DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Dependable and Modular Command and Data Handling Platform for Small Spacecraft using MicroPython on RODOS

Tejas Kale

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Dependable and Modular Command and Data Handling Platform for Small Spacecraft using MicroPython on RODOS

Zuverlässige und Modulare Steuer- und Datenverarbeitungsplattform für kleine Raumfahrzeuge mittels MicroPython und RODOS

Author:	Tejas Kale
Supervisor:	Prof. Dr. rer. nat. Martin Schulz
Advisors:	M.Sc. Dai Yang
	M.Sc. Sebastian Rückerl
	M.Sc. Vladimir Podolskiy
Submission Date:	15th September 2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15th September 2019 Tejas Kale

Abstract

In this thesis, MicroPython is integrated into RODOS with the goal of using it as a scripting language for next generation of small satellites being developed at the Chair of Astronautics. MicroPython runs as a RODOS application, sharing its stack with the application's stack and uses a statically allocated heap area for its internal memory allocations. The process of running MicroPython scripts, and extending the base MicroPython libraries with user modules is explored in this thesis. Further, a development board using a STM32F4 microcontroller, 2 CAN transceivers and several sensors is designed, manufactured and programmed. The development board enables testing of MicroPython, RODOS, CAN bus communications and the integrated sensors. Additionally, one can also connect external sensors using the exposed interfaces to the microcontroller. Using this board, several performance analyses are carried out to characterize the impact of using an interpreted language like MicroPython as compared to a native machine language like C. The various available code emitters of the MicroPython compiler are also tested and their performance characterized.

Contents

1. Introduction

1.1. Motivation

Command and Data Handling (CDH) subsystem, sometimes referred to as the On Board Computer (OBC), is the heart of small satellites and spacecraft. This subsystem is the main onboard flight computer and handles data from all other subsystems. The CDH provides a functional interface to the satellite for ground based controllers via the communications subsystem. It receives commands from ground, executes them and relays the results back. As the CDH is typically connected to all the other subsystems on the satellite, it can control them. The Electronic Power System (EPS) houses the power system of the satellite, including the batteries, solar cells, circuitry to charge the batteries and power conversion circuitry to provide the other subsystems with their desired voltage levels. The CDH can thus control the power states of all the other subsystems of the satellite using the EPS.

Spacecrafts are commanded using tele-commands from ground. These tele-commands can be statically defined and frozen into the CDH software, or can be defined as a commanding language that allows for dynamically defining complex tasks. The commanding language can be a custom defined language like Plan Execution Interchange Language (PLEXIL) [1] [2] or System Test and Operations Language (STOL). However the development of a custom language presents a significant challenge and risk. As compared to other mainstream scripting languages, such a custom language is difficult to test and validate fully. As an alternative, more mainstream scripting languages such as Bash(for Linux based systems), Javascript, Python etc. can be used. The primary advantage to using these languages is that they are already widely used in other applications, thus have been extensively tested and validated. Additionally these languages are easy to understand as they use a human readable syntax. Users of the satellite who may not be familiar with programming, can thus easily understand the scripts, reducing development effort and errors.

On a hardware level, CDH systems are implemented using small microprocessors capable of hosting complete Linux environment or also on small low power microcontrollers that can only support a basic real time operating system. The size limitations imposed by satellite structure standards (For e.g cubesats [3]) limit the available power for the satellite. Reducing power consumption of the CDH is thus a primary design goal. While Linux based CDH systems are extremely powerful and flexible, they also are quite power hungry. This power consumption can be reduced drastically by using low power microcontrollers like the STM32L4.

Building on experience gathered from the MOVE-II project, where the Linux based CDH system was one of the contributing factors for the insufficient power budget [4], the MOVE-BEYOND project plans to use multiple low power STM32L4 microcontrollers as a modular flight computer with each node running RODOS as its Operating System (OS). MicroPython would be the ideal choice for a scripting language on this system.

1.2. Goal

The main goal of this thesis is to port MicroPython to work as an application on Realtime Onboard Dependable Operating System (RODOS). This will allow the use of MicroPython as a programming language for future Munich Orbital Verification Experiment (MOVE) projects. MicroPython, being and interpreted language, is expected to be worse in performance as compared to a native C implementation. Knowing how much of a performance impact MicroPython has is important as this allows the programmer to know when to use native C code as opposed to MicroPython. Thus MicroPython performance is characterized and compared to a native C implementation. To enable testing and development on realistic hardware, a development board is designed and manufactured. The board is designed in a way so that it can serve as a starting point for future designs.

1.3. Outline

The thesis is structured as follows: Chapter 2 gives the reader a background on RODOS and MicroPython. Chapter 3 summarizes the use of RODOS, scripting languages in other satellite missions. It also looks at the MOVE-II satellite and how scripting was critical in controlling it after launch. Chapter 4 explains how the MicroPython port works on RODOS. Chapter 5 details the hardware design of the development board. Chapter 6 uses the development board to evaluate the performance impacts of using MicroPython. Chapter 7 concludes the thesis by summarizing the results and looks at some possible future enhancements to the MicroPython port and development board.

2. Background

2.1. RODOS

RODOS is a real-time operating system designed for embedded systems where high dependability is desired [5]. It uses a priority based real-time scheduler which is fully preemptive; using round-robin scheduling for threads with the same priority. The microkernel provides support for thread synchronization, resource management and interrupt handling. The Application Programming Interface (API) is implemented in C++ using a object oriented framework.

RODOS provides a middleware layer that enables transparent communication between both remote and local applications. All communication is asynchronous using the publisher - subscriber protocol. Publishers publish messages tagged with a specific unique topic id. Subscribers receive only those messages which they subscribe to. By default all published messages are only routed to the local subscribers. However, using a Gateway, they may be exposed externally; enabling communication across nodes. Logical tasks can be encapsulated into applications. Each application could be composed of multiple threads, event handlers, subscribers and publishers. The idea being to integrate all services related to a specific task into a single unit.

The RODOS API and middleware are implemented independent of any hardware dependent libraries. All hardware dependent functionality is provided by the Hardware Abstraction Layer (HAL). Thus RODOS can be easily ported multiple embedded platforms, using the HAL specific to that platform. Such a separation of hardware and software domains, allows RODOS applications to be very portable. The applications themselves can be run unmodified on multiple target hardware platforms provided a RODOS port exists for it.

2.2. Micropython

MicroPython is an implementation of the Python 3 programming language that is optimized to run on microcontrollers and resource constrained environments. It is implemented in C with a focus on minimizing memory usage and code size, allowing it to run in as little as 8 Kb of RAM with a binary size of only about 70 Kb [6]. MicroPython includes a compiler and a full Python like runtime. Using this, at runtime, it can compile and execute code, or also load and execute pre-compiled code. It supports most of the Python 3 features including arbitrary precision integers, closures, list comprehension, floating point operations and also exception handling. Care is taken to follow the Python 3 syntax as closely as possible, however there are some differences stemming from the need to conserve every bit of memory.

A subset of the Python standard library is also included. Moreover, users can extend MicroPython with either MicroPython libraries or C modules. A few libraries from Python Package Index (PyPI) have already been ported to Micropython. Dependencies on external code are kept to as low as possible. This

enables MicroPython to be ported to almost any platform. Popular ports include the STM32 port which is used on the popular Pyboards, a ESP32 and ESP8266 port, a PIC16 port and even a Javascript port.

2.2.1. MicroPython Internals

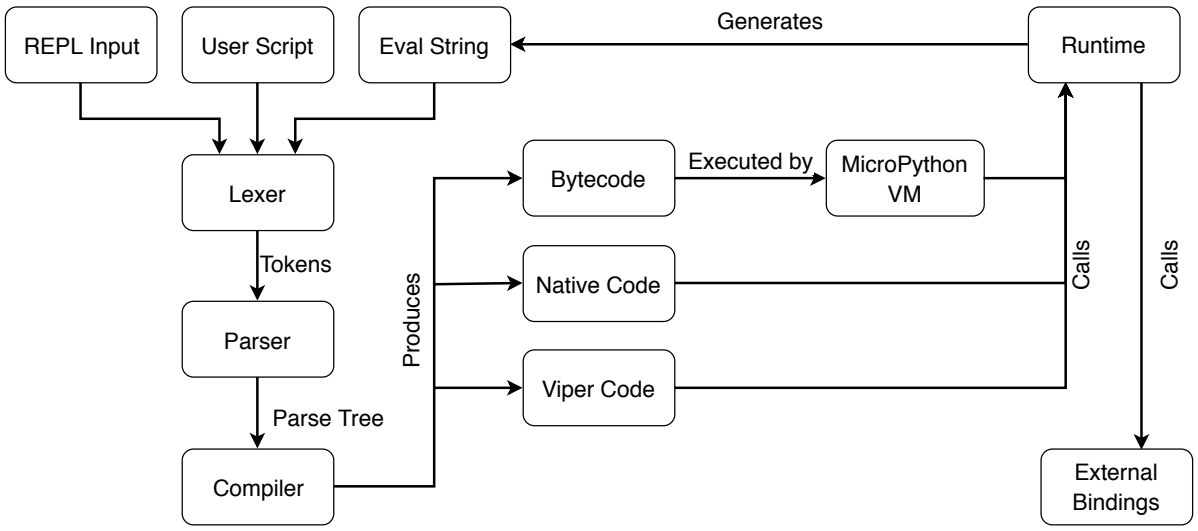


Figure 2.1.: MicroPython internals showing how the compiler, runtime and virtual machine interact with each other.

Figure 2.1 shows the key components of how the MicroPython environment works. The MicroPython runtime, on the top right, is responsible for providing all the necessary components required for executing MicroPython code. It implements the MicroPython built in types like float, tuples, dicts etc. External C bindings with a MicroPython interface allow it to interact with hardware.

The MicroPython Virtual Machine (VM) is responsible for maintaining the global state of the MicroPython environment. Bytecode execution is done by the VM. Some operations like basic arithmetic, binary operations etc. are directly executed by the VM. For other more complex operations like for eg. list sorting, the VM calls helper functions provided by the runtime environment. Hardware access or access to external API's is also provided via wrapper functions that are defined as part of the runtime environment. The runtime call stack is also maintained by the VM. Memory allocations are done by the VM on the heap and when required , a garbage collector is run to try and free up some memory. Pre-compiled code, which can be stored either as native machine code or optimized viper code, can also be loaded and executed directly by the runtime.

2.2.2. Micropython Compiler

On the left side of Figure 2.1 are the parts related to the MicroPython compiler. The lexer is the first stage of the compilation process. It can receive strings as inputs from various sources, like the Read Eval Print Loop (REPL), eval strings generated by the MicroPython runtime environment or user scripts. It breaks down the input into a stream of tokens and passes it to the parser. The parser then generates a

parse tree out of the token stream which is then passed on to the compiler to generate bytecode or native machine code.

```
@micropython.bytecode
def x(a,b):
    return a + b

```

Listing 2.1: A MicroPython function.

```
00:      b0      LOAD_FAST_0
01:      b0      LOAD_FAST_1
02:      db      BINARY_OP_ADD
03:      5b      RETURN_VALUE
```

Listing 2.2: The generated bytecode

The MicroPython compiler features 3 different code generators(called code emitters).Each of the three code emitters offer varying performance levels which are measured in chapter 6.

- **Bytecode Emitter** produces bytecode that can be interpreted by the MicroPython virtual machine. The VM decodes each bytecode and calls the appropriate C runtime function. Listing 2.1 shows an example of a MicroPython function and Listing 2.2 shows the corresponding generated bytecode. This is the default mode, but can be forced with the `@micropython.bytecode` annotation.
- **Native Code Emitter** generates native machine code by inlining the function corresponding to each bytecode. This removes the function call overhead when executing each bytecode, significantly speeding up the execution. Currently x86, x64, armv6, armv7m and xtensa architectures are supported by the native code emitter. This emitter can be invoked with the `@micropython.native` annotation.
- **Viper Code Emitter** builds on top of the Native code emitter by using the fact that many bytecode operations(for eg. the addition shown in Listing 2.1) can be realized with a single machine instruction. Thus rather than calling the appropriate C runtime function, it emits native machine code. However, not all of the MicroPython bytecodes are currently supported. This can be invoked with the `@micropython.viper` annotation.
- **Inline Assembler** allows for embedding assembly code in MicroPython scripts.This code is translated directly to machine instructions at runtime and executed. Currently only a subset [7] of the ARM Thumb-2 instruction set is supported. The inline assembler can be invoked with the `@micropython.asm` annotation.

MicroPython also provides a cross compiler which can be used to compile MicroPython scripts to both bytecode and native/viper code offline. This can be used to precompile user scripts to save storage space and save compile time onboard the microcontroller.

