TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Sander Aasaväli 142757IAPB

SCRIPTING ENGINE FOR EXECUTION OF EXPERIMENTAL SCRIPTS ON TTÜ NANOSATELLITE

Bachelor's thesis

Supervisor: Martin Rebane
MSc
Lecturer
Co-Supervisor: Peeter Org

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Sander Aasaväli 142757IAPB

INTERPRETAATOR TTÜ NANOSATELLIIDIL EKSPERIMENTAALSETE SKRIPTIDE KÄIVITAMISEKS

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc
Lektor
Kaasjuhendaja: Peeter Org

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Sander Aasaväli

22.05.2017

Abstract

Main subject for this thesis is choosing a scripting engine for TTÜ (Tallinna Tehnikaülikool) nanosatellite. The scripting engine must provide functionality, like logging, system debugging, determination, and perform certain tasks, like communicating with the bus, file writing and reading. The engine's language must be powerful enough to fill our needs, yet small and simple enough to have as small flash and RAM (Random Access Memory) footprint as possible.

The scripting engine should also be implemented on an external board (STM32f3discovery). This way the engine's flash footprint, RAM footprint and performance can be tested in our conditions.

The outcome was that, both Pawn and My-Basic were implemented on the external board. The flash and RAM footprint tests along with performance tests were executed and results were analysed.

This thesis is written in English and is 38 pages long, including 5 chapters, 6 figures and 2 tables.

Annotatsioon

Interpretaator TTÜ nanosatelliidil eksperimentaalsete skriptide käivitamiseks

Töö põhieesmärk oli valida ja implementeerida TTÜ nanosatelliidile interpreetator, et oleks võimalik läbi viia eksperimente, siluda süsteemi või logida andmeid, kui satelliit on juba õhus.

Esimene ja teine peatükk kirjeldas missiooni tausta. Seal on välja toodud, miks on üldse interpretaatorit vaja ja milleks peab see võimeline olema.

Kolmas peatükk keskendus erinevate interpretaatorite analüüsimisele internetist leitud info põhjal. Iga interpretaatori tunnused ja võimalused olid välja toodud. Samuti oli ära märgitud mälu kasutus, kui seda infot oli saadaval. Kolmanda peatüki lõpus valiti välja kaks interpretaatorit üheteistkümnest, mida testiti lähemalt.

Neljandas peatükis analüüsisiti implementeeritud interpretetaatoreid. Samuti kirjeldati probleeme, mis tekkisid keskkonna üles seadmisel ning interpretaatori implementeerimisel, ja pakuti ka neile lahendus. Interpretaatorid kõrvutati mälukasutuse ja jõudluse poolest. Tänu testimisele sai välja valitud lõplik interpretaator, mis antud projekti sobib suurepäraselt. Valituks osutus Pawn, mis võttis mälu umbes 6 korda vähem ja lõpetas toimingud umbes 7 korda kiiremini, kui My-Basic.

Esialgse implementatsiooni võib leida TTÜ satelliidi gitlab hoidlast. Lisaks on esitatud ka näitekood, et interpretaatori uuendamine ja uute skriptide kirjutamine oleks kiirem ja ladusam.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 38 leheküljel, 5 peatükki, 6 joonist, 2 tabelit.

List of abbreviations and terms

API	Application Programming Interface
ARM	Acorn RISC Machines
CISC	Complex Instruction Set Computing
CPU	Central Processing Unit
DPI	Dots per inch
EEPROM	Electrically Erasable Programmable Read-Only Memory
GB	Gigabyte
HDLC	High-Level Data Link Control
IBM	International Business Machines
IDE	Integrated Development Environment
KB	Kilobyte
MB	Megabyte
MCU	Memory Controller Unit
RAM	Random Access Memory
RISC	Reduced Instruction set Computing
ROM	Read Only Memory
RTOS	Real Time Operating System
TTÜ	Tallinna Tehnikaülikool
UAV	Unmanned Aerial Vehicle
UC	University of California
USB	Universal Serial Bus

Table of contents

1 Introduction 11
1.1 About TTÜ nanosatellite project 11
1.2 Scripting engine 11
1.2.1 Tasks 12
1.2.2 Functionality 13
1.3 Goals of this work 13
2 Background and requirements 14
2.1 CubeSat standard 14
2.2 Subsystems 14
2.3 Processors 14
2.3.1 CISC 15
2.3.2 RISC 15
2.3.3 RISC vs CISC 17
2.4 Our satellite's hardware architecture 18
2.5 Expectations for scripting engine 20
3 Evaluation of engines 21
3.1 Pawn 21
3.2 ELua 21
3.3 Squirrel 22
3.4 Amforth 22
3.5 PyMite 23
3.6 Armpit Scheme 23
3.6.1 Scheme 23
3.7 PicoC 24
3.8 Angelscript 24
3.9 TinyTCL 25
3.10 Wren 25
3.11 My-Basic 26
3.12 Conclusive table 26

3.13 Analysis 26
4 Implementation.. 29
4.1 Environment 29
4.1.1 Issues fixed during setup 29
4.2 Performance tests on MCU.. 29
4.2.1 Technical issues.. 30
4.2.2 Flash footprint 31
4.2.3 RAM footprint.. 31
4.2.4 Performance test 32
4.2.5 Conclusive table 34
4.3 Our choice.. 34
5 Summary.. 35
References 36

List of figures

Figure 1. Nanosatellite floating in space [1]... 11
Figure 2. RISC vs. CISC conclusive table [13]... 18
Figure 3. TMS320C6727 board [18]... 19
Figure 4. Associative array in Lua language [24]... 22
Figure 5. STM32f3 board [44]... 30
Figure 6. Testing script for Pawn... 33

List of tables

Table 1. Theoretical info about the scripting engines 27
Table 2. Info about tested scripting engines 34

1 Introduction

1.1 About TTÜ nanosatellite project

The project started in March 2014. Goal of TTÜ nanosatellite project is to provide students from TTÜ with theoretical and practical experience to work with highly rated technology such as a satellite [1]. This experience provides high quality work force for both Estonian and International tech companies [1].

