Tampere University

Donal Johny

# EXPLORING IMPLEMENTATION OF RUST-BASED SECURE OPERATING SYSTEMS ON RISC-V MP SOC

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
Examiners: Prof. Timo Hämäläinen
Henri Lunnikivi, PhD researcher
January 2024

# Abstract

Donal Johny: EXPLORING IMPLEMENTATION OF RUST-BASED SECURE OPERATING SYSTEMS ON RISC-V MP SOC
Master's thesis
Tampere University
Master's Degree Programme in Software Development
January 2024

The rapid advancement of technology in contemporary times has catalysed the proliferation of specialised processors, mainly driven by the open-source nature of the RISC-V ISA. This shift from royalty fees has rendered RISC-V an appealing foundation for processor design, creating intricate Systems-on-Chip (SoCs) featuring diverse subsystems and processor cores tailored to various functions.

As software platforms for embedded applications become increasingly sophisticated, the limitations of controlling them with bare metal drivers have become more pronounced. Operating systems customized for microcontrollers have become vital, especially for enabling embedded systems like Internet of Things (IoT) devices. However, these operating systems require more advanced features, such as dynamic memory allocation, flexible concurrency, and fault isolation, to cope with the intricate demands of these devices.

Tock, an innovative operating system designed specifically for low-power platforms, addresses these challenges by leveraging specialized hardware protection mechanisms and the advanced type-safety features of the Rust programming language. This combination produces a reliable multi-programming environment for microcontrollers, ensuring optimal performance through software fault isolation, robust memory protection, and efficient management of dynamic application workloads.

This thesis delves into the intricate process of porting Tock OS to a custom RISC-V processor, aiming to gain a deeper understanding of the complexities involved in this task. While the actual porting will not be implemented in this study, the research provides a comprehensive breakdown of the requirements for a custom Tock OS port, encompassing hardware and software aspects. The thesis scrutinises the Tock OS architecture and project structure, examining existing works on porting Tock OS to various platforms.

The customised ballast chip produced by the SoC hub is scrutinised within the scope of this investigation. To enhance research and education, Tampere University collaborates with partnering corporations to explore SoC development, including its architecture and toolchain. The feasibility report presented details of the porting process, outlining the approach, challenges faced, and findings and recommendations derived from this study.
**Keywords:** Rust, MPSoC, Tock OS, SoC Hub.

# Preface

I want to extend my deepest thanks to all the individuals who have contributed in various ways to help me complete my thesis. Firstly, I am grateful to Prof. Timo Hämäläinen for allowing me to work at SoC Hub and providing valuable guidance throughout my research. Secondly, I want to acknowledge Henri Lunnikivi for offering unwavering support and assistance, especially during the more challenging aspects of my study. Additionally, I would like to thank Tom Szymkowiak for his consistent help with Ballast hardware. Lastly, I am indebted to the Tock OS team, including Amit Levy and Lawrence Esswood from Google, for their assistance with Tock OS.

Moreover, I sincerely thank my family for their constant backing and motivation throughout this journey. I am particularly thankful to my dearest friend, Sanni Lehtikunnas, whose steady encouragement kept me going.

Tampere 22nd January 2024,
Donal Johny.

# Contents

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Preface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . .

List of Programs . . . . . . . . . . . . . . . . . . . . . . . . .

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . .

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 1

2 Tock System Architecture . . . . . . . . . . . . . . . . . . . . 3

2.1 User Space . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.3 Hardware Drivers . . . . . . . . . . . . . . . . . . . . . . 5

2.4 Capsules . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.5 Other security measures . . . . . . . . . . . . . . . . . . . 6