2.2.3. Garbage collection

The MicroPython heap space is managed by the garbage collector. The available memory is divided into a set of blocks, with each block having a 2 bit allocation status attached to it. The garbage collector is a simple mark and sweep garbage collector which maintains an allocation table to keep track of allocated memory. By default the garbage triggers automatically when a memory allocation is requested but no free memory can be found. This may not be desirable as it makes the runtime of the program unpredictable. However to avoid this problem, the GC can be triggered manually.

3. Related work

RODOS due to its real-time capabilities and task communication via a publisher subscriber model is used in various time critical control tasks including quadcopter control, rocket telemetry and telecommand system and satellite CDH software. Scripting languages, being easy to learn and understand have been widely used for task automation in traditional software tasks. However they can also be used as high level programming languages for complex control tasks on embedded devices. Bash, JavaScript and Python have been used for applications ranging from small Internet of Things (IOT) devices to control of large spacecraft. This chapter provides and overview on the use of RODOS, MicroPython and JavaScript in various applications.

3.1. RODOS

3.1.1. TubiX Nanosatellite Platform

The TU Berlin inovative neXt generation satellite bus (TUBiX) is nanosatellite platform series developed at TU berlin for use in satellites with masses ranging from 10 kg(TUBiX10) to 20 kg(TUBiX20). The platform focuses on modularity and reusability as its core design concepts in both the hardware and software domains. In the hardware domain, a base set of common components including the microcontroller and Controller Area Network (CAN) transceiver's are used. Each subsystem is realized as a node, with each node having its own hardware and software. A CAN bus is used as the system bus providing inter-subsystem communication. The use of the same microcontroller on each nodes , allows each of them to run RODOS. All subsystem functionality is split in to applications which exchange data via topics managed and abstracted by the middleware. Inter-subsystem communication is achieved by publishing the topic data on the CAN bus. As the inter-subsystem communication is facilitated via the RODOS middleware, Hardware in the Loop (HiL) testing is carried out by emulating middleware on a Linux host PC. This allows for easy communication between the simulation software and target subsystems [8][5]. The Technosat mission is one such mission that uses the TUBiX20 platform [9].

3.1.2. MAIUS-I

MAIUS-I is a 2 stage sounding rocket developed as a demonstration mission for creation of Bose-Einstein condensates in space The onboard telemetry and telecommand system for the rocket is based on RODOS. The different software modules are implemented as independent RODOS applications(called services) that communicate with each other using RODOS middleware topics. The *BoardConfig* service gathers housekeeping data. The *ConfigManager* is used read the stored configuration databases. The *FlightParameter* service is used for gathering sensor data from Global Positioning System (GPS) and other sensors. The *MassStorage* service provides an interface to the onboard mass storage device. The

Timing service allows for synchronization with a GPS clock. The *Telemetry and Telecommand (TMTC)* service acts as a gateway between the middleware topics and ground support equipment [10].

3.1.3. Other RODOS Use Cases

RODOS as a realtime OS has also been used in various other applications. It was used as the onboard OS for the Autonomous Vision Approach Navigation and Target Identification (AVANTI) experiment. The experiment was launched on the Berlin Infrared Optical System (BIROS) satellite. The software consists of a set of RODOS applications which are run every 30 s [11] [12]. It is also being used as a real-time OS on mini Unmanned Aerial Vehicle (UAV)s and quadcopters [13] [14].

3.2. Use of Scripting Languages in Spacecrafts

3.2.1. MOVE-II Cubesat

Munich Orbital Verification Experiment II (MOVE-II) is a 1-Unit cubesat developed at the Chair of Astronautics. The CDH system for the satellite is powered by a SAMA5D2 processor. A custom Linux kernel is used as the base operating system. The system consists of a multiple background services called daemons. Each subsystem has one daemon and all communication to the subsystem is handled by it. Most subsystems only have a data connection to the CDH. Inter-subsystem communication is handled by using a software bus(called D-bus), which the individual daemons can send and receive messages on. In-orbit commanding and control is achieved through uploading Bash scripts from the ground station over Ultra High Frequency (UHF) link.

After launch, MOVE-II was found to be spinning extremely fast with rotational rates over $500\,^{\circ}\,\mathrm{s}^{-1}$ [4]. The resulting unstable communication line severely limited the uplink bandwidth and a two way communication link was never reliably established. Only very small commands were found to be reliably uplinked and no data download other than the exit status of the script was received back on the ground. However, eventual control of the satellite was achieved by shortening the Bash scripts by relying on Bash parameter expansions and creating short symlinks to frequently used scripts. This allowed the eventual recovery of the satellite by activating the Attitude Determination and Control System (ADCS) system over a period of 7 months.

3.2.2. PyCubed Platform

The PyCubed platform is an Cubesat platform that combines an EPS, CDH, an ADCS and a communications system into a single PC/104 compatible module. The hardware is radiation tested and has been flight proven on the KickSat2 mission. The software architecture is based on CircuitPython which is a fork of the MicroPython project that adds many additional libraries. All of the flight software is written using the Python syntax and is executed by the MicroPython interpreter [15].

3.2.3. LEON 3 MicroPython port

LEON 3 is a 32-bit microprocessor core based on the SPARC-V8 instruction set. It was developed at European Space Research and Technology Centre (ESTEC) which is a part of European Space Agency

(ESA). The processor core is designed in VHDL and is highly configurable to the specific needs of the applications. Fault tolerance and protection against single event latchups is a core design principle fot this microprocessor architecture, making it particularly suitable for use in the harsh space environment [16]. The MicroPython interpreter has been ported to the LEON 3 microprocessor architecture for use as the primary On Board Control Procedure (OBCP) controller. The MicroPython interpreter is highly configurable and controllable, allowing complete control over it by the underlying operating system. Efforts have also been made to formally test and qualify the MicroPython interpreter according to European Cooperation for Space Standardization (ECSS) standards(ECSS-E-ST-70-01C [17]) [18].

3.2.4. James Webb Space Telescope

The James Webb Space Telescope (JWST) is a large space based telescope with a 6.5 meter primary mirror. It images in the infrared range to look at the origins of the universe [19]. The telescope uses and event driven system architecture that uses a Javascript execution environment. All operations related tasks are programmed as scripts written in using Javascript with some custom extensions to allow it to talk to the telescope hardware. The ScriptEase Javascript engine is embedded into the payload computers real time operating system(VxWorks). The engine runs as an self contained application with sufficient isolation to ensure that if the engine itself crashes, other parts of the system are unaffected [20].

4. Micropython Port on RODOS

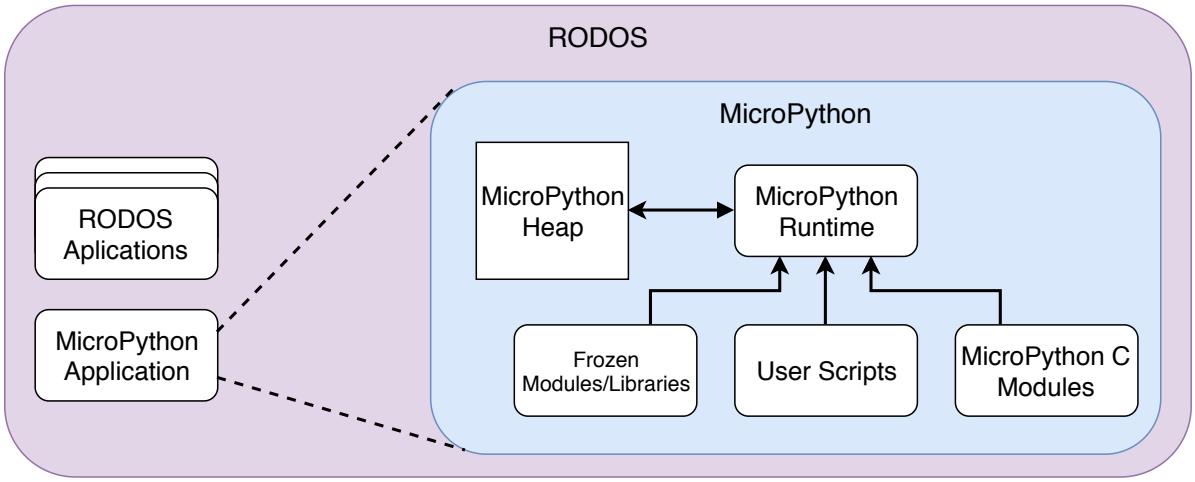


Figure 4.1.: The MicroPython application running inside RODOS.

Figure 4.1 shows the anatomy of a MicroPython instance running as a RODOS application. RODOS applications are a convenient way of encapsulating the MicroPython instance, allowing for a clear isolation of the MicroPython stack and heap space from the other applications running in the system. The MicroPython call stack is shared with the application's stack. This is a direct consequence of the fact that most bytecode operands are realized as C functions.

4.1. MicroPython Build Process

MicroPython itself is compiled as a static library(`libmicropython.a`) using a Makefile. The `Makefile` and `mpconfigport.h` (Appendix B) defines all the necessary configuration required to build MicroPython. Only the target architecture related flags are set by the RODOS build process, enabling MicroPython to be compiled for the correct target platform that RODOS is being compiled for. During the build process, the MicroPython core, comprising of the the VM, compiler and runtime are combined. Weak references to the some of the RODOS API calls needed by MicroPython are left undefined at this state. These are filled up later when the static libraries is linked against the RODOS API.

MicroPython can be extended by defining C modules with MicroPython bindings to make them available for use in the MicroPython environment. Such modules are compiled and stored along with the other Micropython core components. The external C modules with MicroPython bindings are also added to the build for the MicroPython VM.

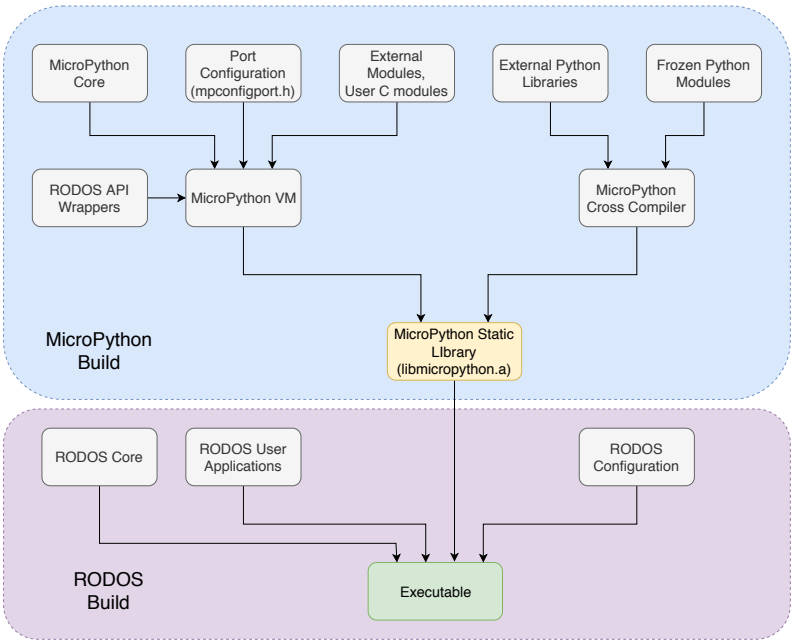


Figure 4.2.: The Micropython build process.

The MicroPython cross compiler is used to precompile the external libraries and user scripts. These precompiled libraries(called frozen modules) are then combined with the MicroPython VM core and archived into a static library. The frozen modules can then be used in MicroPython code by importing them.

The RODOS build process defines the target architecture in its configuration files(`dosis.build`). This enables MicroPython to be automatically built for the target that RODOS is being built for. Figure 4.2 gives an overview of the MicroPython build process.

4.2. Micropython Core Configuration

The file `mpconfigport.h` defines all of the options that control the MicroPython build process. The following options are enabled for the port:

- `MICROPY_ENABLE_COMPILER` : Enables the compiler and adds it to the binary.
- `MICROPY_EMIT_INLINE_THUMB` : Allows the compiler to use the Native and Viper code emitters to generate inline ARM Thumb-2 instructions.
- `MICROPY_REPL_EVENT_DRIVEN` : Configures the REPL to use a event driven architecture, triggered by a character received on the UART.
- `MICROPY_ENABLE_GC` : Enables the garbage collector. The garbage collection threshold is set to 0 to disable automatic garbage collection.
- `MICROPY_MODULE_FROZEN_MPY` : Causes frozen modules to be loaded at runtime.

A more complete configuration file can be found in Appendix B.

4.3. RODOS API Wrappers

A MicroPython port requires the following functions to be defined in the context of the parent OS:

- Print Functions are used by MicroPython to output data and render the REPL environment. MicroPython uses the RODOS `PRINTF` functions for this purpose.
- Along with the RODOS print functions, the UART API is also used to provide character input to the REPL application.
- For time measurement, MicroPython uses the `getNano()`, `mp_hal_ticks_ms()`, `mp_hal_ticks_us()` and `mp_hal_ticks_cpu()` functions. These are mapped to the RODOS `NOW()` function with appropriate conversions for CPU ticks, milliseconds and nanoseconds.
- The `mp_hal_delay_ms()` and `mp_hal_delay_us()` functions use the RODOS `Thread::suspendCallerUntil(time)` function to suspend the MicroPython thread.
- `mp_import_stat()` and `mp_lexer_new_from_file()` provide access to the filesystem API via RODOS.
- Uncaught MicroPython exceptions cause the control flow to call the `nlr_jump_fail()` function which currently prints an error message and enters into a infinite while loop.