Scientific/technological goal of this project is to test line-scan camera on a nanosatellite [1]. Line-scan camera has not yet been installed on a nanosatellite.

The satellite is being built and tested in 2016 and 2017 and it is being launched at the end of 2017 or in 2018 [1].

Figure 1. Nanosatellite floating in space [1]

1.2 Scripting engine

A scripting engine can be thought of as a vehicle for implementing scripts [2] or an interpreter that converts script into machine code immediately [3]. “Like Java Virtual

Machine where Java programs run, scripting engine provides the necessary steps for a script to run." [4]

Communication speed between ground station and satellite is approximately 1 kilobyte per second, which is very low speed for transmitting bigger files. Also, as we can only establish a connection 4 times a day for 10 minutes (12 minutes, if the satellite flies directly over ground station), we cannot transmit files as it comes to our heads. Doing a firmware update when some minor changes are needed is too time consuming.

The scripting engine solves this problem by allowing us to send a small script file, which for example chooses only a small fraction of research or log data and sends it down to ground station, saving us a lot of upload/download time.

1.2.1 Tasks

Our scripting engine has three main tasks:

Read and write files – Read/write some research data, update software. In case we need some previously logged data (or research information) on earth, we need to read some previously written files and filter out the data we need. From there, the scripting engine can either send it directly to ground station or write the data to another file (which is smaller) and send it down to earth later.

Reading from and sending messages to bus – communicating with other modules cannot be done directly. Bus is our satellite's physical connection, similar to ethernet connection. To communicate with another module, a message must be written to the bus with HDLC (High-Level Data Link Control) protocol message format. The package contains an address, a control sum, message type and the message itself. Every module is constantly listening to the bus. If a package addressed to the given module is detected, it is received and checked. When the control sum is false, the package is returned.

The scripting engine can write on or read from the bus through RTOS (Real Time Operating System). Only messages via RTOS can be read by the scripting engine. Because a lot of messages can be sent to RTOS that do not concern our running script, message types and content need to be constantly checked. RTOS does not offer thread safe messaging, but it can be developed on it.

Logging – In case we need to log some information (for example: from some sensors) for a short period of time, we do not want to update our satellite's firmware. Sending a script that will command to do so, will be much quicker and effective.

1.2.2 Functionality

A scripting engine must also provide four functionalities:

Calculations – Mostly when some parameters need to be compared. For example, we need the sun sensor's readings on an exact time. These comparisons need some calculation power.

Determination – This comes in forms of "if" sentences. Since communication between ground station and satellite is slow, we cannot send the data back and forth, but the scripting engine needs to be able to execute conditional statements based on information on other subsystems.

System debug – on some malfunction, instructions on what to test, will come from scripting engine so that a report can be sent to earth. Also, analysing log files to find out where some problem first occurred. Depending on that data, maybe some instant fix can be made or else, that data can be analysed on Earth for final solution.

Logging - Logging can only be possible when files can be written and edited.

1.3 Goals of this work

Primary goal of this thesis is to choose and implement a proper scripting engine for TTÜ satellite, so that we can perform certain actions like taking a picture, debugging system, logging data or updating on the fly. Also, scripting engine is responsible for some calculations that is necessary for satellite's life-cycle. As a result, a functional engine with example scripts will be running on test board.

2 Background and requirements

2.1 CubeSat standard

TTÜ satellite is built according to CubeSat standard. CubeSat is a cube-shaped satellite, with 10 cm sides and with weight below 1.33 kg. This standard was developed in 1999 by California Polytechnic University and Stanford University with purpose to provide access to space for small objects [5]. Those objects (or satellite in our case) are launched as secondary payloads on launch vehicles or put in orbit by deployers on the International Space Station. TTÜ satellite will be launched as a secondary payload on a larger satellite.

2.2 Subsystems

The TTÜ satellite will have these following subsystems on board:

Communications system – subsystem, that organizes communication between the satellite and ground station [6].

Power system – subsystem, that is responsible for generating power, distributes it between subsystems, charging the battery and handling emergency communications [6].

On-board computer – primary control system of the satellite [6], that hosts our scripting engine.

Attitude determination and control system – subsystem for "determining and controlling satellite's attitude (orientation) in space" [6].

Optical payload – responsible for handling the camera and taking photographs.

2.3 Processors

This chapter introduces the two main processor architecture types. Knowing, which processor architecture we have, is important for the development of scripting engine, because it may limit datatypes and their lengths. For example the program code for

floating point calculations is the same for both architectures in higher-level language (e.g. C), but generated machine code differs in memory footprint and speed.

2.3.1 CISC

CISC stands for "Complex Instruction Set Computing" and is a type of microprocessor design [7]. The CISC architecture is different from RISC (Reduced Instruction Set Computer), because a single instruction can execute different low-level instructions or do multi-step operations [8]. The logic behind was to have least number of instructions per task, so it would be the most efficient [9]. Later it was discovered that instead by using only the small, short [7] and most frequent instructions, the task where completed faster [9]. This discovery led to the invention of another microprocessor design, called “RISC” [7].

In the early days, most of programming were done in assembly languages or machine code [9]. That is why CPU (Central Processing Unit) designers tried to make powerful and easy to use instructions, that do as much work as possible [9].

Most common characteristics of a CISC processor:

- **Many special purpose registers**: Special registers are for interrupt handling, stack pointer etc. This makes the instruction set more complex, but simplifies the hardware design [9].

- **Few general-purpose registers**: Instruction, that operate directly on memory. These registers have limited space on the chip [9].

- **Complex instruction-decoding logic**: It is needed so that a single instruction can support numerous addressing modes [9].

- **“Condition code” register**: The register compares the last result to zero and records if some errors occur [9].

2.3.2 RISC

RISC stands for “Reduced Instruction Set Computer”, which is a type of microprocessor architecture that is designed to perform a smaller number of types of computer instructions [10], which increases its operating speed [11]. Because less instruction types

that must be performed require less transistors and circuity, which makes the microprocessor less complicated and faster in operation [11].