3 Tock Project Structure . . . . . . . . . . . . . . . . . . . . . 7

3.1 Tock repository . . . . . . . . . . . . . . . . . . . . . . . 7

3.2 The Tock repository structure . . . . . . . . . . . . . . . . 7

3.3 Development Environment Setup . . . . . . . . . . . . . . . . 8

4 Setting up a custom Tock project . . . . . . . . . . . . . . . . 10

4.1 Crate Modifications . . . . . . . . . . . . . . . . . . . . . 10

4.1.1 arch crate . . . . . . . . . . . . . . . . . . . . . . . 10

4.1.2 chip crate . . . . . . . . . . . . . . . . . . . . . . . 11

4.1.3 board crate . . . . . . . . . . . . . . . . . . . . . . 12

4.1.4 Tockloader . . . . . . . . . . . . . . . . . . . . . . . 17

4.1.5 Adding a Platform to Tock Repository . . . . . . . . . . 18

4.1.6 Related work . . . . . . . . . . . . . . . . . . . . . . 18

5 Ballast Architecture . . . . . . . . . . . . . . . . . . . . . . 25

5.0.1 PULP-platform . . . . . . . . . . . . . . . . . . . . . 27

5.0.2 PULPissimo . . . . . . . . . . . . . . . . . . . . . . . 27

5.0.3 Ibex-core . . . . . . . . . . . . . . . . . . . . . . . . 29

5.0.4 System Control CPU . . . . . . . . . . . . . . . . . . 30

5.0.5 Ballast Toolchain . . . 31
6 Feasibility of porting . . . 35
6.0.1 Test setup . . . 35
7 Results . . . 40
7.0.1 Future work . . . 40
8 Conclusions . . . 44
References . . . 46

# List of Figures

2.1 Tock OS stack [31] [26] . . . . . . . . . . . . . . . . . . . . . . . . . 4

3.1 Terminal showing installation of tools on a Linux machine [19] . . . 8
3.2 udev rules for Tock OS [19] . . . . . . . . . . . . . . . . . . . . . . . 9

5.1 Ballast High-level Overview . . . . . . . . . . . . . . . . . . . . . . . 25
5.2 Granitti Board with Ballast MPSoc . . . . . . . . . . . . . . . . . . . 26
5.3 PULPisimo SoC block diagram . . . . . . . . . . . . . . . . . . . . . 28
5.4 Ibex Core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.5 System Control CPU . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.6 Comparison of PULPisimo and System Control CPU . . . . . . . . 31

6.1 Granitti Board with Ballast MPSoc setup for UART testing . . . . 35
6.2 UART transmission and reception application . . . . . . . . . . . . 36
6.3 Directory tree of Granitti board files . . . . . . . . . . . . . . . . . . 37
6.4 Diff showing the changes made for one of the Granitti board files . 38
6.5 Directory tree of Ballast chip files . . . . . . . . . . . . . . . . . . . 39
6.6 Diff showing the changes made for one of the Ballast chip files . . . 39

# List of Programs

4.1 Example of component instantiation for light and an ambient light sensor [27] . . . . . 13
4.2 Example of basic component creation [25]. . . . . 13
4.3 Example of using a component [25]. . . . . 15
6.1 snippet from the layout.ld file of Granitti board. . . . . 38
7.1 Before Refactoring . . . . . 42
7.2 After Refactoring . . . . . 42

# List of Tables

4.1 Boards with tier 1 support from Tock OS [28]. . . . . . . . . . . . . . . 20
4.2 Boards with tier 2 support from Tock OS [28]. . . . . . . . . . . . . . . 21
4.3 Boards with tier 3 support from Tock OS [28]. . . . . . . . . . . . . . . 22

# Abbreviations

| | |
|---|---|
| • ABI | Application Binary Interface |
| • APB | Advanced Peripheral Bus |
| • API | Application Programming Interface |
| • ASIC | Application Specific Integrated Circuit |
| • AXI | Advanced eXtensible Interface |
| • BSP | Bootstrap Processor |
| • FLL | Frequency-locked loop |
| • FPGA | Field Programmable Gate Array |
| • GDB | GNU Debugger |
| • HIL | Hardware Interface Layer |
| • Iot | Internet of Things |
| • IPC | Inter Process Communication |
| • IPC | Inter-Process Communication |
| • ISA | Instruction Set Architecture |
| • MCU | Micro Controller Unit |
| • MPSoC | Multiprocessor Systems-on-Chip |
| • NAPOT | Naturally aligned power-of-two region |
| • OpenOCD | Open On-chip Debugger |
| • PAC | Peripheral Access Crate. |
| • PLL | Phase-Locked Loop |
| • PMP | Physical Memory Protection |

| | |
|---|---|
| • PULP-platform | Parallel Ultra Low Power Platform |
| • RISC | Reduced Instruction Set Computer |
| • RLS | Rust Language Server |
| • SDIO | Secure Digital Input/Output |
| • SoC | System on Chip |
| • SRAM | Static Random-Access Memory |
| • SVD | System View Description |
| • TOR | Top Of Range |

# 1 Introduction

In today's world of technology, the emergence of the RISC-V Instruction Set Architecture (ISA) has ignited a boost in the development of specialised processors. This is primarily attributed to the open nature of the ISA, which removes the requirement for royalty fees, making it a highly desirable foundation for processor design. This has led to the creation of intricate SoCs featuring a variety of subsystems and processor cores responsible for a range of functions, including running main applications, managing communications, handling signal processing, ensuring security, and managing storage.