4.4. User C Module

User defined C modules can be used to extend the MicroPython language by providing functionality to access hardware and operating system API's. They are implemented as C functions with some boilerplate code to add them the the global MicroPython runtime environment.

Appendix D shows an example module which defines a function to add 2 integers.

5. RODOS Development Board

The RODOS development board was designed to enable rapid testing of RODOS and MicroPython on realistic hardware. By defining a standard design for some of the more commonly used components, it also endeavors to be used as a starting point for future hardware designs. Figure 5.1 shows the various logical components of the development board. This chapter explains the design choices for the various components used on the development board Printed Circuit Board (PCB) and also the overall physical design of the board. Complete schematics for the development board can be found in Appendix A. The first page shows the overall hardware architecture.

5.1. Microcontroller

The STM32F407(specifically STM32F407ZGT6) [21] microcontroller is used as the central processor for the development board. It features an ARM®32-bit Cortex®-M4 CPU with a Floating point unit. It has 1 Mbytes of flash storage and 192 Kbytes of SRAM. It supports 15 communication configurable interfaces including 3 x I^2C, 4 x USART, 3 x SPI, 2 x CAN and a SDIO interface. Also available are up to 136 GPIO ports.

The STM32F4 microcontroller family was chosen primarily because both RODOS and MicroPython have already been ported to the STM32F4 microcontrollers. The microcontroller is programmed using a JTAG interface. The required programming connections are made using the debug header located above the microcontroller. Using this interface. the development board can be programmed using a standard STM32F4 Discovery board. The debug header also includes a UART connector which can used to print debug messages and send data back to the microcontroller. The SDIO interface is connected to a microSD card holder allowing for high capacity data storage to be attached to the development board. Figure 5.1 shows the microcontroller along with all the peripherals around it. Also labelled are the interfaces used by each peripheral to interface with the microcontroller. Page 2 of Appendix A shows the complete schematic of the microcontroller.

5.2. CAN Transceivers

Each of the CAN controllers on the microcontroller(CAN T1 and CAN T2) are connected to a TCAN332D [22] transceiver from Texas Instruments. The TCAN332D also operates at 3.3 V and can be powered from the same 3.3 V power supply as the microcontroller. It supports a 1 Mbps CAN interface and has a wide operating temperature range from −45 °C to 125 °C which is useful for space applications. The 120 Ω CAN termination resistors are connected to ground over a 100 pF capacitor. When plugging multiple boards together, only one of the boards needs to have the termination resistors. Figure 5.3 shows the connections of CAN T1 on the development board.

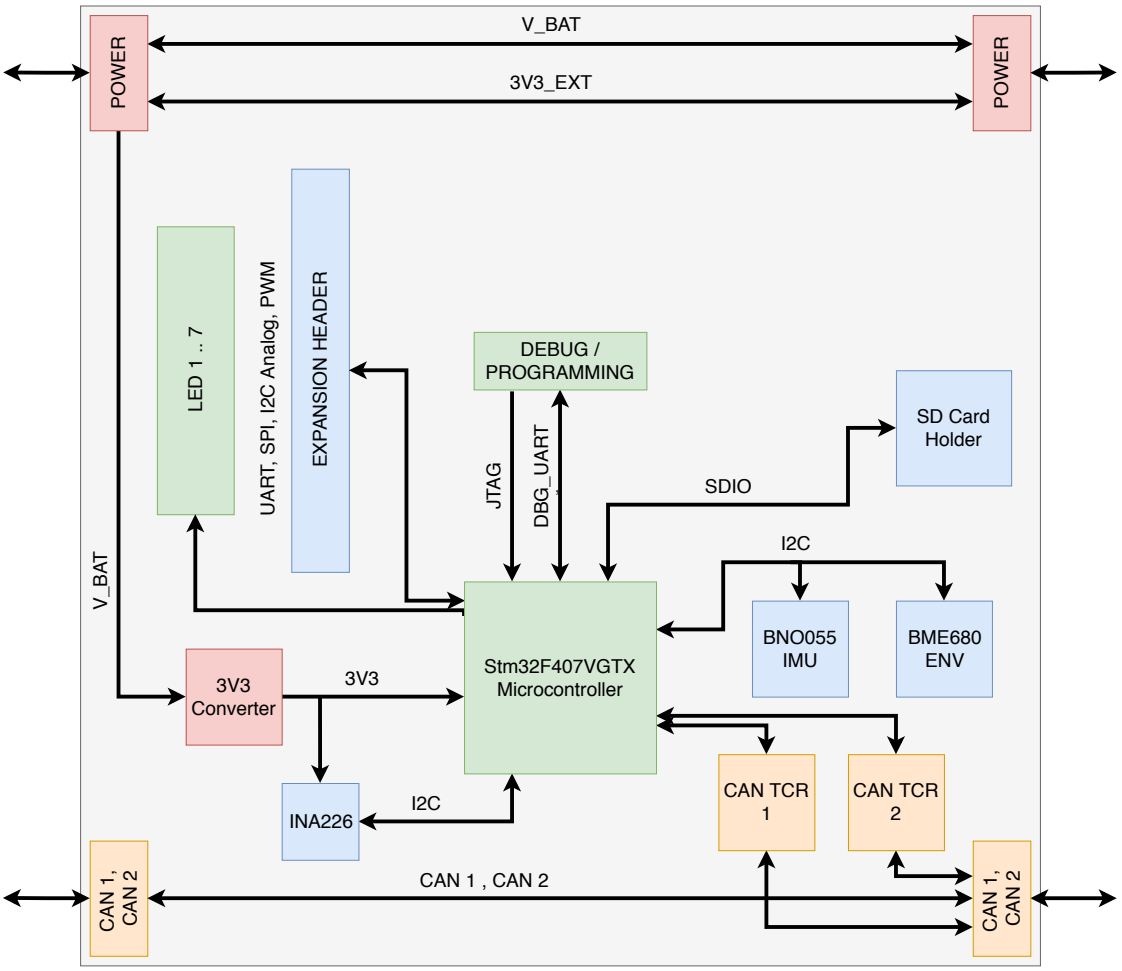


Figure 5.1.: Logical architecture of the development board showing the microcontroller and the various sensors(Not to Scale).

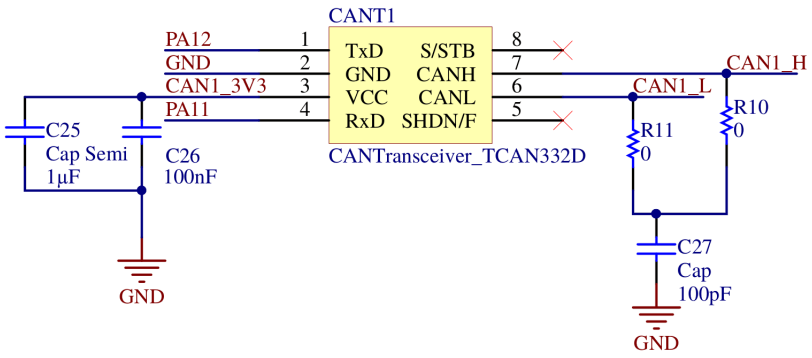


Figure 5.2.: Connections for CAN T1 on the development board.

5.3. Power Supply

The development can be supplied with 2 different power sources: unregulated battery voltage(V_{BAT}) or externally regulated 3.3 V ($3V3_{EXT}$) supply. In case of being supplied with the battery voltage, the voltage needs to be stepped down to 3.3 V for the microcontroller and other peripherals. For this purpose, the LMZM23601 [23] from Texas Instruments is used. It supports an input voltage range of 4 - 36 V, thus supporting most common battery voltages. On its output, it is capable of delivering an output current of upto 1 A. The LMZM23600 was particularly chosen because of its extremely small size. Input and output decoupling capacitors are the only other required external components. A jumper header allows for selection between the onboard 3.3 V supply or the externally regulated 3.3 V supply.

Immediately after the power selection header, a 0.1 Ω shunt resistor is placed. This is used by the INA226(described in section 5.4) current sensor to measure power the consumption of the development board. Both V_{BAT} and $3V3_{EXT}$ are fused using resettable polyfuses that prevent short circuits, can also be reset by removing the fault. A LED placed at the output of the LMZM23601 serves as a power status indicator. Page 3 of Appendix A shows the power supply related components.

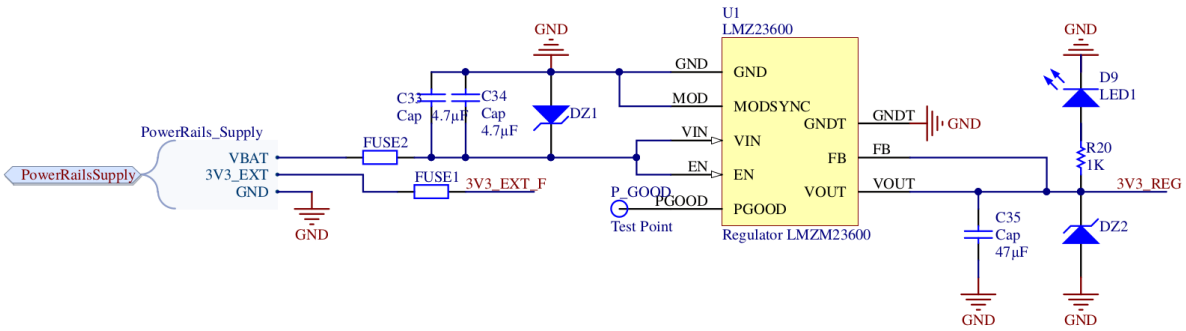


Figure 5.3.: The onboard LMZM23601 DC-DC converter.

5.4. Peripherals

The following peripherals are included on the development board.

5.4.1. Sensors

INA226 Current Sensor

The INA226 [24] current sensor measures the current consumption of the microcontroller and the peripherals by measuring the voltage drop over a small shunt resistor(0.1 Ω). It is connected the the microcontroller over the I^2C bus.

BNO055 IMU

The Bosch BNO055 [25] Inertial Measurement Unit (IMU) integrates a 3-axis accelerometer, a 3-axis gyroscope and a 3-axis magnetometer into one device. Additionally, it also features on-board configurable

sensor fusion which uses the data from all 3 sensors and provides an absolute orientation in the form of a quaternion. It also uses the I^2C interface of the microcontroller.

BME 680 Enviornment Sensor

The BME680 [26] is a low power gas resistance, pressure, humidity and temperature sensor. It has a extremely low power consumption(few mA for gas resistance measurements and few µA for other measurements. It is also connected to the microcontroller using the I^2C bus.

Extension Headers

In order to provide easy access to the microcontroller's peripherals, the pins from the microcontroller are exposed externally via an expansion header. The header has the following digital and analog signals present: Serial Peripheral Interconnect (SPI), Universal Asynchronous Reciever/Transmitter (UART), I^2C, 4 Analog Inputs, 6 Timer I/O's which can be used as Pulse Width Modulation (PWM) outputs, 6 General Purpose Input/Output (GPIO) pins from port F of the microcontroller and 3.3 V power and ground connections. All the extension header and other connectors are shown in detail on page 6 of Appendix A.

5.5. Physical design

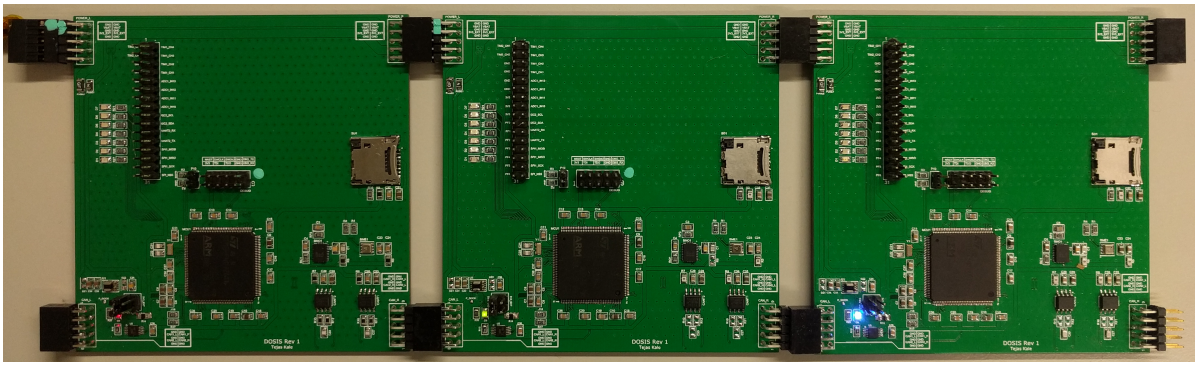


Figure 5.4.: 3 Development boards connected together. They are powered from a laboratory power supply connected at the top left.