Some systems from the 1960s and 1970s have been identified as precursors for RISC [6], but the first modern RISC projects came from IBM (International Business Machines), Stanford, and UC(University of California)-Berkeley in the late 70s and early 80s [12]. Most of RISC processors share these characteristics:

- ***Instruction set philosophy***: Because RISC processors can usually do one instruction per cycle due to good optimization [12], the amount of work any instruction requires is reduced [10].
- **Instruction format**: Simple encoding and fixed-length instructions greatly simplify fetch, decode and issue logic [10].
- **Few data types in hardware**
- **Identical general purpose registers**: This allows any register to be used in any context needed [10], which means that less different registers are needed or more can be used for more complex instructions.

Some advantages that RISC provides:

- Programming on RISC processors is easier with a smaller set of instructions, thus making the progress of programming faster [11].
- Because RISC in terms of different hardware (registers etc.) is a lot simpler (than CISC), it leaves more space on the microprocessor. More space gives more opportunities to place items on the microprocessor [11].
- If newly developed microprocessors aim to be less complicated, they can be developed and tested faster [11].
- Compilers for high-level programming languages compile more efficient code, because back when compilers where developed, a smaller instruction set could be used [11].

- As most of the instructions can be done in one clock cycle, pipelining is possible [13].

2.3.3 RISC vs CISC

Searching a number from memory on CISC design board would be: FIND (start address, end address, value). This means, that there's only one instruction. With a RISC design board: a cycle has to be created to read values from memory and compare them to the searched value. That takes about ten instructions.

The simpler way to show the differences of a RISC processor and a CISC processor, is to show multiplication by both architectures.

The main goal for CISC is to complete the given task with as few steps as possible [8]. Multiplying two numbers the CISC way would be using a specific prepared instruction (like "MULT"). As the instruction is executed, it loads the two values to two separate registries, multiplies the operands in the execution unit and then stores the answer to one of the two registries [14]. This example would only take one line of code.

The multiplication can be achieved with one instruction:

- MUL 1:3 4:2 [13].

As RISC design dictates, it uses simpler instructions that can usually be done by one clock cycle [13]. So for the same multiplication example, the whole task is divided into three smaller instructions: Load, which moves data from memory bank to a register, Prod, which finds the product of two operands located within the registers, and Store, which moves the data back to the memory bank from the register [13].

The multiplication on RISC is thus achieved with four instructions:

- LOAD A, 1:3
- LOAD B, 4:2
- PROD A, B
- STORE 1:3, A

Modern RISC processors can be much more complex than CISC processors in terms of electronic circuit complexity, number of instructions or the complexity of their encoding patterns [8]. The main characteristic, that separates them, is that RISC designs use uniform instruction length for most of their instructions [8].

To conclude this chapter, this table is presented: Summary is presented on figure 2

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in Instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code Sizes
Transistors used for storing complex Instructions	Spends more transistors on memory registers

Figure 2. RISC vs. CISC conclusive table [13]

All the testing is on external board, with RISC architecture, to minimize future work on porting all the code from computer to board. It must be done anyway. The sooner it is done, the less problems have to be faced at once. Some problems that can occur: absence of libraries (like stdio.h,), change of instruction set, memory shortage.

2.4 Our satellite's hardware architecture

As previously mentioned, our satellite has 5 subsystems total: attitude determination and control system, communication, on-board computer, optics and EPS.

For memory management, we have:

- NAND[1] flash (2 GB (Gigabyte)) – for file system.

[1] Most suitable technology for writing and erasing data [15]

- NOR[1] flash (32 MB (Megabyte)) – for our operating system.
- SD[2] RAM (256 MB) – Operating memory.

Our satellite's CPU is TMS320C6727 (seen on figure 3), a floating-point digital signal processor. It is the next generation of Texas Instruments' C67x family of high-performance processors [17]. TMS320C6727 has RISC type of architecture. Next are presented some of the features of this processor:

- On-chip memory: 32KB Program Cache, 256 KB RAM, 384 KB (Kilobyte) ROM (Read Only Memory) [17].
- Scalable frequency up to 300 MHz [17].

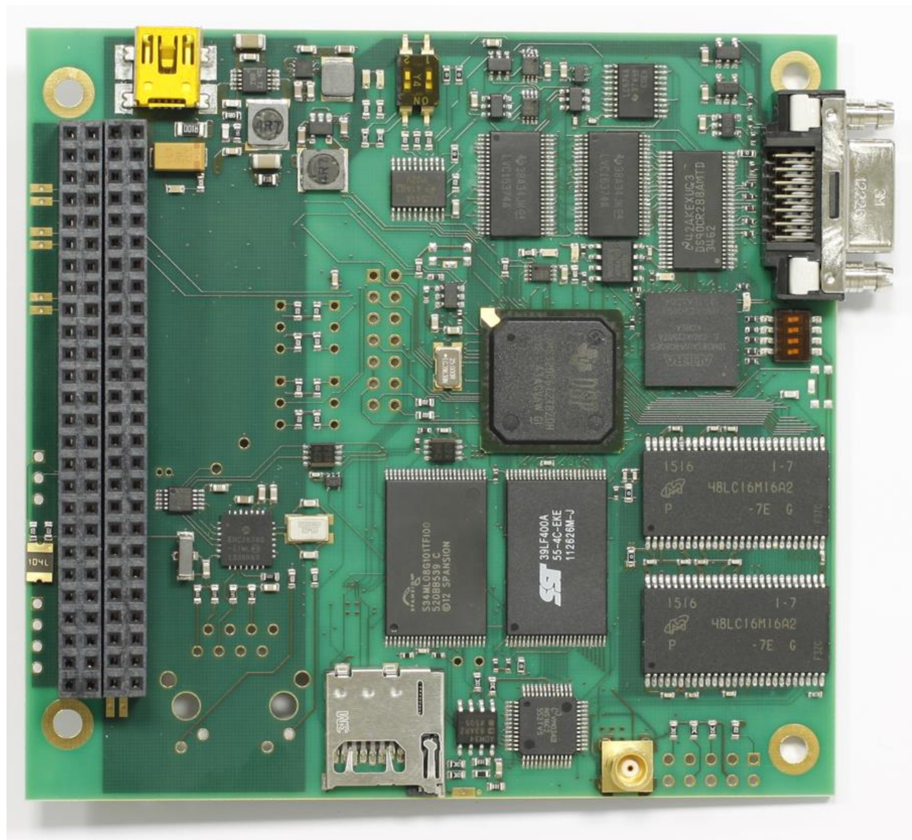

Figure 3. TMS320C6727 board [18]

