in on-board software. Blue denotes application software.

Calling application functions is more tricky, as they can move around in memory. They would be callable by using a system-global offset table (Figure 2). This table would contain the current memory address for each function. It will be stored in volatile internal SRAM and re-created every time the system boots (by walking over all stored functions and storing the address where each function appears last). The table itself would be located on fixed memory location and contain functions sorted by id. It can be also changed during the run-time, when new functions are added. Changing it during the run-time when a function is changed requires additional checks to guarantee that previous memory address does not remain in use (as some function pointer somewhere) and is therefore out of the scope of this work, but is a potential future improvement.

Additionally, firmware can't call functions, whose signatures are not known at the time of firmware compilation, since compiler must generate code for passing arguments. For that reason, some functions are bundled with the firmware (Figure 1). Examples would include functions dealing with the internal communications protocol and software updating itself. They can be updated, but their signatures must remain unchanged.

The table of function offsets would be periodically checked for errors. Functions' checksums could also be checked during the generation of this table. During reboot, flash memory can also be defragmented.

4.2.3 Compiling application functions

Compilers support generating indirect function calls out of the box for PIC. However, the commonly-used Unix System V Release 4 scheme expects the GOT to be at a fixed offset from the code (Levine 1999, Chapter 8). This has the effect that each independent portion of code needs its own copy of the GOT, but having one for each function would cause unacceptable size overhead. To avoid having to modify the compiler itself, a way to rewrite all function calls using the preprocessor will be used.

For each function that would otherwise result in an unresolved external symbol, a preprocessor macro (Figure 3) will be generated. It contains instructions to read function's address from the function table, cast it to the function's type, cast all arguments to required types, and call the function. Since the macro call syntax will be identical to that of a function call, application code can be agnostic to the updating platform.

```
typedef void (*f0_t)();
#define f0() (*((f0_t)(*((uint32_t *) (0x20000000)))))()
```

Figure 3. Call interception macro for the hypothetical function `f0()` with id 0, where the function table is located at a fixed address `0x20000000`.

For calling important functions, the checksum of the function can be checked immediately before jumping by incorporating modified interception macro. Normally this is not done.

4.2.4 Global variables

In order to simplify the design, global variables would be treated very similarly to functions. First, they would be compiled, since then the the compiler would calculate their lengths. Compilation should include the flag `-fdata-sections`, this way they can be extracted to separate packages, prepended with headers, and uploaded to the satellite.

Memory space after `.data` and before heap would be reserved for updatable global variables (Figure 2). Linker would select an address for each variable in that area, and then all functions using the variable could be statically pre-linked on the ground. Headers for global variables would contain the appropriate type, and additionally the memory address allocated for this variable (Table 1). During boot, when function addresses are being added to the offset table, global variables would be copied to their respective addresses. As with a function, the latest occurrence of a global variable overrides any previous ones.

5 Testing

A build methodology was used to explore the feasibility of proposed system. After implementation, viability of the solution was tested by simulating updates to a development version of the firmware running on a prototype board.

5.1 Implementation details

A set of Python scripts aids with function compilation and packaging. Functions are discovered by searching function definitions from all source files. Global variables are extracted from symbol tables after all sources have been compiled. They are handled differently because function arguments and their types can not be obtained from symbol tables, but since all information about global variables is available there, it is preferable to let compiler do the source parsing. A side effect of implementing function discovery with regular expressions is the inclusion of function definitions from within block comments. This, however, should not cause any issues, since they will not have a corresponding binary section and therefore they will not be included in any snapshots. Data about functions and global variables, along with respective object files, gets stored in an SQLite3 database. Scripts also exist for generating interception macros, linker script, and packages with headers.

Function table can not be implemented as a branch table, since internal flash and SRAM in the MCU are too far away in the memory space. Function table will therefore contain pointers, 4 bytes per function. Storing 3 bytes per function would require 2 instructions of additional overhead per function call, so it has been decided that 4 bytes per function will be stored (Appendix B). Pointers will point to the first instruction of the function (right after the header).