The emergence of intricate software platforms geared towards embedded applications has rendered controlling them with bare metal drivers impractical. Consequently, operating systems tailored for microcontrollers have become indispensable for empowering embedded systems, especially Internet of Things (IoT) devices [1]. However, these operating systems often need more fundamental features such as dynamic memory allocation, flexible concurrency, and fault isolation, which are increasingly essential due to these devices' intricate nature and multi-programming requirements [1].

Tock is an innovative operating system designed specifically for low-power platforms [1]. It utilises specialised hardware protection mechanisms in tandem with the advanced type-safety features of the Rust programming language, resulting in a reliable multi-programming environment for microcontrollers. Tock ensures optimal performance by providing software fault isolation, robust memory protection, and efficient management of dynamic application workloads.

This thesis delves into the intricate process of porting Tock Operating System (OS) to a custom RISCV processor, with the primary goal of gaining a deeper understanding of the complexities involved in this task. Although the porting will not be implemented in this study, the research will provide a comprehensive breakdown of the requirements for a custom Tock OS port, including hardware and software requirements. Additionally, the thesis will explore the Tock OS architecture and project structure in detail and existing works on porting Tock OS to various platforms.

The customised Ballast chip produced by the SoC hub will be examined within the scope of this investigation [37]. Tampere University and partnering corporations have teamed up to enhance research and education on SoC development, emphasising exploring its architecture and toolchain [37]. A feasibility report detailing the porting process will also be provided, outlining the approach utilised, challenges faced, and findings and recommendations from this study.

The thesis comprises eight chapters, with Chapter 2 providing an overview of the Tock system architecture and its various components, such as the kernel, hardware drivers, user space, capsules, and security features. Chapter 3 focuses on the project structure, discussing the repositories and development environment setup. Chapter 4 delves into setting up a custom tock project, including necessary crate modifications, Tockloader information, a brief examination of existing Tock ports, and an overview of development progress. Chapter 5 introduces the Ballast Architecture, detailing the various subsystems in the SoC, along with an introduction to the Ballast toolchain. Chapter 6 explores the feasibility of porting and the resulting outcomes. Finally, Chapter 7 concludes the thesis, analysing the goals and achievements and suggests potential avenues for further research.

# 2 Tock System Architecture

Tock is an embedded operating system (OS) developed as part of an academic research project. The OS is designed to enable multiple applications simultaneously on embedded platforms that use Coretex-M and RISC-V technologies. Tock has been designed to safeguard against harmful applications and device drivers, even in cases where these applications are not mutually trusting [1].

This chapter will examine the Tock system architecture and how it helps Tock protect itself from malicious code.

Tock follows two methods to safeguard various parts of the operating system. Firstly, the Tock kernel and device drivers are coded in Rust, which ensures compile-time memory safety and type safety. Rust also aids in isolating platform-specific drivers and device drivers from the kernel [19] [31]. Secondly, Tock utilizes memory protection units to isolate applications from each other and the kernel [1].

In the Tock stack depicted in Figure 2.1, the hardware device, or MCU, is highlighted in red at the bottom. This device comprises a processing unit, or CPU, and a range of peripherals that enhance its functionality, including a random number generator (RNG) and encryption capabilities (AES) [31]. Additionally, these peripherals enable the MCU to interface with other hardware components, such as GPIO, USB, I2C, ADC, SPI, and UART [1] [31].

The hardware is equipped with low-level drivers (highlighted in orange) that enable direct interaction with the hardware, forming a Hardware Interface Layer (HIL) for the kernel and capsules [31] [19]. The kernel, depicted in teal, primarily functions as an intermediary between the low-level drivers and capsules, offering the process scheduler, inter-process communication (IPC) driver, and memory management. The upper portion of the kernel, illustrated in blue, comprises capsules that are high-level drivers that interact with one another through HILs and provide an API to the user space [31]. Applications, shown in light green, operate above the kernel and use the API to request services from the kernel instead of being connected to it [31].