[1] Most suitable technology for reading and random accessing data [15]

[2] A lot faster than normal RAM [16]

2.5 Expectations for scripting engine

As we are using bus for communication between modules, it sets us some limitations. When a script sends out a message on to the bus, it does not know when or if a message will be returned. It must loop for some time to wait and listen to the message and maybe time out, if too much time will pass.

The power usage of the scripting engine should be rather minimal. The performance of the scripting engine is important at that part. The more ticks it takes to run a script through, the more power the engine uses.

Since a lot of log files will be created during experiments, other files should take minimal space. That also means that our scripting engine flash footprint should be as small as possible, but without cutting any functionality.

When an emergency occurs, a rudimentary boot could be necessary, which leaves us less space. For example, if some part of our RAM breaks down, we must write a script to test it through, so that we can write a new firmware that would avoid the broken parts.

3 Evaluation of engines

This chapter presents 11 scripting engines/interpreters, which could possibly be used on our satellite.

3.1 Pawn

Pawn uses a C like syntax and has only one data type – the cell. A cell is most used as an integer, but it can be treated also as a character, a boolean, or a floating-point value. Pawn does not support classes or structures, but the latter can be simulated with named array positions [19].

Pawn separates its compiler from its virtual machine. The compile time check is also very static. Every variable must be declared and all native functions must have forward declarations. This makes finding errors much quicker, because you can find them on compiling, thus preventing many errors that may or may not appear on run-time [19].

Pawn has no garbage collection, which means that all allocated memory should usually be removed by code. Pawn is single-threaded; thus, it is not thread safe [20].

Pawn is not functional, it is procedural, and does not support lambda functions or late binding, which makes it a low-level language [20].

Pawn's code must be run through a compiler, which produces a binary. Produced binary will work on any platform that the host application uses. Thanks to that, loading time is reduced [20].

3.2 ELua

ELua stands for Embedded Lua. Since Lua is a minimal, but very functional language, which makes it a very viable candidate for the embedded world. ELua syntax is a mixture of both Lua and C [21]. Changing hardware in the future should be no problem, since eLua supports many boards, hence minimal to none modifications are needed to make the code work [22].

Lua is more dynamic than Pawn and has interesting features including easier and better control over strings and arrays. It means that no exact size memory has to be allocated [23]. Lua also has garbage collection. Since Lua is more popular language, it has better documentation and has more problems solved by users [23] [19]. Lua offers tables as a complex data type [24]. Tables are associative arrays and since functions can be stored to variables, tables are able to imitate classes or objects [19].

Sample of an associative array in eLua is presented on figure 4.

```
a = {}     -- create a table and store its reference in `a'
k = "x"
a[k] = 10        -- new entry, with key="x" and value=10
a[20] = "great"  -- new entry, with key=20 and value="great"
print(a["x"])    --> 10
k = 20
print(a[k])      --> "great"
a["x"] = a["x"] + 1     -- increments entry "x"
print(a["x"])    --> 11
```

Figure 4. Associative array in Lua language [24]

3.3 Squirrel

Squirrel is a dynamically typed language with C like syntax. It is a high-level object oriented language with support for classes and inheritance. Another great feature of Squirrel is that it has higher order functions [25], meaning that functions can be passed as function parameters and/or functions can be returned as its results[26]. This allows more complex procedures and gives more opportunities for programmers.

Squirrel has already good documentation [19]. Some more features that Squirrel offers: tail recursion, exception handling, cooperative threads, automatic memory management [27].

3.4 Amforth

AmForth is an extendible command interpreter for embedded use [28]. AmForth uses an old language called Forth, which is imperative stack-based [29]. It does not need any additional hardware besides the controller. It’s core system needs only 8 to 12 KB of flash memory, 80 bytes EEPROM (Electrically Erasable Programmable Read-Only Memory) and 200 bytes RAM [28].

Sadly it cannot be embedded into other programs [28].

3.5 PyMite

PyMite is a flyweight Python interpreter to execute on 8-bit and larger microcontrollers [30] with low resources (55 KB of flash and 8 KB (recommended) of RAM) [31]. It supports some of the Python 2.5 syntax and can even execute some of Python 2.5 bytecodes. As PyMite is designed to run on 8-bit architectures, it can't do everything Python can [30].

PyMite supports integers, floats, tuples, lists, dictionaries, functions, modules, classes, generators, decorators and closures [31]. It can run multiple stackless green threads and has a mark-sweep garbage collector. Mark-sweep garbage collector differs from the regular by not freeing memory as it goes out of reference, but when the system starts running out of it [32]. PyMite virtual machine does not have a built-in compiler [31].

The fact, that PyMite's github doesn't have active issues [33] is a bit suspicious at first, because it can mean that people aren't using it as actively anymore, resulting worse support and/or solution finding. It could also mean the opposite, that most of the problems are fixed swiftly.

3.6 Armpit Scheme

Armpit Scheme is an interpreter for the Scheme language. It is designed to support multitasking and multiprocessing [34]. Another strongpoint for this engine, is that it supports many ARM (Acorn RISC Machines) based boards [34].

Armpit Scheme's smallest target is the LPC2103 with flash footprint of 32 KB and RAM footprint of 4KB [35].

3.6.1 Scheme

Scheme is a functional programming language, which follows a minimalist design philosophy specifying a small standard core with powerful tools for language extension [36]. It is similar to other members of the Lisp programming languages. Scheme's syntax is based on s-expressions: "parenthesized lists in which a prefix operator is followed by its arguments." [36] So, it's programs consist of sequences of nested lists. Personally, it

doesn't seem as comfortable and may be a bit too uncomfortable to learn (comparing to languages with more familiar syntaxes) for who-ever needs to write a script for the nanosatellite in Scheme.