Firmware contains a table generation function (Appendix C) that must run before any application software gets called. It walks over package headers through the space allocated for such packages, jumping over function bodies based on length, and stops on the first header with the type 'nothing' (`0xff`, Table 1). Function pointers are added to the function table, and global variables get copied to their respective allocated memory areas.

5.2 Function discovery and preparation

Aforementioned Python scripts were tested against simple test functions, as well as functions from the AOCS repositories. For discovering global variables, two methods were compared: parsing source code for variable definitions, and compiling source with `-fdata-sections` to discover global variable names from symbol table. Former method was not sufficiently effective due to large amount of different global variables, including matrices and other complex types. Using the latter method, most functions and all global variables were successfully packaged.

5.3 Running application functions

Since linking was known to be a difficult aspect, the proposed solution was first tested in a situation with minimal number of external symbols. Once this was observed to be functional, a more typical use case was analyzed as well.

5.3.1 Fully self-contained function

As the first test, a fully self contained function was written, meaning one that did not depend on any external variables or functions, including any standard library or operating system methods. This function blinked an indicator light by writing appropriate values directly to the MCU registers.

It was compiled independently of the rest of the firmware and stored in flash memory following the firmware image. Firmware successfully discovered the package, populated the table of function offsets, and called the function through this table. A minor inconvenience was noted: debug symbols for applications functions are not available to the debugger from the firmware's ELF.

5.3.2 Calling firmware functions

Next, a function was written that blinks the indicator lights by calling appropriate methods from HAL API. All arising issues were mitigated and the test was concluded mostly successfully. Following difficulties were experienced:

Firstly, author was unable to find a way to tell the GNU linker `ld` to use symbol addresses from a map file generated by prior linking. Even though Dunkels et al. (2006) claim to have used just the map file for pre-linking, no information on how to achieve that was found by the time of writing from documentation, different online sources nor an online forum (https://stackoverflow.com/q/48028126/7088748). This resulted in the need to link firmware separately once, and then again with each application function, hoping that the firmware layout stays unchanged. Initially that was not the case, but after explicitly specifying section order in the linker script, layout consistency was achieved.

Secondly, issues arose with the way the GNU C compiler handles local constants. HAL functions for accessing general-purpose input/output pins take a struct of port letter and pin number as an argument. Unfortunately, compiler placed the structs to `.rodata` section separately from the function code, even though they were local. More about that issue can be found in Section 5.5. For the purpose of this test, the issue was mitigated by storing all port letters and pin numbers in single-byte local variables, which the compiler then decided to store directly within the code section.

5.4 Performance

Performance overhead with the implemented system is three additional instructions per function call (Appendix B). Storage overhead per function is 12 bytes (Table 1).

An in-development AOCS library was packaged, and results were analyzed. Average size of a compiled function was 533 bytes (σ=1194 bytes) (Figure 4). Largest function was just over 8.5 kilobytes. On the other hand, the library also contained several extremely small functions. For example, one function (shown in Appendix D) was just 28 bytes after compilation, therefore the 12 byte header caused a 30% storage overhead. However, this function could be inlined, replaced with a preprocessor macro, or replaced with longer a implementation in later versions of this AOCS library.