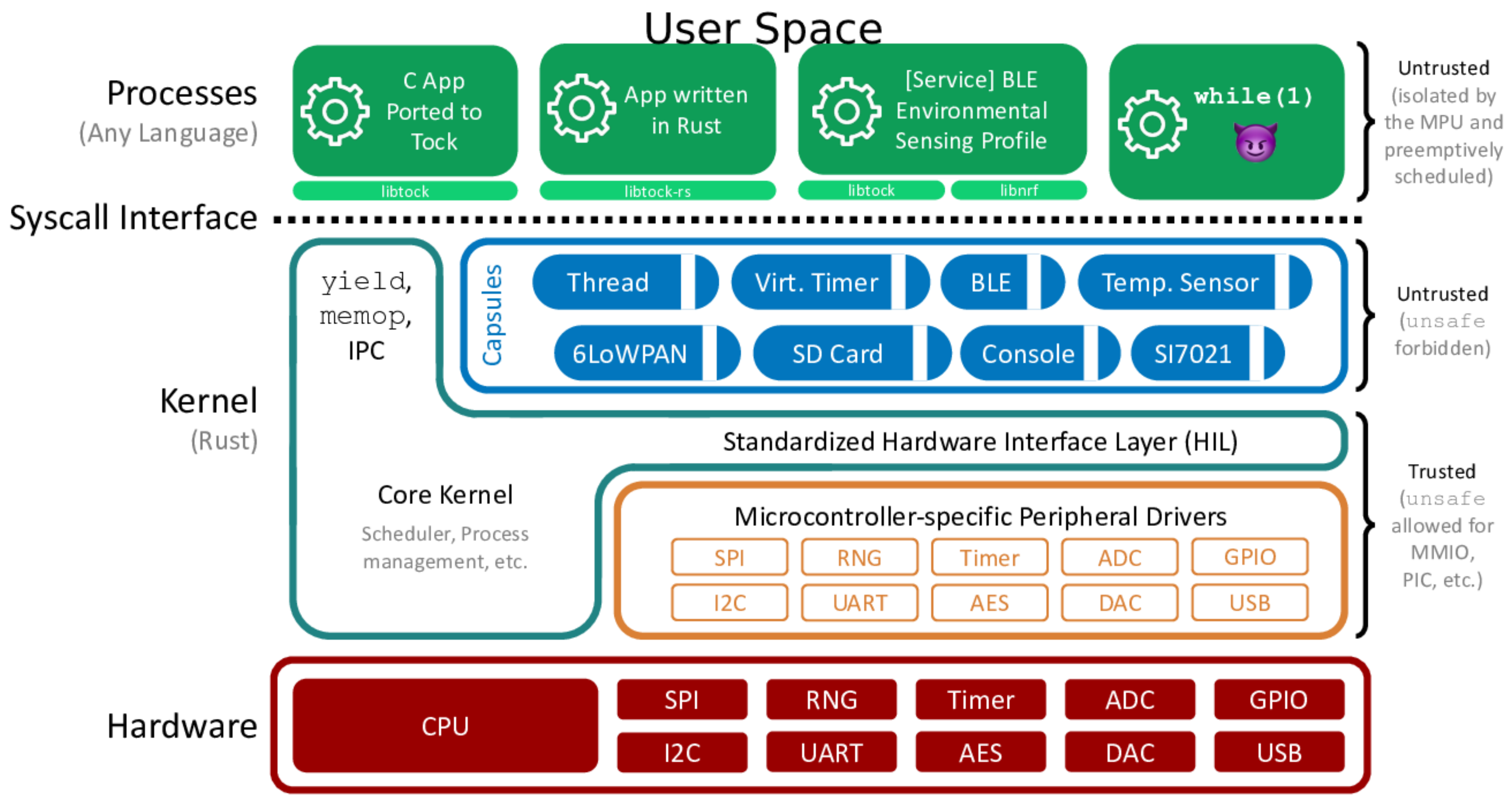


***Figure 2.1*** *Tock OS stack [31] [26]*

## 2.1 User Space

Tock operates at a high level of abstraction, providing various applications and services that can be written in any language compatible with the device's MCU architecture [19] [31]. However, these programs operate under a restricted mode, where they can only access resources authorised by the OS and cannot interact directly with hardware. In addition, application processes can only access memory allocated by the kernel and cannot access memory belonging to the kernel or other applications [19] [31]. As a result, Tock refers to these applications and services as running in user mode, comprising the user space of the operating system [19] [31].