Scheme is a dynamically typed language, that supports first class procedures, meaning that procedures can be assigned as values to variables or passed as arguments to procedures [36].

3.7 PicoC

PicoC is a tiny C interpreter. At first it was written as a script language for a UAV's (Unmanned Aerial Vehicle) on-board flight system, but is also very suitable for other embedded applications [37]. Since the core C source code is around 3500 lines of code, it doesn't have a complete implementation of ISO C, but has all the essentials [37].

PicoC is promoted to be really small in terms of flash footprint (only a few KB) [37]. It is also feature frozen to keep the size as small as possible [37]. PicoC is very well tested, even the tinier features (but still important) like comments, variable assignments, etc. are tested.

3.8 Angelscript

AngelScript is an extremely flexible cross-platform object oriented scripting library [38] with a familiar C/C++ like syntax [19]. It has a good native binding and functionality to create classes [19]. Good binding means that a class or a function needs to be registered with AngelScript virtual machine and later every script can use it [19]. Most other scripting engines do not provide that. The down side for this, is that each native function needs to be registered before the script can be compiled [19]. This makes sending pre-compiled scripts harder [31].

Instead of pointers, AngelScript uses object handles, which control the lifetime of the object they hold [38]. The documentation refers to them as "smart pointers" [38]. AngelScript has no support for tables, which is a good thing, since the engine is statically typed and it would make using tables uncomfortable [38].

AngelScript’s library is well documented and has an active online forum, which makes problem solving for developers a lot easier.

3.9 TinyTCL

TinyTCL is a smaller version of the tcl language, which can be embedded in to other applications [39]. It compiles to less than 60 KB [39]

Tcl is a high-level dynamically typed language [40]. It was designed to be simple, yet powerful, supporting object-oriented, imperative and functional programming [40]. Tcl has simple syntax and is programmable, so programmers can write command procedures to provide more powerful commands [39].

TinyTCL has a thorough manual focusing on the language, but as a downside, no active community nor a proper homepage can be found.

3.10 Wren

Wren is class-based scripting language, which is suitable for embedding in applications [41]. Wren is small- it has no dependencies, a small library and the virtual machine implementation is under 4000 semicolons [41]. The code is well commented [41] and rather easy to understand. It is fast, because it has a compact object representation and because programs are compiled to bytecode [41].

Wren’s syntax is similar to C like languages, but is simpler and more streamlined [41]. Values in Wren are immutable – once created, they can’t be changed [41]. Wren has four main variable types – numbers, strings and booleans and ranges [41].

Wren has fibers, which are like threads, except they are cooperatively scheduled [41].

The language guide for wren is very straight-forward and easy to follow, thus making construction of scripts easier. Wren is a rather new language/engine, but it already has a lively community and a good homepage.

3.11 My-Basic

My-Basic is a lightweight (memory usage less than 128KB) BASIC interpreter written in C. It is a dynamic typed programmic language, which supports structured grammar and implements prototype-based programming paradigm [42]. My-Basic can be either an embeddable scripting language or just a standalone interpreter [42]. Since My-Basic is under MIT license, it is free to use [42]. Has one of the easiest to read documentation in the list.

It is said to be compatible with RTOS and MCU's (Memory Controller Unit) [42], which might save much time porting it to the board. The whole interpreter is in one header and one source file [42], making the file system a bit cleaner and easier to follow.

My-Basic has integers, reals, strings, booleans, usertype with array and dictionary support [42]. It also offers garbage collection [42]. The language seems to be easy to learn, as it resembles C [42]. That makes creating scripts in the future a lot easier.

3.12 Conclusive table

In table 1, there are theoretical flash and RAM footprints for each language. Also, there is info, if each engine has types, is statically or dynamically typed or whether it has garbage collection.

3.13 Analysis

Next I am going to choose most promising scripting engines to make a performance test, analyse the results and choose the best engine for us. Before that we need to rule out the rest of the choices by complexity, footprint, etc.

The easiest to rule out is AmForth, because it is stated that it cannot be embedded into other programs.

	Has types	Statically/ dynamically typed	Flash footprint[1]	RAM footprint[2]	Garbage Collector
Pawn	No	Dynamically[3]	10 KB [43]	2 KB [43]	No
eLua	Yes	Dynamically	128 KB [43]	32 KB	Yes
Squirrel	No	Dynamically	100 KB [43]	100 KB [43]	Yes
amforth	No	-	8–12 KB	200 byte	No
PyMite	No	Dynamically	55 KB	8 KB	Yes
Armpit Scheme	No	Dynamically	32 KB	4 KB	Not found
PicoC	Yes	Dynamically	Few KB	Unknown	Not found
Angelscript	Yes	Statically	Not found	Not found	Yes
TinyTCL	Yes	Dynamically	60 KB	Not found	Not found
wren	Yes	Dynamically	Not found	4 KB	Yes
My-Basic	Yes	Dynamically	Not found	8 KB	Yes

Table 1. Theoretical info about the scripting engines

Since Scheme's syntax is rather rare, it is a bad choice when different people want to write scripts for the satellite in the future. Learning an unfamiliar syntax takes unnecessary time, when you could write code in some more comfortable language. As Scheme does not provide any other great advantages, there is no point in doing a performance test on it.

Because both AngelScript and Squirrel are written in C++, it is a massive work to write them over to C and include it in our project. Also, the theoretical flash and RAM footprints of Squirrel aren't too promising.

---

[1] Flash footprint as advertized on websites

[2] RAM footprint as advertized on websites

[3] Pawn has only one datatype - „cell"

TinyTCL has no active community and finding information may be challenging. Hence it can be challenging for future developers, who need to develop scripts and/or modify the engine.

ELua does seem to have a very big and active community, but finding necessary information was challenging. While it does support different boards for testing, STM32f3discovery was not one of them. For this reason, I would eliminate eLua from possible choices as there are still better options left.