Physical design of the development board was guided by the following requirements:

- The development board should have a 10 x 10 cm form factor. As this is the most common size for electronics boards in cubesats [3].
- Multiple boards should be able to be connected to each other, enabling testing of the common CAN bus shared amongst them.
- When stacked, the boards should share a power over the connector.

- When stacked, the boards should be freely accessible to allow easy access to all the connectors.
- The microcontroller, CAN controllers and the power supply components should accommodate as low a area as possible on the PCB, leaving most of the area free for other future components on the board.

Horizontal board to board connectors are placed on the four corners of the PCB. This enables multiple development boards to be connected together as shown in Figure 5.4. Connectors on the top side of the board are used to interconnect the battery voltage(V_{BAT}) and the external 3V3($3V3_{EXT}$) supply across boards. The bottom connectors carry the 2 CAN bus signals. This placement of the connectors allows multiple boards to be plugged in to each other while lying on a flat surface. Debug headers of each board can be easily reached form the top. The shared power and CAN bus signals can also be probed by connecting to the respective connectors on the first or the last board. The whole setup needs only one external power connection, and power is distributed to all the boards over the top connectors. Figure 5.1 shows the rough placement of all the components and the data/power signals between them on the PCB. Figure 5.4 shows a photograph of 3 development boards connected to each other.

The bottom left of the board houses the power supply components. A jumper header is provided to switch between the onboard DC-DC converter(i.e using V_{BAT}) and the external 3V3($3V3_{EXT}$) supply. In the middle is the large microcontroller surrounded by its decoupling capacitors and Realtime Clock (RTC) oscillator crystal. Its programming header is located above it. On the left are the 2 CAN transceivers. These components form the basis of future hardware designs. Additional sensors(described in detail in section 5.4) are located above the CAN transceivers, along with a SD card holder module. The LED's and expansion header is located on the top right of the PCB.

6. Evaluation

MicroPython being an interpreted language, inherently has a performance disadvantage to a pure C/C++ implementation. Knowledge of what overheads exist and what steps can be taken to mitigate them is crucial in determining when MicroPython is the appropriate language of choice. This chapter provides a quantitative analysis of MicroPython performance by measuring the overheads in executing MicroPython code. It also performs a comparative analysis between MicroPython code and a pure C implementation of the same algorithm. Additionally, the power consumption of the development board is also measured using the included INA226 sensor.

6.1. Experimental Setup

The RODOS development board described in chapter 5 is the base hardware for all the performance evaluations. A lab power supply provides 10 V over the V_{BAT} input. The microcontroller is clocked to run at a base clock speed of 168 MHz. A STM32F4 Discovery board is used to program the development board. The Discovery board also serves as a USB to UART converter, which is used to log debug messages to a computer. All measured timings are recorded using this method.

Individual stages of MicroPython execution are measured by timing each stage of the execution(Parsing, compilation and execution) using the `NOW()` function in RODOS. The `NOW()` function uses a hardware timer that ticks with a resolution of 166.66 ns and returns the current time in nanoseconds. As printing data to the debug UART output is a slow operation, care is taken to ensure that the measurements do not include any print statements. All data is printed after the measurement is complete.

RODOS is built with the following compile time options; hardware floating point support is enabled with the `-mfloat-abi=hard` and `-mfpu=fpv4-sp-d16` options, the GCC optimization level is set to `-O3` and all debugging flags are disabled. A more complete build configuration can be found in Appendix B. Only a single RODOS application is loaded which has a single thread that executes the desired test.

6.1.1. Micropython Configuration

The MicroPython VM is built with the following configuration parameters (Configured in `mpconfigport.h`):

- The Thumb and Inline Thumb code emitters are enabled.
- The garbage collector is enabled but its automatic garbage collection threshold is set to 0; effectively disabling automatic garbage collection. The garbage collector is manually called after each test run. This excludes the garbage collector runtime from the measurements.
- MicroPython compiler support is enabled.

- Frozen bytecode and Frozen string modules are enabled.
- Hardware floating point support is enabled.
- Arbitrary precision integer support is enabled and uses the `MICROPY_LONGINT_IMPL_MPZ` implementation.

MicroPython is run as a RODOS application. A separate 32 Kb of heap space is allocated to MicroPython at compile time. All scripts which are measured for their runtime are stored as frozen modules on the microcontroller flash. A full listing of the MicroPython build configuration can be found in Appendix B.

6.2. Experiments

C++ implementations, being compiled directly to machine code are inherently much faster than a interpreted MicroPython scripts. The MicroPython compiler generates bytecode. Each bytecode operand is implemented as a C function call, thus adding a function call overhead to each line of code in Micropython. MicroPython does feature a native code emitter that produces native machine code, however this still does not provide performance comparable to a C++ implementation as it only avoids the function call overhead. The viper optimized native code, as described in chapter 2, is only useful in certain situations like register access and bit operations. Additionally, in order to use the viper optimization, one must step away from standard Python syntax and use special data type hints. This section described the various performance evaluations done and how they help in characterizing MicroPython performance.

6.2.1. Overheads due to Parsing and Compilation

As described in chapter 2, before a MicroPython program is executed, it needs to be parsed and then compiled to machine code. In order to characterize the performance of the MicroPython parser and compiler, a MicroPython script of increasing length was parsed and compiled. The script simply defines a MicroPython function which is a repeated series of additions of 2 variables. Listing 6.1 shows the function being parsed and compiled. Starting with a single addition statement, at each iteration, the same line is appended to the end of the function and the resulting longer script is parsed from scratch. As we are only in characterizing the parsing and compilation time, the function is never called. Thus the execution time remains minimal and can be ignored for this test.

```
1 def f():
2     a = 1
3     b = 2
4     c = a + b
5     c = a + b
6     ...
```

Listing 6.1: The MicroPython function used to test the parsing and compilation performance. The `c = a + b` line is repeated multiple times to generate a long script.

Figure 6.1 shows the time required for each of the three steps(parsing, compilation and execution) as a function of the script length. A very linear relationship is observed. In Figure 6.2 we can clearly see that the percentage of time required for each of the 3 steps remains constant with increasing script size.

6.2.2. Matrix Multiplication : CPU Bound Application

In order to measure the execution speed of a Central Processing Unit (CPU) bound workload, a 3 x 3 matrix multiplication is implemented using MicroPython arrays. The multiplication is carried out using nested loops and the 3 required matrices are pre-allocated. Listing 6.2 and Listing 6.3 lists the respective MicroPython and C functions that are timed.

```
def matrix_mul():
a = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
b = [[3, 2, 1], [6, 5, 4], [9, 8, 7]]
c = [[0 for row in range(3)] for col in range(3)]
x = 0.0;
for reps in range(1000):
    for i in range(3):
        for j in range(3):
            c[i][j] = 0
            for k in range(3):
                c[i][j] = c[i][j] + a[i][k]* b[k][j]
matrix_mul()
```

Listing 6.2: MicroPython 3x3 matrix multiplication.

```
uint64_t matrix_mul() {
    uint64_t t = NOW();
    uint32_t a[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    uint32_t b[3][3] = {{3, 2, 1}, {6, 5, 4}, {9, 8, 7}};
    uint32_t c[3][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
    for (uint16_t n = 0; n < 1000; n++) {
        for (uint8_t i = 0; i < 3; i++) {
        for (uint8_t j = 0; j < 3; j++) {
            c[i][j] = 0;
            for (uint8_t k = 0; k < 3; k++) {
            c[i][j] += a[i][k] * b[k][j];
            }
        }}}
    return NOW() - t;
}
```

Listing 6.3: C++ 3x3 matrix multiplication

Each is run independently of each other and the time taken to execute the script or function is measured as an average of 1000 runs. Figure 6.3 compares the runtime of the MicroPython with the

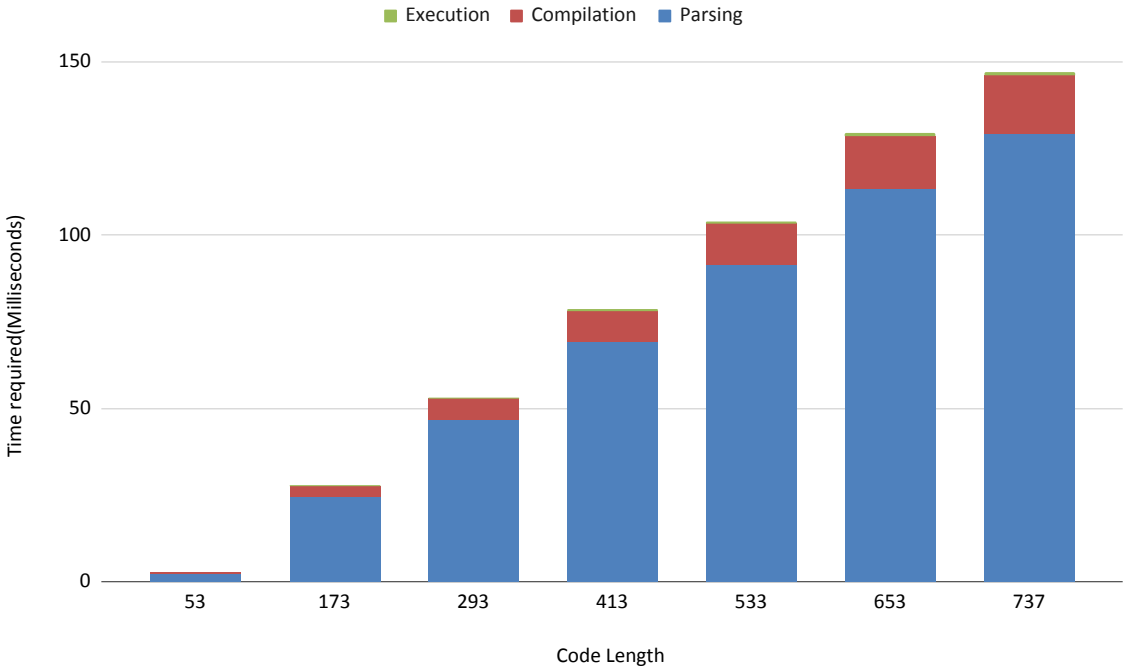


Figure 6.1.: Total time required for parsing, compiling and executing Micropython code.

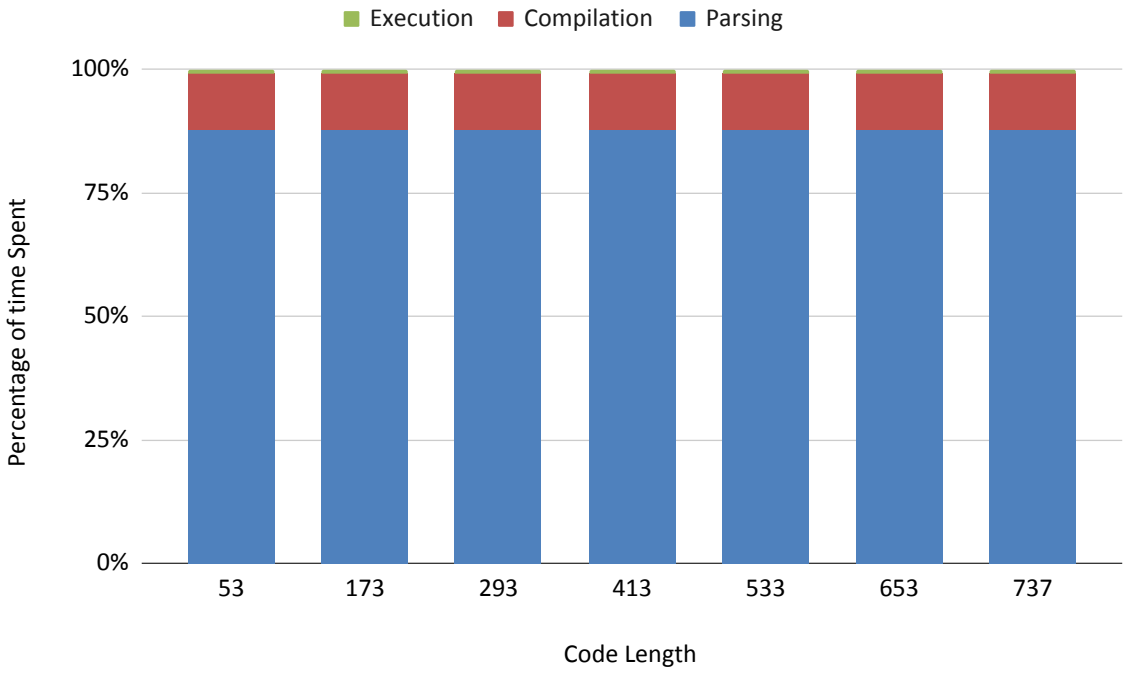


Figure 6.2.: The time spent(in %) in the various stages of Micropython code execution as a function of code size.