To achieve its objectives, Tock mandates the presence of a Memory Protection Unit (MPU) in hardware. Although Tock can operate on hardware that lacks an MPU, it cannot restrict application memory access [19] [31].

## 2.2 Kernel

The Tock kernel is responsible for various critical tasks, including scheduling applications and services, allocating memory, and facilitating inter-process communication [31] [19]. It also provides an essential interface for hardware access through the Hardware Interface Layer (HIL) and grants applications and services access to driver functions via the system call interface (syscall). Additionally, the kernel carefully regulates CPU usage to ensure optimal performance and prevent unauthorized access [31] [19].

## 2.3 Hardware Drivers

In Figure 2.1, the hardware drivers are easily identifiable by their orange colour. Their primary function is establishing communication with hardware components while implementing a HIL and exposing functionality to the kernel [31] [19]. These plugins also facilitate direct communication between the hardware interface and drivers. Rust uses unsafe blocks for specific memory access to allow the kernel to interface with hardware [31] [19]. In order to reduce the need for unsafe blocks, the kernel provides a register interface that all drivers can utilise. This approach ensures that all drivers can uniformly access the hardware, utilising the same code and method [31] [19].

## 2.4 Capsules

Tock uses capsules as upper-level drivers to provide an Application Binary Interface (ABI) for applications and services in the user space [31] [19]. These capsules utilise hardware drivers and other capsules through the HIL interface to offer system calls. Tock's capsules come in two types: Syscall capsules, which provide system calls to applications and services within the userspace, and Service capsules, which provide services to other capsules, usually interfacing with a specific hardware device via a standard bus such as SPI, I2C, or serial [31] [19].

## 2.5 Other security measures

Tock takes excellent care to ensure the system remains safe from harmful code. The kernel's code is contained entirely within its repository, with no external dependencies except the Rust core library [31] [1]. Any capsules containing unsafe blocks are not compiled, guaranteeing they are memory-safe and can only access hardware via a HIL. Furthermore, applications and services operate in user mode, preventing direct access to hardware and limiting their memory access to their processes, which ensures the system stays secure and stable [31] [1].

# 3 Tock Project Structure

The following section provides a detailed overview of the Tock project structure and outlines the necessary steps for creating an ideal environment for Tock OS development.

## 3.1 Tock repository

The official tock repository can be accessed via the following link: https://github.com/tock The following three are relevant to this research out of all the existing repositories [26].

- tock - this repository contains all the source code for the Tock operating system
- libtock-c - this repository contains libraries for running C applications on top of Tock and some examples.
- libtock-rs - this repository contains libraries for running Rust applications on top of Tock and some examples.

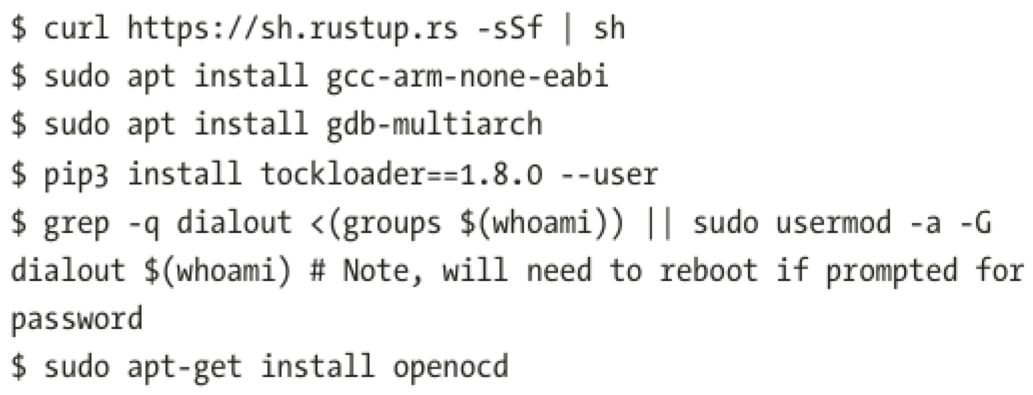
## 3.2 The Tock repository structure

The Tock repository holds the source code for the Tock kernel, which is structured in the following manner [26].