Wren has a good homepage, where info can be find easily, but it's memory handling doesn't seem to work with our project. Setting up the virtual machine was impossible thanks to this reason.

PicoC and PyMite does seem like good options, but Pawn and My-Basic show more promise as they have better support and more active community. Thus, Pawn and My-Basic were chosen for detailed investigation on satellite's hardware.

4 Implementation

4.1 Environment

Whole testing is in a makefile project in Eclipse IDE (Integrated Development Environment), compiled with GCC, with target ARM, using MinGW toolset. There was also a need to install “gdb” for Eclipse.

4.1.1 Issues fixed during setup

At the end of installing MinGw compiler, default path from installer was added to the system variables. This made compiling the project in Eclipse and in command prompt full of errors. The solution was found thanks to the “where” command, which pointed out, that the “make” command was searched from a false path. Author fixed the setup and updated documentation.

After the project was built, OpenOCD was needed to communicate with the board. The communication did not work. After intensive debugging the solution was to install new USB (Universal Serial Bus) drivers, which fixed the communication protocols. A note was added to the satellite project documentation.

Before debugging, the “gdb” software was needed for eclipse for remote debugging. Finally, after configuring “gdb” paths, the debugging could be started.

4.2 Performance tests on MCU

To finally choose the engine for our satellite, further testing is needed on MCU. The engines will be compared on 3 parameters:

- Flash footprint
- Memory footprint
- Script running speed (similar scripts will be made for all engines)

The MCU for testing is STM32f3, presented on figure 5.

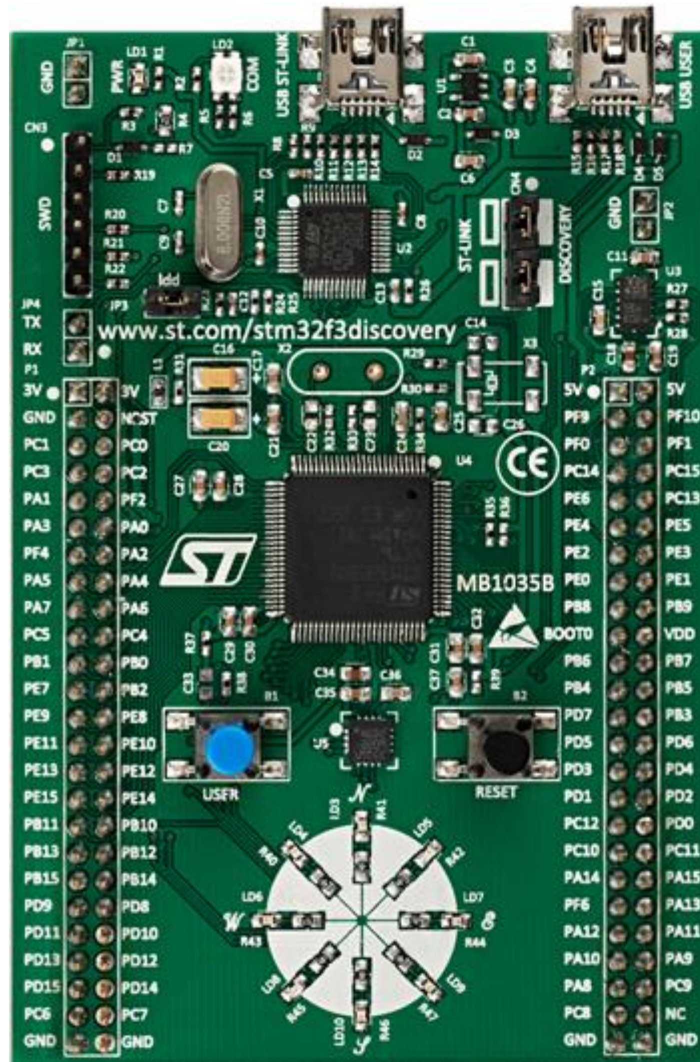

Figure 5. STM32f3 board [44]

4.2.1 Technical issues

Setting up both engines presented their own issues:

- Absence of libraries – many engines needed syscalls.c. Since these functions aren't necessarily needed in this project, dummy implementations were provided to please the compiler. Author used dummy implementation of Joe Ferner [45].

- File input/output – Some of the engines only provided the ability to read the script from a file. Because the file system is yet to be developed for this project, it was impossible. Since the file system is a larger piece of this project, it went out of my scope. The solution was to write custom functions to parse script from a string

variable instead of an actual file. For this, the binary code had to be converted to a string array with bin2header application [46].

- Memory handling – As the native functions for the engines were registered, the system ran out of memory resulting a hard fault exception. This problem happened because FreeRTOS has a bit different memory handling, so using FreeRTOS malloc and free functions solved this issue.

4.2.2 Flash footprint

Flash footprint for the engines can be obtained by observing the project's .map file, created by the compiler. To separate RAM memory addresses from flash memory addresses, we can use the first three characters. From the .map file, the engine's functions memory addresses (not starting with 0x2…) had to be looked, starting with text, rodata, ARM attributes. Adding the differences, it gives us the exact flash footprint.

4.2.2.1 Pawn

The first memory address in text section for pawn is 0x08000724 and because all of the pawn's addresses are in succession, we have to watch the next address, which is not for pawn. This address is 0x08006658. After the subtraction, the result is 24372 B ~ 23.8 KB. The next section is rodata, resulting 2648 B ~ 2.6 KB (0x0800dea4 to 0x0800e8fc). ARM attributes memory addresses go from 0x00000113 to 0x000001ef resulting to 220 B. Total flash footprint is close to 26.6 KB.

4.2.2.2 My-Basic

My-Basic's first text entry is 0x08000702 and the next address not reserved for my-basic starts at 0x080297ac. The subtraction and conversion to decimal results 168106 B. The rodata section starts from 0x080390ec and ends at 0x0803c3bc resulting 13008 B. ARM attributes memory addresses start at 0x0000014a and end at 0x00000181 resulting 37 B. My-Basic has a shocking 176.9 KB flash footprint.