C++ implementation. For the MicroPython implementation, all 3 types of code emitters are separately measured. Additionally a MicroPython C - module which implements the matrix multiplication in C code is also measured.

Bytecode is the worst measuring at about 0.383 ms per matrix multiplication. Moving to the native code emitter, we see a 40% performance improvement(0.231 ms) as compared to bytecode. The native and viper code emitters have a very similar runtime as the viper optimizations have no effect on standard MicroPython objects. They are rather meant to be used for pointer and register manipulation operations. The RODOS and MicroPython C module implementations are significantly faster. They require only 0.0120 ms and 0.0125 ms per matrix multiplication; a gain of 96%!

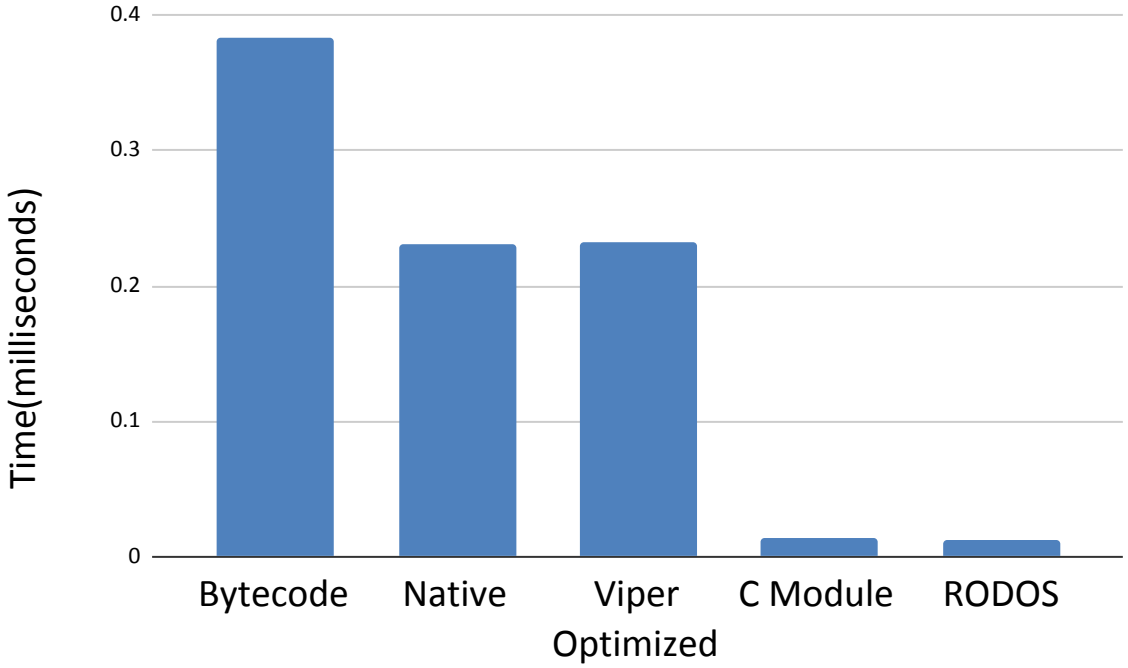


Figure 6.3.: Comparison of runtime of a 3x3 matrix multiplication implemented in MicroPython and RODOS

6.2.3. Large Array Sorting : Memory Bound Application

Memory bound operations like accessing large static arrays stored in flash memory are also compared. For this, a 500 element array containing random integers generated using the Python random library. A simple bubble sort algorithm is implemented in MicroPython, C++ and as a MicroPython module. Additionally the inbuilt list sort function is also included in the timings. It should be noted that the inbuilt sort function uses the quicksort algorithm rather than bubble sort. Listing 6.4 shows the bubble sort script in MicroPython and Listing 6.5 shows the sorting implemented in C++.

```
1 # 'large' is an array with 500 integers.
```

```
large  = [708, 383, 644, ....]
@micropython.native
def bubble_sort():
    for i in range(500):
        for j in range(i, 500):
            if(large[i] > large[j]):
                large[i], large[j] = large[j], large[i]
bubble_sort()
```

Listing 6.4: Bubble sort implemented in MicroPython and compiled using the native code emitter.

```
static uint32_t large[500] = {708, 383, 644, ...};
uint64_t bubble_sort() {
    uint64_t t = NOW();
    for (uint16_t i = 0; i < 500; i++) {
        for (uint16_t j = 0; j < 500; j++) {
        if (large[i] > large[j]) {
            uint32_t temp = large[i];
            large[i] = large[j];
            large[j] = temp;
        }
        }
    }
    return NOW() - t;
}
```

Listing 6.5: C++ bubble sort implementation.

Figure 6.4 shows the runtime of the array sort script using different implementations. We see a 15% increase in performance when using the native code emitter as compared to bytecode. Bytecode needs 3.69 s, Native needs 3.18 s and the Viper emitter needs also 3.18 s for sorting the array. Again, the Native and Viper emitters are closely matched. The C++ bubble sort implementation is more than 98% faster and needs only 50 ms. For memory bound operations, the native code emitter has a much lesser impact on the runtime as compared to CPU bound operations.

6.2.4. Micropython Script execution overhead

MicroPython scripts in this project are primarily aimed at being used in time critical scenarios, where the script is executed as a reaction to some external event like an interrupt. Execution of a script includes several overheads like script load, parse and compilation. When executing a pre-compiled frozen script, the parsing overhead is absent. Additionally, if any import statements are present, the module in question is loaded, parsed and compiled if required, before proceeding with script execution. In realistic scenarios, scripts would likely contain multiple import statements and multiple functions, class definitions etc; However we can establish a baseline by measuring the time required for importing a single MicroPython

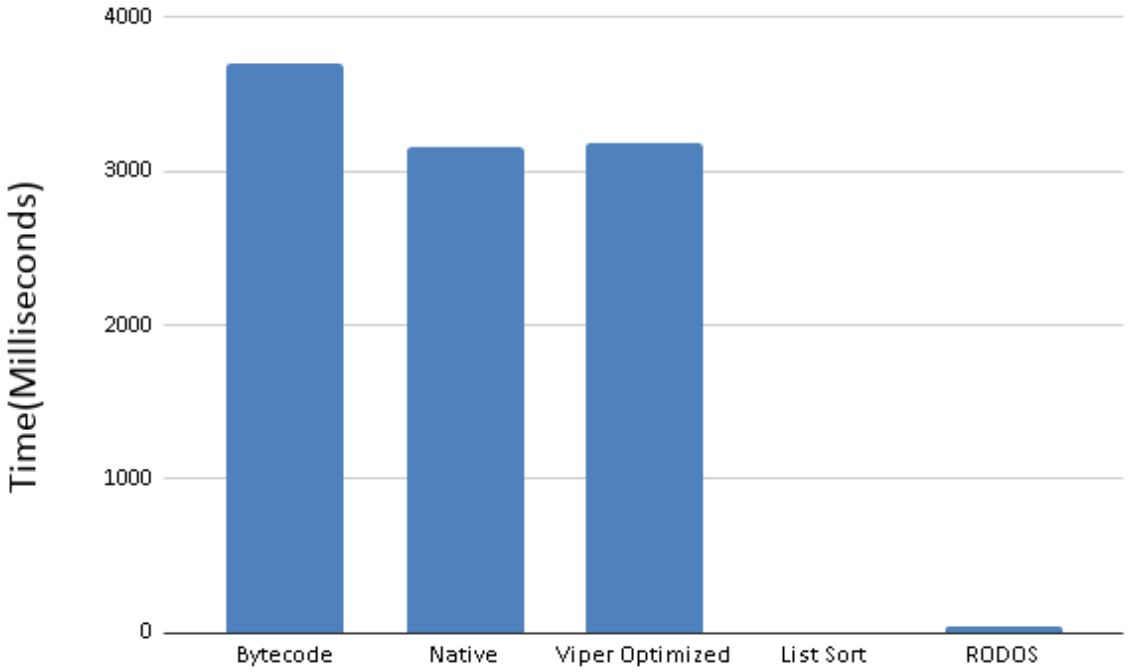


Figure 6.4.: Comparison of runtime of sorting a 500 element array.

module. Using the test module which defines a single function(Appendix D), it is found that on an average, importing the test module requires 1.82 ms.

6.2.5. Garbage Collection

In all of the above tests, automatic garbage is disabled to prevent it from influencing the measurements. This also makes sense in time critical applications where a automatic garbage collection can cause the runtime to be unpredictable. To schedule manual garbage collection , its useful to know how much time the garbage collector requires on average. The garbage collector runtime was measured by triggering a manual garbage collection after each of the above mentioned tests. The average runtime of the garbage collector was 0.341 ms. For a worst case analysis, the script shown in Listing 6.6 was run. The script creates new MicroPython dictionary objects and appends it to an array, exhausting almost all of the 32 Kb heap space. After the script ends, the array goes out of scope and subsequently all the dict objects and the array itself are garbage collected. The average worst case runtime for the garbage collector is 1.943 ms.

```
1 a = []
2 for i in range(1,500):
3     a.append({10:20.0})
```

Listing 6.6: Script for measuring worst case runtime for the garbage collector.

6.3. Discussion

MicroPython, at first glance, seems to be orders of magnitude worse in performance. However its primary use is as a scripting language to control and automate tasks rather than to perform computationally heavy tasks. MicroPython is perfectly adequate for reading a sensor value and saving it once every second. However if the task is computationally expensive, like grabbing an image from a camera and compressing it before saving, a pure MicroPython implementation would be a severe performance bottleneck.

MicroPython makes several optimization options available to the user. When constrained to only MicroPython code, the Native code emitter, depending on the situation, offers up to 40% improvement in performance. When dealing with bit operations, register manipulations and other low level operations, the Viper code emitter performs a little better than the Native code emitter. Performance critical and computationally expensive code can be outsourced to MicroPython C modules. Their performance is on par with a plain C/C++ implementation.

The script compilation and execution overheads can be also be reduced in certain cases. The MicroPython cross compiler allows for offline precompilation of MicroPython code to either bytecode or native machine code. At runtime, for frequently used scripts, a caching strategy can be used. The compilation would only be required on the first run and subsequent runs will be faster.

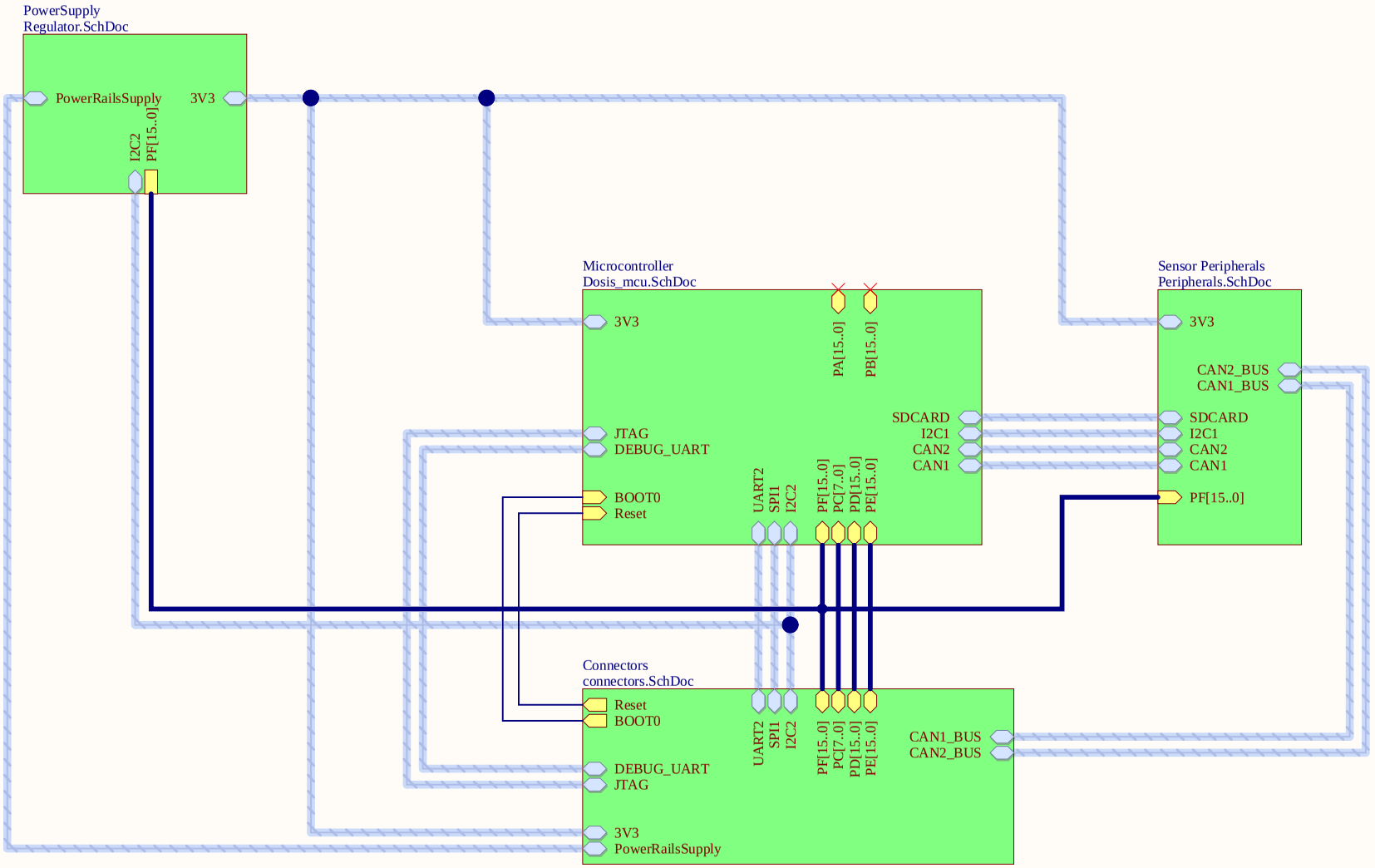
7. Conclusion and Future Work

The MicroPython port for RODOS was programmed with the objective of using MicroPython as a scripting language for future MOVE projects. The hardware development board proved very useful in the development and testing of not only MicroPython but also other RODOS applications. The board has been setup as a testing and development environment with remote access. The CAN transceivers integrated onto the development board have enabled testing of the CAN bus and its latencies.