- arch -This directory contains customised content for various architectures, providing detailed information about their structures and functions. Currently, Tock offers support for both Cortex and RISC-V architectures [26] [19].
- boards - Contained within this directory is a collection of code designed to cater to each device presently integrated into Tock [26] [19].
- capsules - The capsules in the Tock kernel are analogous to drivers in other operating systems. This specific directory contains capsules that facilitate either peripheral support or communication channels [26] [19].

- chips - The code presented here has been specifically tailored to support Tock's MCU implementations. This folder has been designed to work with the arch folder and includes functions unique to each SoC [26] [19].
- doc - This resource furnishes an all-encompassing set of guidelines on proficiently employing Tock and its intricate framework.
- kernel -Contained within this directory is the comprehensive Tock kernel implementation, encompassing various essential components such as the system call, Inter-Process Communication (IPC), memory management, scheduler, and Hardware Interface Level (HIL) definition files.
- ibraries - contain libraries used by all the source codes in this repository. [26] [19].
- tools - These scripts create builds, execute tests, and carry out similar operations.

## 3.3 Development Environment Setup



```
$ curl https://sh.rustup.rs -sSf | sh
$ sudo apt install gcc-arm-none-eabi
$ sudo apt install gdb-multiarch
$ pip3 install tockloader==1.8.0 --user
$ grep -q dialout <(groups $(whoami)) || sudo usermod -a -G
dialout $(whoami) # Note, will need to reboot if prompted for
password
$ sudo apt-get install openocd
```

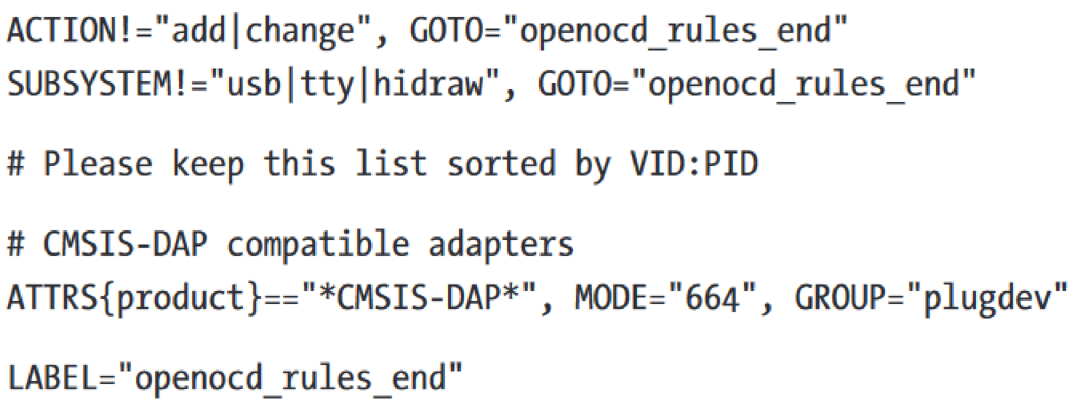
***Figure 3.1*** *Terminal showing installation of tools on a Linux machine [19]*

The following tools and libraries must be installed to develop and deploy the Tock kernel applications in development mode [26] [19].

- tock - the Tock kernel source code Github repository;
- libtock-rs the Rust application library repository;

- rustup - an installer for the Rust programming language;
- Rust - the compiler and tools for Rust applications;
- OpenOCD - a tool to program and debug the devices;
- Tockloader - a tool to manage the installation of the Tock kernel and Tock applications on the devices;
- GCC for arm/RISC-V - relevant compilers for the devices;
- Any additional device-specific tools.