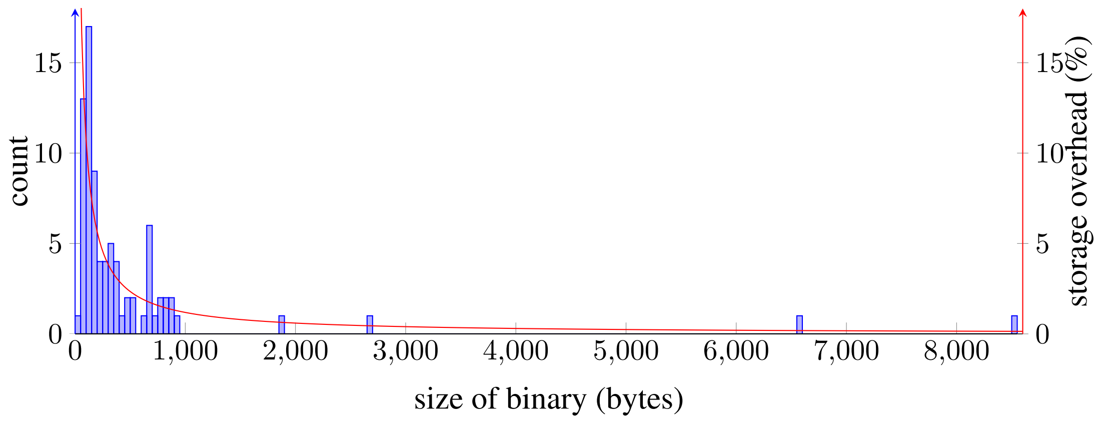

Figure 4. Compiled size and respective storage overhead for functions from an in-development AOCS library.

The overall storage overhead caused to the tested software version by function headers was 2.25%. Since very small functions could be eliminated at later stages of development, the actual storage overhead can be assumed to be even lower.

5.5 Section .rodata

ESTCube is using GNU Tools for Arm Embedded Processors (version 7.2.1 at the time of writing) to compile OBC software. When the GNU C compiler finds local constants that are longer than a byte, it separates them from the code and creates a `.rodata` section for them. The `.text` section will then contain relative reference (`//1`) to an absolute address (`//2`) inside the `.text`, which in turn points to the data (`//3`) inside `.rodata` (Figure 5). This structure remains even when the linker is told to put the `.text.function` and the corresponding `.rodata` into a single output section.

Even when using PIC, the code uses expressions relative to the program counter (PC) to access the GOT, but the GOT still contains absolute memory addresses to `.rodata`. However, in all cases, absolute memory addresses to application function components, even when in the GOT, are undesirable, because by requirements, code should not require on-board modifications and should be movable in memory space.

```
int function() { int a[] = {97, 98, 99, 100, 101, 103}; }
```

```
// Undesirable:
0000 <.text.function>:
/---/
6:  4b07    ldr     r3, [pc, #28] ; (24 <function+0x24>) //1
/---/
24: 0000    .word   0x0000 //2 absolute address of .rodata,
                           //  currently not linked
0000 <.rodata>: // section currently not yet relocated
0:  0061    .word   0x0061 //3
4:  0062    .word   0x0062
8:  0063    .word   0x0063
c:  0064    .word   0x0064
10: 0065    .word   0x0065
14: 0067    .word   0x0067
```

Figure 5. Local constants after compilation with GNU C compiler

Research on documentation, an online forum (https://stackoverflow.com/q/45371949/7088748) and the compiler support mailing list (https://gcc.gnu.org/ml/gcc-help/2018-03/msg00015.html) have not revealed any potential solutions by the time of writing.

This is not a fundamental problem with the proposed approach, since the compiler is already storing data (absolute memory addresses in this case) at the end of the `.text` section, and referencing it by PC-relative expressions. The local constants could themselves be stored in the same way. Instead, this problem is caused by the fact that nobody has happened to implement a compiler flag to turn off the generation of `.rodata` section for local constants. This is probably due to lack of use cases prior to this work.

6 Conclusion

The aim of this thesis has been to design and implement a method of updating software more suitable for ESTCube-2 than existing alternatives. Requirements for such a system were identified from a review of literature and use cases, and a design meeting those criteria was proposed. The design was validated on a prototype versions of ESTCube-2 hardware and software.

The designed system enables adding and updating single functions independently, therefore effectively minimizing the need to re-upload unchanged code. Additionally, it requires no on-board modifications to uploaded binaries. It does introduce performance overhead of three instructions per application function call, and estimated 2.25% storage overhead. Some limitations were also discovered with the proposed design, most importantly the inability to function in the presence of large local variables.