During the development several areas of future work were identified. While the MicroPython port works, it still is running in a sandboxed environment as it has no access to hardware. As we want to rely on the hardware abstractions provided by RODOS, a possible solution to this problem would be to implement MicroPython wrappers for the RODOS API. This will enable MicroPython applications to fully utilize the capabilities of the hardware while still interacting with the other parts of the system in a safe manner. A starting point to this would be the RODOS Topic API. Porting this would immediately enable MicroPython applications to communicate with other applications by publishing or receiving messages.

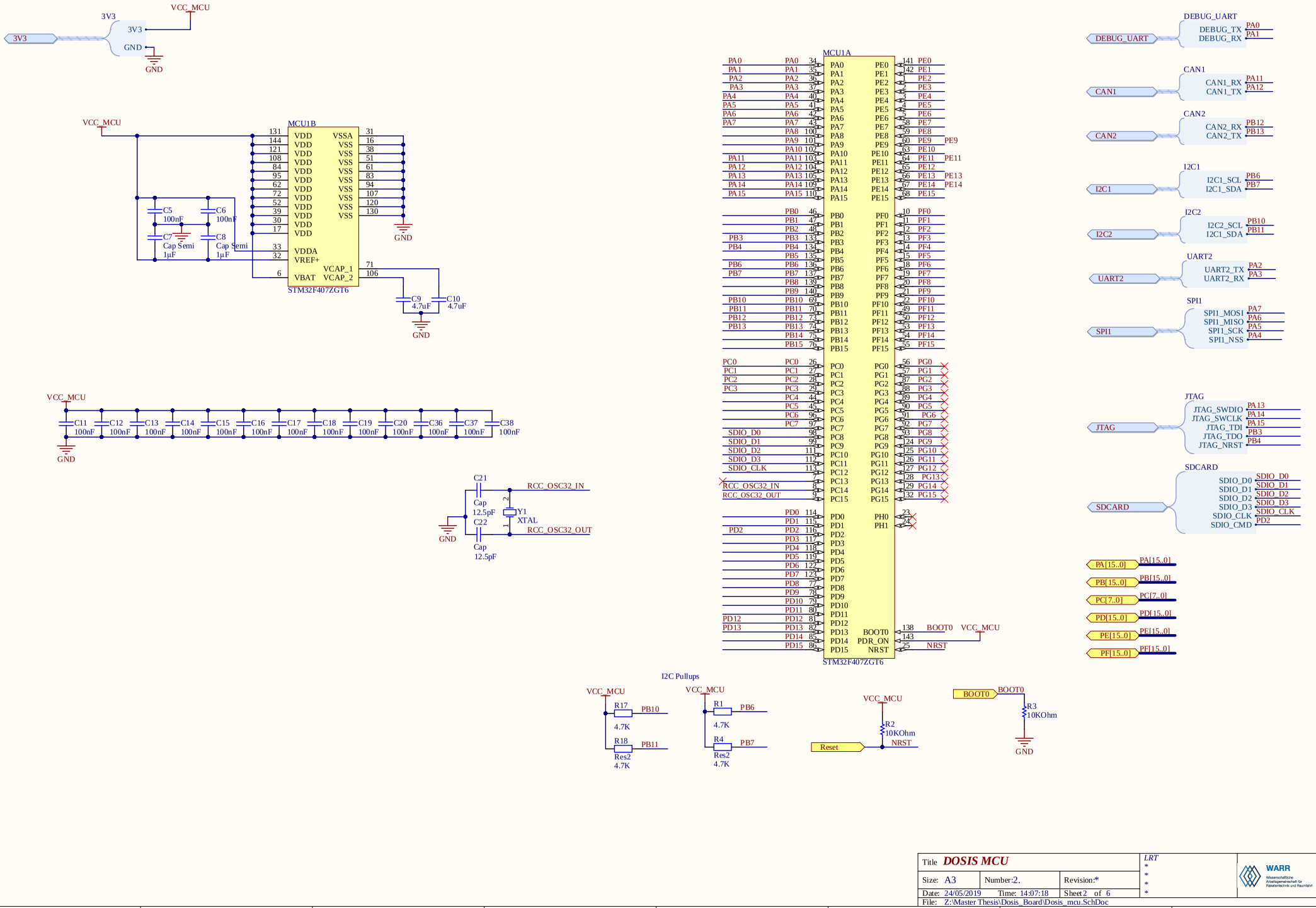
The STM32**F**4 microcontroller used currently on the development board consumes a lot of power. The power consumption of the board can be lowered by changing the microcontroller to a STM32**L**4. Both the microcontrollers are pin compatible with only a few changes to some of the power pins. Thus, from a hardware point of view, migrating to a SMT32L4 microcontroller is easy. However RODOS does not have a working STM32L4 port yet. The current development board does not feature an external clock oscillator for providing the system clock. While the internal oscillator works fine for testing, it is not very stable over long periods of time or varying temperature. Adding an external oscillator crystal will improve the stability of the clock.

A. Development board schematics



Title	***DOSIS TOPLEVEL***		*LRT* *Tejas Kale*
Size: A4	Number: 1.	Revision: 1	
Date: 24/05/2019	Time: 14:07:18	Sheet 1 of 6	
File: Z:\Master Thesis\Dosis_Board\Dosis_toplevel.SchDoc			

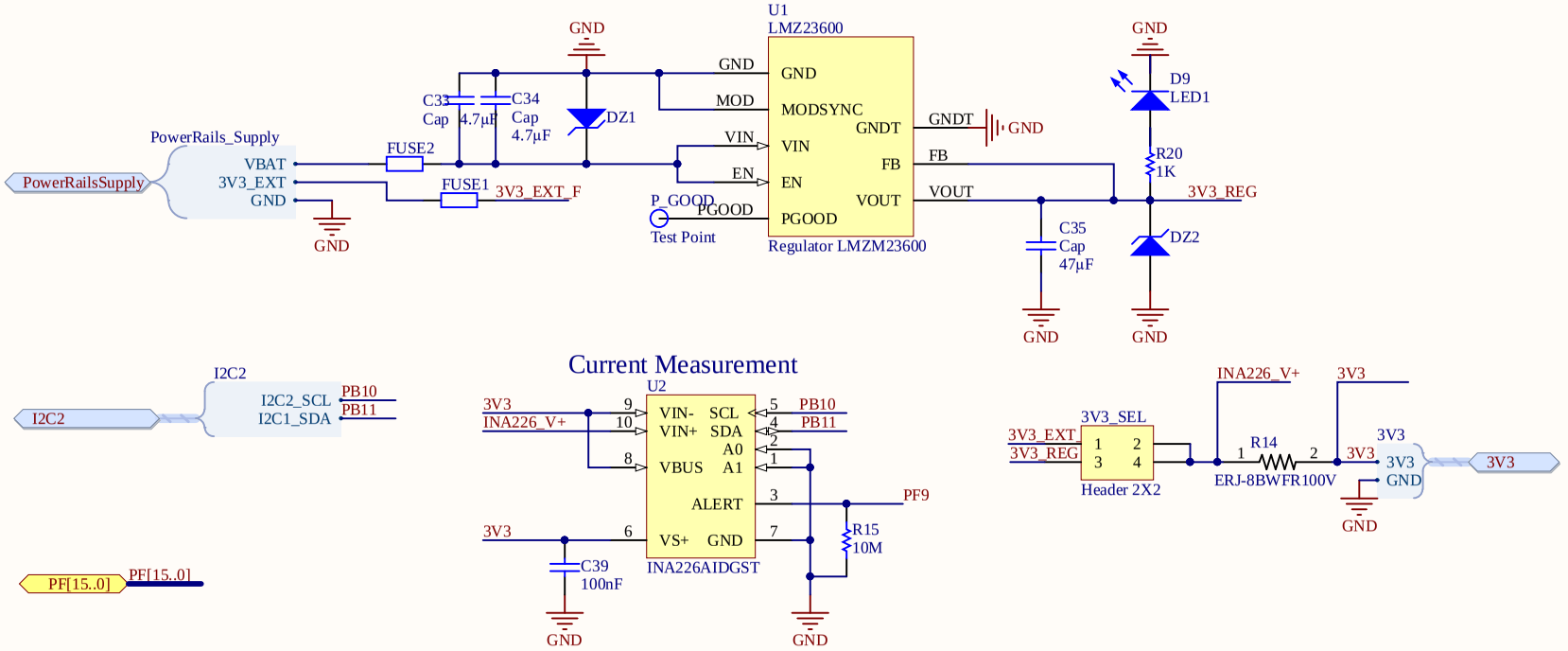
WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt



Title	**DOSIS MCU**	
Size: A3	Number:2.	Revision:*
Date: 24/05/2019	Time: 14:07:18	Sheet 2 of 6
File: Z:\Master Thesis\Dosis_Board\Dosis_mcu.SchDoc		