Once the tools are installed. The next step is to setup the udev rules for the hardware. This will allow OpenOCD to interact with the USB connection. To achieve this, create a new file in the udev directory located in **etc/udev/rules.d** and add the following to the file.



```
ACTION!="add|change", GOTO="openocd_rules_end"
SUBSYSTEM!="usb|tty|hidraw", GOTO="openocd_rules_end"

# Please keep this list sorted by VID:PID

# CMSIS-DAP compatible adapters
ATTRS{product}=="*CMSIS-DAP*", MODE="664", GROUP="plugdev"

LABEL="openocd_rules_end"
```

***Figure* 3.2** *udev rules for Tock OS [19]*

Once the files has been saved, to load the new configuration, simply restart udev system with the following command **sudo udevadm control –reload-rules** [19] [26]

# 4 Setting up a custom Tock project

A new ”board” crate must be created to incorporate a new platform onto Tock. This may entail adding ”chip” and ”arch” crates [25]. In Rust, a crate refers to the minimum amount of code the Rust compiler processes simultaneously [32]. Crates can encompass modules, which could be defined in other files compiled with the crate. There are two types of crates: binary crates and library crates. Binary crates are executable programs, such as command-line tools or servers, that can be compiled and run [32].

The board crate specifies the hardware resources available on the platform and connects capsules with chip crates [25]. The chip crate, in turn, implements peripheral drivers for a specific microcontroller, utilising traits from kernel/src/hil [25]. A rust trait defines the functionality a particular type has and can be shared with other types. Traits are similar to a feature called interfaces in other languages, although there are some differences in rust [33].

The existing chip crate may be utilised if the platform already utilises a Tock-supported microcontroller. On the other hand, the arch crate implements low-level code for a specific hardware architecture, such as the initial boot process and system call implementation [25] [19].

## 4.1 Crate Modifications

This section includes more details on what is required to implement each type of crate for a new hardware platform.

### 4.1.1 arch crate

Tock is compatible with various architectures, including the ARM Cortex-M0, Cortex-M3, and Cortex-M4, as well as the RISC-V 32IMAC [25]. Its codebase is architecture-agnostic mainly, except for a few components, such as syscall entry/exit, interrupt configuration, top-half interrupt handlers, MPU configuration (if applicable), and power management configuration (if applicable). This approach maximises portability and minimises the need for platform-specific code. If Tock is being ported to a new architecture, relevant architecture code should be added to this crate [25].

### 4.1.2 chip crate

When developing a crate for a microcontroller, it is vital to strike a balance between specificity for a particular microcontroller family and generality to support multiple microcontrollers within that family[25]. For instance, the **chips/nrf52** and **chips/nrf5x** crates share support for the nRF58240 and nRF58230 microcontrollers in order to streamline the process of adding new microcontrollers and avoid duplicative code [25].

The chip crate incorporates microcontroller-specific implementations of the interfaces defined in **kernel/src/hil**. Given the many features of chips, Tock offers a variety of interfaces to express them. The process of implementing a new chip begins with getting the reset and initialisation code up and running and configuring it to run on the chip's default clock [25]. Next, a GPIO interface should be added, and a minimal board incorporating the chip should be created. Validation should be performed using an end-to-end userland application that utilises GPIOs. This approach enables incremental progress and provides an opportune moment to submit a pull request to Tock, showcasing one's efforts and drawing further support [25].

As one progresses, breaking down the chip into manageable work units is advisable. For instance, implementing **kernel::hil::UART** for the chip and submitting a pull request would be prudent. It is worth noting that historically, Tock chips defined peripherals as static mut global variables, which led to the use of unsafe code and prevented boards from instantiating only necessary peripherals. Now, peripherals are instantiated at runtime in **main.rs**, addressing such issues [25]. To facilitate the use of the chip by other boards without adding unnecessary overhead and code size, the chip should provide a **ChipNameDefaultPeripherals** struct that defines and creates all available peripherals for the chip in Tock.

### 4.1.3 board crate

The board crate, which is located in the boards/src directory, is specifically designed to cater to the needs of a particular hardware platform [25]. By configuring the kernel, the board file facilitates the hardware setup to be supported. This involves many tasks, such as creating sensor drivers, mapping communication buses to those sensors, and configuring GPIO pins.

Tock is currently utilising ”components” to establish board crates [25]. These components are structured containers containing all the necessary setup code for a specific driver and only require boards to provide unique options specific to that platform.

```
let isl29035 = components::isl29035::Isl29035Component::new(
    sensors_i2c, mux_alarm)
     .finalize(components::isl29035_component_static!(sam4l::ast::Ast)
    );

let ambient_light = components::isl29035::AmbientLightComponent::new(
     board_kernel,
     capsules::ambient_light::DRIVER_NUM,
     isl29035,
)
.finalize(components::ambient_light_component_static!());
```

***Program 4.1*** *Example of component instantiation for light and an ambient light sensor [27]*

Although board initiation should primarily be done using components, not all components have been developed yet, so board files typically consist of a combination of components and verbose driver instantiation. To begin with, it is advisable to use an existing board's main.rs file and make modifications as necessary. Initially, it is recommended to delete most capsules and gradually add them back as progress is made [25].