Therefore, the overall result is that limitations present in other existing solutions and outlined in requirements were successfully mitigated. Proposed and implemented solution requires less uplink bandwidth than full image replacement, provides unlimited inter-component communication unlike virtualization solutions, and does not require any on-board modifications unlike binary differences or dynamic module loading. On the other hand, some existing solutions, like full image replacement for example, may be applicable in more different situations despite their shortcomings. Future work in this area should focus on overcoming the shortcomings outlined in this thesis by exploring alternative compilers or exploring possibilities of adding additional code generation options to GNU C compiler.

The review of existing solution and the in-depth analysis of one novel solution, presented in this thesis, can be used by the ESTCube project and by other embedded systems to make better decisions about what methods to use for updating software. This can help those systems to better maintain their functionality in an uncertain environment, and to even take advantage of uncertainty by adding novel applications or improving functionality during their missions.

References

Aasaväli, S. (2017). *Scripting engine for execution of experimental scripts on TTÜ nanosatellite* (Bachelor's Thesis, Tallinn University of Technology). (Cit. on p. 13).

Barry, R. (2005). Re: Dynamic linking/loading on at91sam7s. Retrieved January 2, 2018, from https://www.freertos.org/FreeRTOS_Support_Forum_Archive/September_2005/freertos_dynamic_linking_loading_on_at91sam7s_1346460.html. (Cit. on p. 15)

Chacon, S., & Straub, B. (2018). *Pro git* (2nd ed.). Apress. ISBN: 978-1484200773. (Cit. on p. 18).

Dong, W., Chen, C., Liu, X., Bu, J., & Liu, Y. (2009). Dynamic linking and loading in networked embedded systems. In *2009 IEEE 6th international conference on mobile adhoc and sensor systems* (pp. 554–562). IEEE. doi:10.1109/MOBHOC.2009.5336957. (Cit. on p. 15)

Dunkels, A., Finne, N., Eriksson, J., & Voigt, T. (2006). Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on embedded networked sensor systems* (pp. 15–28). ACM. ISBN: 1-59593-343-3. doi:10.1145/1182807.1182810. (Cit. on pp. 6, 15, 24)

Ehrpais, H., Sünter, I., Ilbis, E., Dalbins, J., ... Slavinskis, A. (2016). Estcube-2 mission and satellite design. In *Small satellites, system & services symposium*. (Cit. on pp. 6, 7).

Garrido, B., Garcia, A., Alfaro, N., & Asensio, J. D. M. (1998). MINISAT01 on-board software maintenance. In *DASIA 98 - data systems in aerospace* (Vol. 422, p. 65). ESA Special Publication. (Cit. on pp. 12, 14).

Greco, M. E., & Snyder, J. F. (2005). Operational modification of the Mars exploration rovers' flight software. In *2005 IEEE international conference on systems, man and cybernetics* (Vol. 1, pp. 8–13). IEEE. doi:10.1109/ICSMC.2005.1571114. (Cit. on pp. 12, 14)

Haljaste, H. (2017). *Electronics design and testing for ESTCube-2 on-board computer system with sensors for attitude determination* (Master's thesis, University of Tartu). (Cit. on pp. 6, 10).

Han, C.-C., Kumar, R., Shea, R., Kohler, E., & Srivastava, M. (2005). A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on* *mobile systems, applications, and services* (pp. 163–176). ACM. ISBN: 1-931971-31-5. doi:10.1145/1067170.1067188. (Cit. on pp. 6, 15)

Iakubivskyi, I., Ehrpais, H., Dalbins, J., Oro, E., … Merisalu, M. (2016). ESTCube-2 mission analysis: Plasma brake experiment for deorbiting. In *67th international astronautical congress (IAC 2016): Making space accessible and affordable to all countries* (IAC-16,E2,4,4,x33190), International Astronautical Federation. (Cit. on pp. 6, 7).