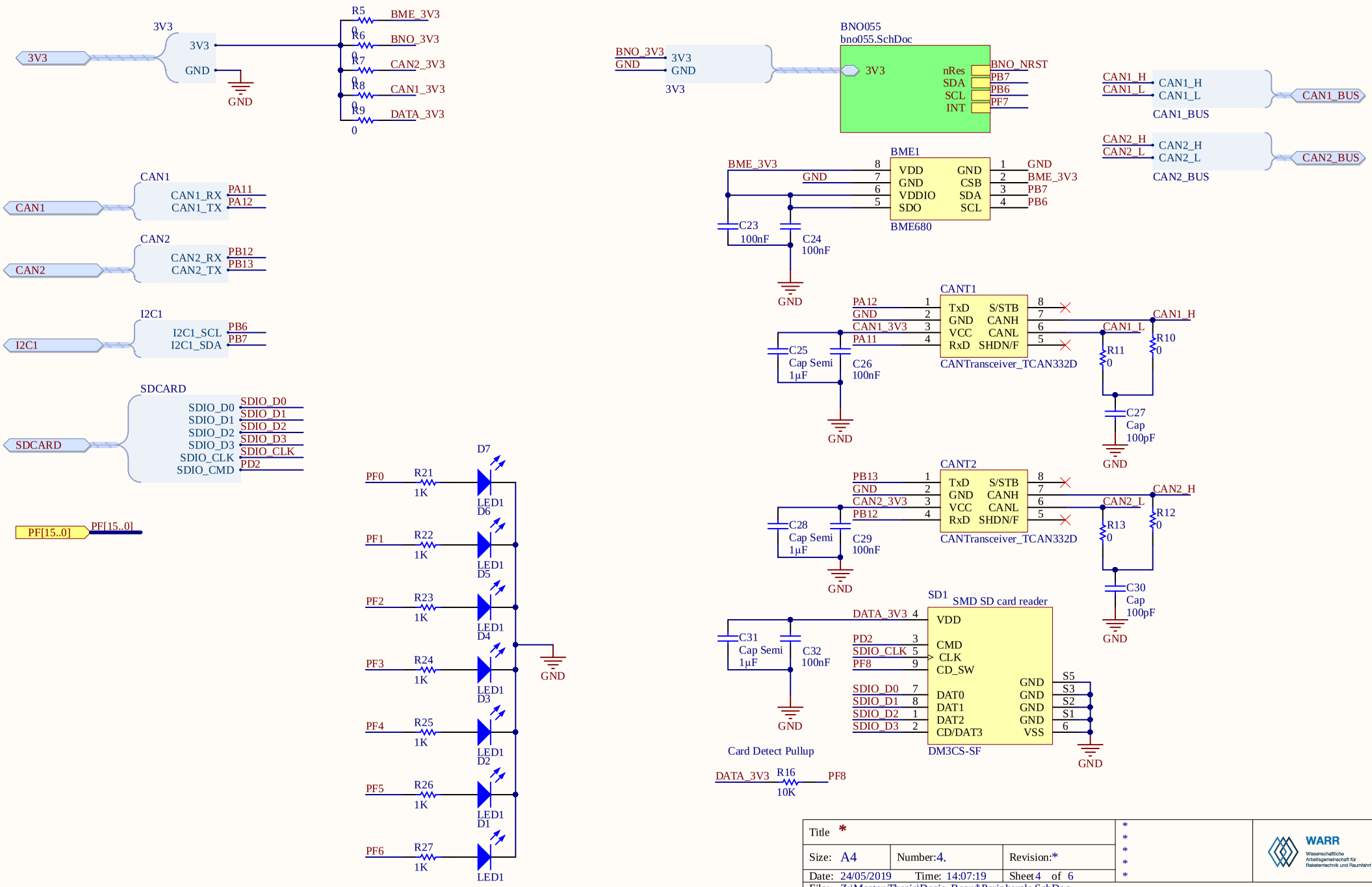
LRT
*
*
*
*

WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt



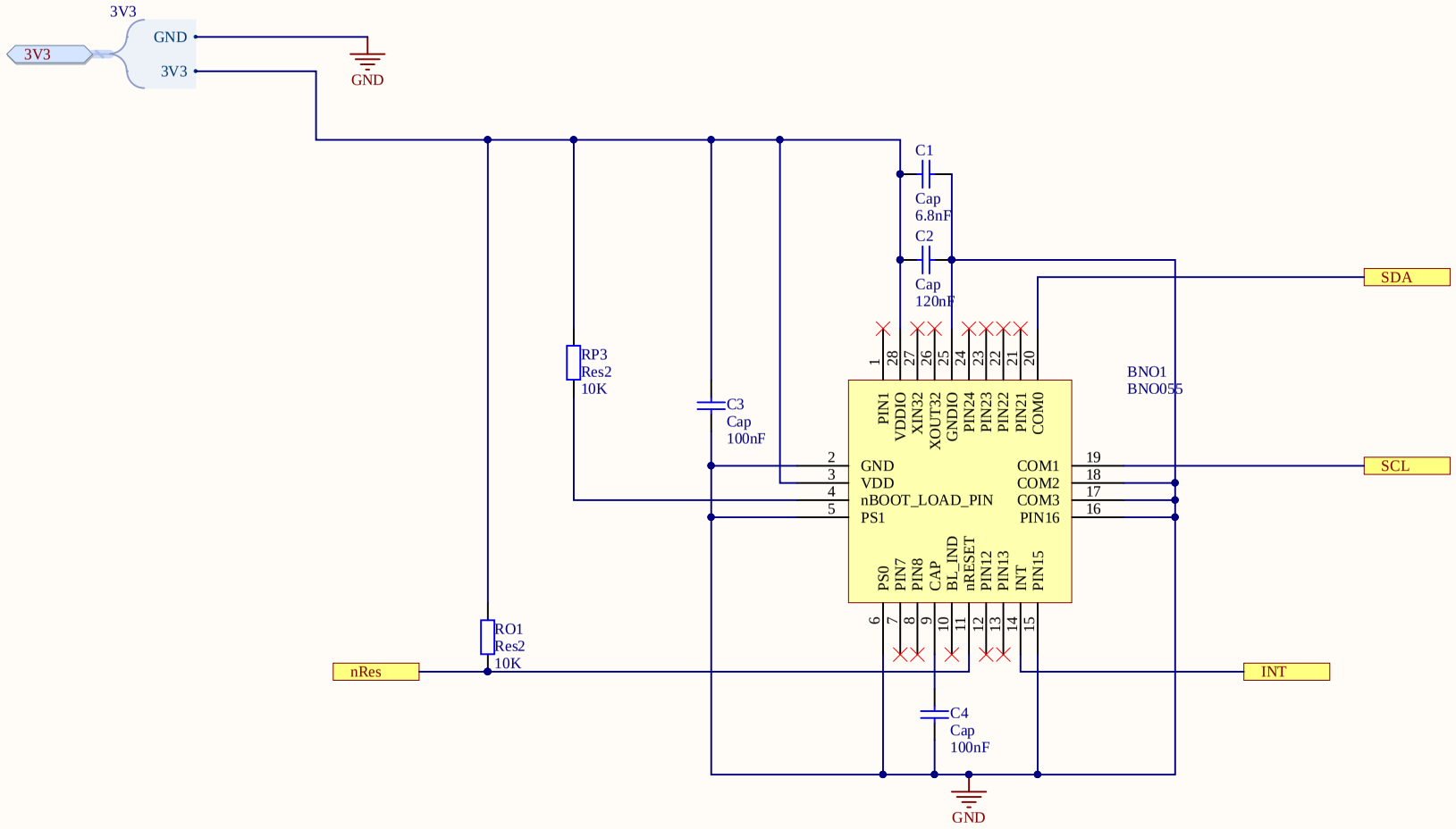
Title	***3V3 Regulator & Power Measurement***		*LRT* * * * *
Size: A4	Number:3.	Revision:*	
Date: 24/05/2019	Time: 14:07:19	Sheet 3 of 6	
File: Z:\Master Thesis\Dosis_Board\Regulator.SchDoc			

WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt



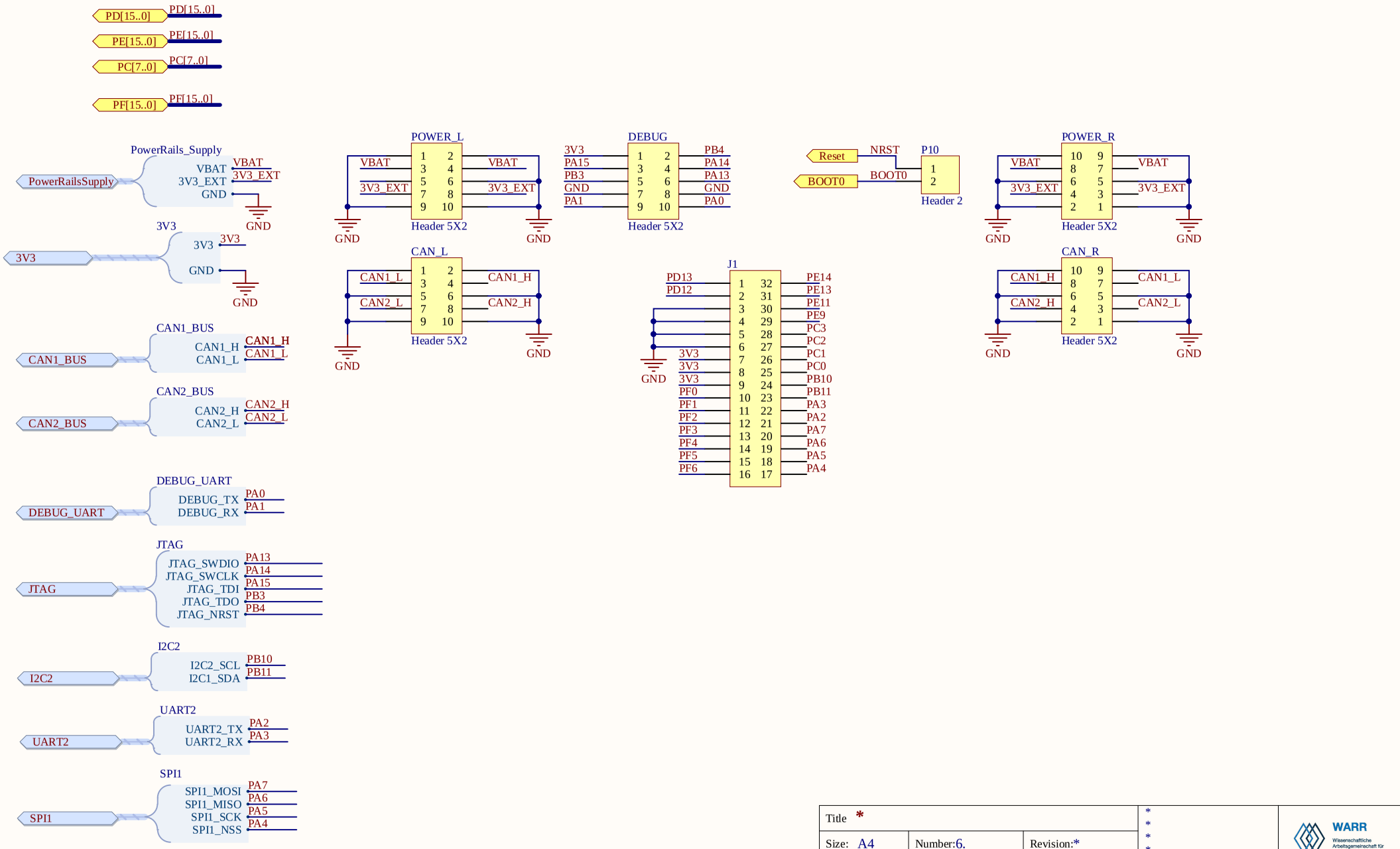
Title *			
Size: A4	Number:4.	Revision:*	
Date: 24/05/2019	Time: 14:07:19	Sheet 4 of 6	
File: Z:\Master Thesis\Dosis_Board\Peripherals.SchDoc			

WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt



Title *			
Size: A4	Number:5.	Revision:*	
Date: 24/05/2019	Time: 14:07:19	Sheet 5 of 6	
File: Z:\Master Thesis\Dosis_Board\bno055.SchDoc			

WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt



Title *			*
Size: A4	Number: 6.	Revision: *	*
Date: 24/05/2019	Time: 14:07:20	Sheet 6 of 6	*
File: Z:\Master Thesis\Dosis_Board\connectors.SchDoc			*