4.2.3 RAM footprint

RAM footprint can also be obtained by observing the project's .map file, but this time looking at the memory address starting with "0x2…". Same as before, adding up the differences, it gives us the RAM footprint.

4.2.3.1 Pawn

Running the script in the abstract machine took less than 128 B of memory, as this is usually the minimum for a task to run, but in our case, it was also the maximum. The binary script variable in header file took space from 0x20000138 to 0x20000908 resulting in 2000 B. This won't be counted as RAM footprint, because later the task can read files.

4.2.3.2 My-Basic

The bss memory section starts at 0x20002afc and ends at 0x20002b08 resulting 12 B.

4.2.4 Performance test

For the performance test, a script should be created for both engines, containing some simple calculations (Fibonacci calculations in our case), because calculations must be made in every script, and some string manipulation, which is needed for communicating with the bus.

Next the time will be measured in MCU ticks[1], obtained from RTOS API (Application Programming Interface). The ticks are asked before and after running the script. Subtracting the starting tick number from the tick number after the script was ran, the performance is retrieved.

4.2.4.1 Pawn

Pawn's abstract machine completed the code in 9 ticks. The code for Pawn's performance test is presented on figure 6. It consists of Fibonacci calculations and simple string manipulations, where needed lines are extracted from a bigger string. This would be a normal case on the satellite as some lines from the log files interest us more than others.

[1] An unit for measuring internal system time [47]

```
#include <string>

main()
    {
    new string[200]
    string = ''line 1\nline 2\nline 3\nline 4\nline 5\nline 6\nline 7\nline
8\n''
    new wantedLines[3] = [0, 2, 5]
    new v = 40
    new wantedLinesLength = strlen(getWantedLines(string, wantedLines, 3))
    if (v > 0)
        return fibonacci(v)
    else
        return wantedLinesLength
    }

fibonacci(n)
    {
    assert n > 0

    new a = 0, b = 1
    for (new i = 2; i < n; i++)
        {
        new c = a + b
        a = b
        b = c
        }
    return a + b
    }

getWantedLines(str[], neededLines[], numOfNeededLines)
      {
      new wantedLines[200], tempStr[200]
      new currentLine = 0, from = 0, to = 0
      for (new i = 0; i < numOfNeededLines;)
      {
             to = strfind(str, ''\n'', false, from)
             if(currentLine == neededLines[i]) {
                    strmid(tempStr, str, from, to + 1, sizeof tempStr)
                    strcat(wantedLines, tempStr)
                    i++;
             }
             from = to + 1
             currentLine += 1
      }
      return wantedLines
      }
```

Figure 6. Testing script for Pawn

4.2.4.2 My-Basic

My-Basic had a shocking initialization time. Just starting up the engine, loading a script, executing it and finally shutting the engine down, took 69 ticks. The loaded script was just 8 characters long comment.

4.2.5 Conclusive table

To conclude the testing table 2 is presented.

	Flash footprint	**RAM footprint**	**Performance**
Pawn	26.6 KB	Less than 128 B	9 ticks
My-Basic	176.9 KB	Less than 128 B	69 ticks

Table 2. Info about tested scripting engines

4.3 Our choice

Pawn has much smaller flash footprint, a bit tinier RAM footprint and it performs a lot better than My-Basic. It also has been successfully implemented on nanosatellites before like ESTCube. My-Basic took much time to be initialised, compared to pawn, which almost took no time at all. Time in this case is important, as the more time is consumed, the more power is consumed for the purpose.

5 Summary

Primary goal of this thesis was to choose and implement a scripting engine for TTÜ satellite, so that we could perform certain actions like taking a picture, debugging system, logging data or updating on the fly.

The first and second chapter of the thesis described the background of the mission. It was explained why a scripting engine is needed in the first place and what our engine must be capable of. In addition, it introduced the CubeSat standard and the satellite's other subsystems.

The third chapter focused on analysing different engines by the information found on web. Features for every language were brought out as well as the theoretical flash and RAM footprint.

The chapter also contained a conclusive table, where the languages were listed with some parameters like, whether it has types, if the language is statically or dynamically typed, its flash and RAM footprint and whether it has garbage collection.

At the end of the third chapter, two scripting engines were chosen amongst eleven for further testing.

The fourth chapter of the thesis was about implementing the scripting engines chosen in the third chapter. Also, problems faced while setting up the environment and while implementing the engine were introduced with solutions.

Two engines were given a flash footprint, RAM footprint and performance tests to compare them and to finally choose the most suitable engine for TTÜ satellite. The chosen scripting engine was Pawn, which was roughly 6 times smaller flash footprint and performed about 7 times faster than my-basic.

The implementation can be found on TTÜ satellites gitlab repository. In addition, an example script was presented so that upgrading the engine and writing new scripts could go faster and smoother.

References

[1] T. Tehnikaülikool, “Satellite programme.” [Online]. Available: https://www.ttu.ee/?id=115635. [Accessed: 04-Apr-2017].

[2] “What is a Scripting Engine? - Definition from Techopedia,” *Techopedia.com*. [Online]. Available: https://www.techopedia.com/definition/26982/scripting-engine. [Accessed: 23-Mar-2017].

[3] A. Pandey, “What is a scripting engine?” [Online]. Available: https://www.quora.com/What-is-a-scripting-engine. [Accessed: 23-Apr-2017].

[4] G. Devarajan, “What is a scripting engine?” [Online]. Available: https://www.quora.com/What-is-a-scripting-engine. [Accessed: 23-Apr-2017].

[5] “Developer Resources,” *CubeSat*. [Online]. Available: http://www.cubesat.org/resources/. [Accessed: 23-Apr-2017].

[6] P. Org, “SOFTWARE ARCHITECTURE AND DEVELOPMENT TOOLS API FOR ATTITUDE CONTROL SYSTEM OF TUT NANOSATELLITE.” Tallinn University of technology, 2017.

[7] “CISC (Complex Instruction Set Computing) Definition.” [Online]. Available: https://techterms.com/definition/cisc. [Accessed: 11-May-2017].