Kilburn, T., Edwards, D. B. G., Lanigan, M. J., & Sumner, F. H. (1962). One-level storage system. *IRE Transactions on Electronic Computers*, *EC-11*(2), 223–235. doi:10.1109/TEC.1962.5219356. (Cit. on p. 14)

Levine, J. R. (1999). *Linkers and loaders*. The Morgan Kaufmann Series in Software Engineering and Programming. Morgan Kaufmann. ISBN: 1-55860-496-0. (Cit. on pp. 15, 20).

Levis, P., & Culler, D. (2002). MatÉ: A tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on architectural support for programming languages and operating systems* (pp. 85–95). ASPLOS X. ISBN: 1-58113-574-2. doi:10.1145/605397.605407. (Cit. on p. 13)

Nilchiani, R. (2009). Valuing software-based options for space systems flexibility. *Acta Astronautica*, *65*(3), 429–441. doi:10.1016/j.actaastro.2009.02.012. (Cit. on p. 6)

Riemersma, T. (2017). The pawn language. Retrieved February 23, 2018, from https://www.compuphase.com/pawn/pawn.htm. (Cit. on p. 13)

Shen, B.-Y., & Chiang, M.-L. (2010). A server-side pre-linking mechanism for updating embedded operating system dynamically. *Journal of Information Science & Engineering*, *26*(1), 39. (Cit. on p. 14).

Simon, D., Cifuentes, C., Cleal, D., Daniels, J., & White, D. (2006). Java™ on the bare metal of wireless sensor devices: The Squawk Java virtual machine. In *Proceedings of the 2Nd international conference on virtual execution environments* (pp. 78–88). VEE ’06. ISBN: 1-59593-332-8. doi:10.1145/1134760.1134773. (Cit. on p. 13)

Slavinskis, A., Pajusalu, M., Kuuste, H., Ilbis, E., … Noorma, M. (2015). ESTCube-1 in-orbit experience and lessons learned. *IEEE Aerospace and Electronic Systems Magazine*, *30*(8), 12–22. doi:10.1109/MAES.2015.150034. (Cit. on p. 9)

STMicroelectronics. (2017). *PM0253 Programming manual: STM32F7 Series and STM32H7 Series Cortex®-M7 processor programming manual*. Rev 4. (Cit. on p. 17).

STMicroelectronics. (2018). *RM0410 Reference manual: STM32F76xxx and STM32F77xxx advanced Arm$^{®}$-based 32-bit MCUs*. Rev 4. (Cit. on p. 10).

Sünter, I. (2014). *Software for the ESTCube-1 command and data handling system* (Master's thesis, University of Tartu). (Cit. on p. 12).

Sünter, I., Slavinskis, A., Kvell, U., Vahter, A., ... Ilves, T. (2016). Firmware updating systems for nanosatellites. *IEEE Aerospace and Electronic Systems Magazine*, *31*(5), 36–44. doi:10.1109/MAES.2016.150162. (Cit. on pp. 9, 12, 13)

Tarbe, K. (2013). *Bootloader for ESTCube-1 command and data handling system and camera module* (Bachelor's Thesis, University of Tartu). (Cit. on p. 12).

Xinyu, T., Changyou, Z., Chen, L., Aourra, K., & YuanZhang, L. (2017). A code self-relocation method for embedded system. In *2017 IEEE international conference on computational science and engineering (CSE) and IEEE international conference on embedded and ubiquitous computing (EUC)* (Vol. 1, pp. 688–691). doi:10.1109/CSE-EUC.2017.131. (Cit. on p. 15)