WARR
Wissenschaftliche Arbeitsgemeinschaft für Raketentechnik und Raumfahrt

B. Micropython Configuration

Listing B.1 is the main file controlling the Micropython build.

```
// Options to control how MicroPython is built
// Checkout  micropython/py/mpconfig.h for a brief
// description of each option.

#define MICROPY_HW_BOARD_NAME       "DOSIS"
#define MICROPY_HW_MCU_NAME         "STM32F407"


#define MICROPY_ALLOC_PATH_MAX      (128)
#define MICROPY_ALLOC_PARSE_CHUNK_INIT (16)
#define MICROPY_EMIT_THUMB          (1)
#define MICROPY_EMIT_INLINE_THUMB   (1)
#define MICROPY_COMP_MODULE_CONST   (0)
#define MICROPY_COMP_CONST          (0)
#define MICROPY_COMP_DOUBLE_TUPLE_ASSIGN (1)
#define MICROPY_COMP_TRIPLE_TUPLE_ASSIGN (1)
#define MICROPY_MEM_STATS           (0)
#define MICROPY_DEBUG_PRINTERS      (0)
#define MICROPY_ENABLE_GC           (1)
#define MICROPY_GC_ALLOC_THRESHOLD  (0)
#define MICROPY_ENABLE_COMPILER     (1)
#define MICROPY_REPL_EVENT_DRIVEN   (1)
#define MICROPY_HELPER_REPL         (1)
#define MICROPY_HELPER_LEXER_UNIX   (1)
#define MICROPY_ENABLE_SOURCE_LINE  (0)
#define MICROPY_ENABLE_DOC_STRING   (0)
#define MICROPY_ERROR_REPORTING     (MICROPY_ERROR_REPORTING_TERSE)
#define MICROPY_BUILTIN_METHOD_CHECK_SELF_ARG (1)
#define MICROPY_PY_ASYNC_AWAIT      (0)
#define MICROPY_PY_BUILTINS_BYTEARRAY (1)
#define MICROPY_PY_BUILTINS_DICT_FROMKEYS (1)
#define MICROPY_PY_BUILTINS_MEMORYVIEW (1)
#define MICROPY_PY_BUILTINS_ENUMERATE (1)
#define MICROPY_PY_BUILTINS_FILTER  (1)
```

```
35 #define MICROPY_PY_BUILTINS_SET     (1)
36 #define MICROPY_PY_BUILTINS_FROZENSET (0)
37 #define MICROPY_PY_BUILTINS_REVERSED (1)
38 #define MICROPY_PY_BUILTINS_SLICE   (1)
39 #define MICROPY_PY_BUILTINS_PROPERTY (0)
40 #define MICROPY_PY_BUILTINS_MIN_MAX (1)
41 #define MICROPY_PY_BUILTINS_STR_COUNT (1)
42 #define MICROPY_PY_BUILTINS_STR_OP_MODULO (1)
43 #define MICROPY_PY___FILE__         (1)
44 #define MICROPY_PY_GC               (1)
45 #define MICROPY_PY_ARRAY            (1)
46 #define MICROPY_PY_ATTRTUPLE        (1)
47 #define MICROPY_PY_COLLECTIONS      (1)
48 #define MICROPY_PY_MATH             (1)
49 #define MICROPY_PY_CMATH            (1)
50 #define MICROPY_PY_IO               (0)
51 #define MICROPY_PY_STRUCT           (1)
52 #define MICROPY_PY_SYS              (0)
53 #define MICROPY_MODULE_FROZEN_MPY   (1)
54 #define MICROPY_MODULE_FROZEN_STR   (1)
55 #define MICROPY_QSTR_BYTES_IN_HASH  (1)
56 #define MICROPY_QSTR_EXTRA_POOL     (mp_qstr_frozen_const_pool)
57 #define MICROPY_CPYTHON_COMPAT      (0)
58 #define MICROPY_LONGINT_IMPL        (MICROPY_LONGINT_IMPL_MPZ)
59 #define MICROPY_FLOAT_IMPL          (MICROPY_FLOAT_IMPL_FLOAT)
60 #define MICROPY_USE_INTERNAL_PRINTF (0)
61
62 // Extended modules
63 #define MICROPY_PY_UTIMEQ           (1)
64 #define MICROPY_PY_UTIME_MP_HAL     (1)
65 #define MODULE_EXAMPLE_ENABLED      (1)
66 // #define MICROPY_PY_URANDOM          (1)
67
68
69 #define MP_PLAT_PRINT_STRN(str, len) mp_hal_stdout_tx_strn_cooked(str, len)
70
71
72
73
74 #define MICROPY_PORT_BUILTIN_MODULES \
75    { MP_OBJ_NEW_QSTR(MP_QSTR_example), (mp_obj_t)&example_user_cmodule }, \
76     { MP_OBJ_NEW_QSTR(MP_QSTR_argex),    (mp_obj_t)&mp_module_argex }, \
```

```
77
78
79 #define MICROPY_MAKE_POINTER_CALLABLE(p) ((void*)((mp_uint_t)(p) | 1))
80
81 // This port is intended to be 32-bit, but unfortunately, int32_t for
82 // different targets may be defined in different ways - either as int
83 // or as long. This requires different printf formatting specifiers
84 // to print such value. So, we avoid int32_t and use int directly.
85 #define UINT_FMT "%u"
86 #define INT_FMT "%d"
87 typedef int mp_int_t; // must be pointer size
88 typedef unsigned mp_uint_t; // must be pointer size
89
90
91
92 // extra modules available to uPy.
93 extern const struct _mp_obj_module_t mp_module_mymodule;
94 extern const struct _mp_obj_module_t mp_module_argex;
95
96
97 #define MP_STATE_PORT MP_STATE_VM
98 #define MICROPY_PORT_ROOT_POINTERS \
99     const char *readline_hist[8];\
100
101
102 typedef int mp_int_t; // must be pointer size
103 typedef unsigned int mp_uint_t; // must be pointer size
104
105 typedef long mp_off_t;
106
107 #include <alloca.h>
```

Listing B.1: Micropython configuration header : mpconfigport.h

C. Rodos Configuration

Listing C.1 and Listing C.2 show the RODOS build time configuration. The `dosis.build` defines the target environment including the cross compiler to use(`arm-none-eabi-g++`, version 8.2.1 20181213 (release) [gcc-8-branch revision 267074] in this case) and any other target specific options that are needed.

```
target_hw_flags = ['-mcpu=cortex-m4', '-mthumb', '-mfloat-abi=hard', '-mfpu=
    fpv4-sp-d16', '-fno-pic']
target_c_args = target_hw_flags + [
  '-gdwarf-2',
  '-DHSI_VALUE=16000000',
  '-DSTM32F40_41xxx',
  '-DUSE_STM32_DISCOVERY',
  '-DUSE_STDPERIPH_DRIVER',
  '-O3'
]
target_cpp_args = target_c_args + ['-fno-rtti', '-fno-exceptions']
target_link_args = target_hw_flags + [
  '-T' + meson.current_source_dir() + '/scripts/stm32_flash.ld',
  '-nostartfiles',
  '-nodefaultlibs',
  '-nostdlib',
  '-Xlinker',
  '--gc-sections',
  '-fno-unwind-tables',
  '-fno-asynchronous-unwind-tables'
]
target_file_ext = '.elf'
```

Listing C.1: Meson build configuration for RODOS

```
[binaries]
c = '/opt/arm-toolchain/latest/bin/arm-none-eabi-gcc'
cpp = '/opt/arm-toolchain/latest/bin/arm-none-eabi-g++'
ld = '/opt/arm-toolchain/latest/bin/arm-none-eabi-gcc'
ar = '/opt/arm-toolchain/latest/bin/arm-none-eabi-ar'
strip = '/opt/arm-toolchain/latest/bin/arm-none-eabi-strip'

```

```
[host_machine]
system = 'dosis'
cpu_family = 'arm'
cpu = 'armv7'
endian = 'little'

[properties]
board = 'dosis'
```

Listing C.2: The Development board target definition build file.

D. Example MicroPython C Module

```c
// Include required definitions first.
#include "py/obj.h"
#include "py/runtime.h"
#include "py/builtin.h"

// This is the function which will be called from Python as example.add_ints(a, b).
STATIC mp_obj_t example_add_ints(mp_obj_t a_obj, mp_obj_t b_obj) {
    // Extract the ints from the micropython input objects
    int a = mp_obj_get_int(a_obj);
    int b = mp_obj_get_int(b_obj);

    // Calculate the addition and convert to MicroPython object.
    return mp_obj_new_int(a + b);
}
// Define a Python reference to the function above
STATIC MP_DEFINE_CONST_FUN_OBJ_2(example_add_ints_obj, example_add_ints);

// Define all properties of the example module.
// Table entries are key/value pairs of the attribute name (a string)
// and the MicroPython object reference.
// All identifiers and strings are written as MP_QSTR_xxx and will be
// optimized to word-sized integers by the build system (interned strings).
STATIC const mp_rom_map_elem_t example_module_globals_table[] = {
    { MP_ROM_QSTR(MP_QSTR___name__), MP_ROM_QSTR(MP_QSTR_example) },
    { MP_ROM_QSTR(MP_QSTR_add_ints), MP_ROM_PTR(&example_add_ints_obj) },
};
STATIC MP_DEFINE_CONST_DICT(example_module_globals, example_module_globals_table);

// Define module object.
const mp_obj_module_t example_user_cmodule = {
    .base = { &mp_type_module },
    .globals = (mp_obj_dict_t*)&example_module_globals,
};
```

```
34
35 // Register the module to make it available in Python
36 MP_REGISTER_MODULE(MP_QSTR_example, example_user_cmodule,
       MODULE_EXAMPLE_ENABLED);
```

Listing D.1: An example module which shows how to define a MicroPython module in C. The Module simply defines a function which adds 2 integers and returns the result,

List of Figures

Listings

Bibliography

[1] V. Verma, T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, and K. Tso. "Plan execution interchange language (PLEXIL) for executable plans and command sequences". In: *International symposium on artificial intelligence, robotics and automation in space (iSAIRAS)*. 2005.

[2] T. Estlin, A. Jónsson, C. Pasareanu, R. Simmons, K. Tso, and V. Verma. "Plan execution interchange language (PLEXIL)". In: (2006).

[3] A. Mehrparvar, D. Pignatelli, J. Carnahan, R. Munakat, W. Lan, A. Toorian, A. Hutputanasin, and S. Lee. "Cubesat design specification rev. 13". In: *The CubeSat Program, Cal Poly San Luis Obispo, US* 1.2 (2014).

[4] S. Rueckerl, D. Meßmann, N. Appel, J. Kiesbye, F. Schummer, M. Faehling, L. Krempel, T. Kale, A. Lill, G. Reina, P. Schnierle, S. Wuerl, M. Langer, and M. Luelf. "First Flight Results of the MOVE-II CubeSat". In: *Session I: A Look Back: Lessons Learned*. Technical University of Munich. June 2019. URL: https://digitalcommons.usu.edu/smallsat/2019/all2019/49/.

[5] S. Montenegro and F. Dannemann. "RODOS-real time kernel design for dependability". In: *DASIA 2009-DAta Systems in Aerospace*. Vol. 669. 2009.

[6] MicroPython. *MicroPython code statistics*. 2019. URL: http://micropython.org/resources/code-dashboard/ (visited on 09/13/2019).

[7] MicroPython. *MicroPython inline assembler supported instructions*. 2019. URL: http://docs.micropython.org/en/v1.9.3/pyboard/reference/asm_thumb2_index.html#asm-thumb2-index (visited on 09/13/2019).

[8] K. Gordon. *A flexible attitude control system for three-axis stabilized nanosatellites*. Vol. 2. Universitätsverlag der TU Berlin, 2018.

[9] M. Barschke, K. Großekatthöfer, and S. Montenegro. "Implementation of a nanosatellite on-board software based on building-blocks". In: *Proceedings of the Small Satellites Systems and Services Symposium. Porto Pedro, Spain*. 2014.

[10] A. Stamminger, J. Ettl, J. Grosse, M. Hörschgen-Eggers, W. Jung, A. Kallenbach, G. Raith, W. Saedtler, S. Seidel, J. Turner, et al. "MAIUS-1—vehicle, subsystems design and mission operations". In: *Proceedings of the 22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research*. ESA Special Publication. 2015, pp. 183–190.

[11] J.-S. Ardaens and G. Gaias. "Integrated Solution for Rapid Development of Complex GNC Software". In: *Proceedings of the Workshop on Simulation for European Space Programmes (SESP)*. European Space Agency, ESTEC Noordwijk, The Netherlands. 2015.

[12] M. Wermuth, G. Gaias, and S. D'Amico. "Safe picosatellite release from a small satellite carrier". In: *Journal of Spacecraft and Rockets* 52.5 (2015), pp. 1338–1347.

[13] S. Montenegro, Q. Ali, and N. Gageik. "A review on distributed control of cooperating mini UAVs". In: (2015).

[14] Q. Ali. "Distributed Control of Cooperating Mini UAVs". In: (2016).

[15] M. Holliday, A. Ramirez, C. Settle, T. Tatum, D. Senesky, and Z. Manchester. "PyCubed: An Open-Source, Radiation-Tested CubeSat Platform Programmable Entirely in Python". In: ().

[16] F. Sturesson, J. Gaisler, R. Ginosar, and T. Liran. "Radiation characterization of a dual core LEON3-FT processor". In: *2011 12th European Conference on Radiation and Its Effects on Components and Systems*. IEEE. 2011, pp. 938–944.

[17] E. C. for Space Standardization. *Spacecraft on-board control procedures*. Apr. 2010. URL: https://ecss.nl/standard/ecss-e-st-70-01c-on-board-control-procedures/.

[18] D. George, D. Sanchez, and T. Jorge. "Porting of MicroPython to LEON Platforms". In: *Data Systems in Aerospace* (2016).

[19] P. A. Sabelhaus and J. E. Decker. "An overview of the James Webb space telescope (JWST) project". In: *Optical, Infrared, and Millimeter Space Telescopes*. Vol. 5487. International Society for Optics and Photonics. 2004, pp. 550–563.

[20] V. Balzano and D. Zak. "Event-driven James Webb Space Telescope Operations using on-board JavaScripts". In: *Advanced Software and Control for Astronomy*. Vol. 6274. International Society for Optics and Photonics. 2006, 62740A.

[21] S. Microelectronics. *ARM Cortex-M4 32b MCU+FPU, 210DMIPS, up to 1MB Flash/192+4KB RAM*. DocID022152. Rev. 8. ST Microelectronics. Sept. 2016.

[22] T. Instruments. *TCAN33x 3.3-V CAN Transceivers with CAN FD (Flexible Data Rate)*. SLLSEQ7D. Texas Instruments. Apr. 2016.

[23] T. Instruments. *LMZM23600 36-V, 0.5-A Step-Down DC/DC Power Module in 3.8-mm × 3-mm Package*. SNVSB53B. Texas Instruments. May 2019.

[24] T. Instruments. *INA226 High-Side or Low-Side Measurement,Bi-Directional Current and Power Monitor with I2C Compatible Interface*. SBOS547A. Texas Instruments. Aug. 2015.

[25] B. Sensortec. *BNO055, Intelligent 9-axis absolute orientation sensor*. BST-BNO055-DS000-14. Rev 1.6. Bosch. June 2016.

[26] B. Sensortec. *BME680, Low power gas, pressure, temperature and humidity sensor*. BST-BME680-DS001-03. Rev 1.3. Bosch. July 2019.