[8] “Complex instruction set computing,” *Wikipedia*. 28-Apr-2017.

[9] “What is CISC (complex instruction set computer or computing)? - Definition from WhatIs.com,” *WhatIs.com*. [Online]. Available: http://whatis.techtarget.com/definition/CISC-complex-instruction-set-computer-or-computing. [Accessed: 11-May-2017].

[10] “Reduced instruction set computer,” *Wikipedia*. .

[11] M. Rouse, “What is RISC (reduced instruction set computer)? - Definition from WhatIs.com,” *Search400*. [Online]. Available: http://search400.techtarget.com/definition/RISC.

[12] “What is risc?” [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html.

[13] EngineersGarage, “CISC & RISC Architecture,” 7-2012. [Online]. Available: https://www.engineersgarage.com/articles/risc-and-cisc-architecture. [Accessed: 11-May-2017].

[14] “RISC vs. CISC.” [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[15] by Toshiba, “What is the difference between NAND and NOR Flash?,” *EDN*. [Online]. Available: http://www.edn.com/design/systems-design/4321127/What-is-the-difference-between-NAND-and-NOR-Flash-. [Accessed: 21-May-2017].

[16] “Flash memory,” *Wikipedia*. 21-May-2017.

[17] “TMS320C6727 Floating-Point Digital Signal Processor | TI.com.” [Online]. Available: http://www.ti.com/product/tms320c6727?keyMatch=c6726&tisearch=Search-EN-Everything#descriptions. [Accessed: 11-May-2017].

[18] E. Priidel, “TTÜ-Mectory Nanosatellite.”

[19] L. Van Winkle, “Game Scripting Languages.” [Online]. Available: https://codeplea.com/game-scripting-languages.

[20] kimhypno, “Difference between pawn and lua?” [Online]. Available: https://facepunch.com/showthread.php?t=1410011.
[21] “ELua Overview.” [Online]. Available: http://www.eluaproject.net/overview.
[22] “elua/elua,” *GitHub*. [Online]. Available: https://github.com/elua/elua. [Accessed: 08-May-2017].
[23] “What is better? Pawn or lua?,” *Multi Theft Auto: Forums*. [Online]. Available: https://forum.mtasa.com/topic/21562-what-is-better-pawn-or-lua/. [Accessed: 09-May-2017].
[24] “Programming in Lua : 2.5.” [Online]. Available: https://www.lua.org/pil/2.5.html. [Accessed: 09-May-2017].
[25] “Squirrel - The Programming Language.” [Online]. Available: http://www.squirrel-lang.org/. [Accessed: 09-May-2017].
[26] “Higher-order function,” *Wikipedia*. 06-May-2017.
[27] “Squirrel (programming language),” *Wikipedia*. 20-Apr-2017.
[28] “AmForth — AmForth.” [Online]. Available: http://amforth.sourceforge.net/. [Accessed: 16-May-2017].
[29] “Forth (programming language),” *Wikipedia*. 01-May-2017.
[30] “PyMite - Python Wiki.” [Online]. Available: https://wiki.python.org/moin/PyMite. [Accessed: 23-Apr-2017].
[31] “Google Code Archive - Long-term storage for Google Code Project Hosting.” [Online]. Available: https://code.google.com/archive/p/python-on-a-chip/. [Accessed: 23-Apr-2017].
[32] A. B. B. } M. 48 and 316 Points 7 3 3 Recent Achievements New Blog Rater Blog Commentator III Blog Commentator II View Profile, “Back To Basics: Mark and Sweep Garbage Collection,” *I know the answer (it’s 42)*. [Online]. Available: https://blogs.msdn.microsoft.com/abhinaba/2009/01/30/back-to-basics-mark-and-sweep-garbage-collection/. [Accessed: 13-May-2017].
[33] “damnit/pymite,” *GitHub*. [Online]. Available: https://github.com/damnit/pymite. [Accessed: 13-May-2017].
[34] “A Scheme Interpreter for ARM Microcontrollers.” [Online]. Available: http://armpit.sourceforge.net/. [Accessed: 23-Apr-2017].
[35] “embedded - What are the available interactive languages that run in tiny memory? - Stack Overflow.” [Online]. Available: http://stackoverflow.com/questions/1082751/what-are-the-available-interactive-languages-that-run-in-tiny-memory. [Accessed: 16-May-2017].
[36] “Scheme (programming language),” *Wikipedia*. 09-Apr-2017.
[37] “zsaleeba/picoc,” *GitHub*. [Online]. Available: https://github.com/zsaleeba/picoc. [Accessed: 23-Apr-2017].
[38] “AngelScript - AngelCode.com.” [Online]. Available: http://www.angelcode.com/angelscript/. [Accessed: 14-May-2017].
[39] “TinyTcl - Tcl for Embedded Applications - Home Page.” [Online]. Available: http://tinytcl.sourceforge.net/. [Accessed: 16-May-2017].
[40] “Tcl,” *Wikipedia*. 30-Apr-2017.
[41] “Welcome – Wren.” [Online]. Available: http://wren.io/. [Accessed: 23-Apr-2017].
[42] “paladin-t/my_basic,” *GitHub*. [Online]. Available: https://github.com/paladin-t/my_basic. [Accessed: 23-Apr-2017].
[43] I. Sünter, “SOFTWARE FOR THE ESTCUBE-1 COMMAND AND DATA HANDLING SYSTEM.” Tartu University, 2017.

[44] "STM32F3DISCOVERY - Discovery kit with STM32F303VC MCU - STMicroelectronics." [Online]. Available: http://www.st.com/en/evaluation-tools/stm32f3discovery.html. [Accessed: 19-May-2017].
[45] J. Ferner, *Contribute to stm32-utils development by creating an account on GitHub*. 2017.
[46] "Binary to Header," *SourceForge*. [Online]. Available: https://sourceforge.net/projects/bin2header/. [Accessed: 20-May-2017].
[47] "CPU time," *Wikipedia*. 31-Dec-2016.