Appendices

Appendix A Acronyms

AOCS attitude and orbit control system. 7, 8, 23–25, 34

API application programming interface. 13, 16, 24

CDHS command and data handling system. 9

ELF executable and linkable format. 15, 23

GOT global offset table. 15, 20, 25

HAL hardware abstraction layer. 11, 17, 24

MCU microcontroller unit. 7, 10, 12, 22, 23

MMU memory management unit. 14, 15

OBC on-board computer. 6, 7, 9, 10, 15, 16, 25

OS operating system. 11, 15–17

PC program counter. 25, 26

PIC position independent code. 15, 20, 25

SRAM static random-access memory. 10, 18, 20, 22

Appendix B Indirect function calls

```
// Experiment to quantify the exact memory/speed tradeoff when using
   24 vs 32 bits per function pointer in function table
// Decision: 32 bits makes more sense in this case
.syntax unified
.cpu cortex-m7
.thumb

function: // Function to be called
bx lr // Immediately return

jump32: // How would function call overhead look like in case of 32
   bits of RAM per function pointer
ldr r0, [r1]
bx r0

jump24: // How would function call overhead look like in case of 24
   bits of RAM per function pointer
ldr r0, [r1]
lsr r0, r0, #8
orr r0, 0x08000000
bx r0

.type  Reset_Handler, %function
Reset_Handler:
ldr sp, =_estack
ldr r1, =function // Load function pointer into register 1
bl jump32
sub r1, #1 // Subtract one byte from the label address because 24
   bits have to be right-aligned
bl jump24
loop:
b loop

.section  .isr_vector,"a" // Interrupt Service Routines vector table,
    "a"llocatable
g_pfnVectors:
.word _estack // End of stack; defined in linkerscript
.word Reset_Handler
```

Appendix C Boot sequence

```
volatile uint32_t __attribute__((section(".FunctionTable")))
    FunctionTable[255]; // Table of function pointers

typedef enum {
  PACKAGE_DISABLED = 0x00,
  PACKAGE_GLOBAL = 0x55,
  PACKAGE_FUNCTION = 0xaa,
  PACKAGE_NOTHING = 0xff
} package_type_t;

typedef struct __attribute__((__packed__)) {
  uint32_t       crc;
  package_type_t type;
  uint16_t       id;
  uint8_t        version;
  uint16_t       len;
  uint16_t       vma_offset;
} package_header_t;

extern uint32_t _ram; // linker symbol

void gentable(package_header_t *header, uint8_t *end) {
  while ((uint8_t*)header < end) { // safeguard
    if (header->type == PACKAGE_FUNCTION) {
      FunctionTable[header->id] = (uint32_t)header + sizeof(
          package_header_t) + 1; // ARM Thumb jumps need odd addresses
      header += sizeof(package_header_t) + header->len;
    } else if (header->type == PACKAGE_GLOBAL) {
      memcpy(&_ram + header->vma_offset, header + sizeof(
          package_header_t), header->len);
      header += sizeof(package_header_t) + header->len;
    } else if (header->type == PACKAGE_NOTHING) {
      break; // reached the end of last package
    } else if (header->type == PACKAGE_DISABLED) {
      header += sizeof(package_header_t) + header->len;
    }
  }
}
```

Appendix D A short library function

Disassembly of a very short (28 bytes total) function from the in-development ESTCube-2 AOCS Light Unscented Particle Filter library that was analyzed.

```
00000000 <ec_cos>:

float ec_cos( float radIn ) {

   0: b500        push {lr}
   2: b083        sub  sp, #12
   4: 9001        str  r0, [sp, #4]
   6: 9801        ldr  r0, [sp, #4]
   8: 4b03        ldr  r3, [pc, #12]  ; (18 <ec_cos+0x18>)
   a: 4798        blx  r3
   c: 4603        mov  r3, r0
   e: 4618        mov  r0, r3
  10: b003        add  sp, #12
  12: f85d fb04   ldr.w  pc, [sp], #4

  if (!MATH_USE_LUT || !MATH_HAS_TABLE())
    return cosf( radIn );
  // The function has an else part, which did not make it past the
     compilation step due to the shown preprocessor constants

  16: bf00        nop
  18: 00000000   .word 0x00000000

  // relocation 18: R_ARM_ABS32 cosf
}
```

Appendix E Licence