### Component creation

Developing a component is highly recommended to enhance the setup process for capsules. This approach offers two key advantages: firstly, any intricacies involved in configuring the capsule can be isolated within the component, thereby reducing the likelihood of errors during capsule usage. Secondly, the specifics of creating a capsule are abstracted from the overall setup of a board. As such, Tock advocates for boards to leverage components for their primary startup process [25].

Basic components generally have a structure like the following simplified example for a **console** component [25].

```
use core::mem::MaybeUninit;

/// Helper macro that calls `static_buf!()`. This helps allow
    components to be
/// instantiated multiple times.
```

```rust
#[macro_export]
macro_rules! console_component_static {
    () => {{
        let console = kernel::static_buf!(capsules::console::Console<'static>);
        console
    }};
}

/// Main struct that represents the component. This should contain all
/// configuration and resources needed to instantiate this capsule.
pub struct ConsoleComponent {
    uart: &'static capsules::virtual_uart::UartDevice<'static>,
}

impl ConsoleComponent {
    /// The constructor for the component where the resources and configuration
    /// are provided.
    pub fn new(
        uart: &'static capsules::virtual_uart::UartDevice,
    ) -> ConsoleComponent {
        ConsoleComponent {
            uart,
        }
    }
}

impl Component for ConsoleComponent {
    /// The statically defined (using `static_buf!()`) structures where the
    /// instantiated capsules will actually be stored.
    type StaticInput = &'static mut MaybeUninit<capsules::console::Console<'static>>;
    /// What will be returned to the user of the component.
    type Output = &'static capsules::console::Console<'static>;

    /// Initializes and configures the capsule.
    unsafe fn finalize(self, s: Self::StaticInput) -> Self::Output {
        /// Call `.write()` on the static buffer to set its contents
```

```
   with the
        /// constructor from the capsule.
        let console = s.write(console::Console::new(self.uart));

        /// Set any needed clients or other configuration steps.
        hil::uart::Transmit::set_transmit_client(self.uart, console);
        hil::uart::Receive::set_receive_client(self.uart, console);

        /// Return the static reference to the newly created capsule
   object.
        console
    }
}
```

***Program 4.2*** *Example of basic component creation [25].*

```
// in main.rs:

let console = ConsoleComponent::new(uart_device)
    .finalize(components::console_component_static!());
```

***Program 4.3*** *Example of using a component [25].*

When creating components, it is imperative to adhere to the following guidelines [25]:

- All static buffers required for the component must be generated exclusively using the **static_buf!()** macro and not elsewhere. This guarantees that components can be utilised multiple times [25].
- The macro that encapsulates **static_buf!()** should follow the naming convention of [**capsule name**]**_component_static!()**. The macro should solely produce static buffers.
- All resources and configurations not linked to static buffers should be passed to the **new()** constructor of the component object.

In certain instances, capsules and resources may be templated over chip-specific resources, making static buffer definition more intricate. The same macro strategy must be employed for other static buffers to ensure that components are reusable across diverse boards and microcontrollers.

### Board Support

Boards necessitate supplemental support files in addition to kernel code. These files contain vital metadata regarding the board's nomenclature, instructions on how to load code onto it, and any unique requirements that may be essential for userland applications to function on the board [25].

Each board must devise a customised routine to manage panic incidents effectively. While the Tock kernel handles most panic machinery, the board author must ensure that the hardware interfaces, namely the LEDs or UART, are provided with primary access [25]. Each board must author a custom routine to handle **panic!s.**

To start with, the LED-based panic! feature is the easiest to set up. The panic! handler should configure a noticeable LED and then to activate it use the following. **kernel::debug::panic_blink_forever** [25].

If the system has UART, it can access a wealth of helpful debugging information from the kernel; however, we are dealing with a panic! scenario, it is essential to keep the implementation minimalistic. The UART supplied must be synchronous (unlike the rest of the kernel UART interfaces, which are asynchronous). The simplest way is to create a basic Writer that writes one byte at a time directly to the UART. The efficiency of the panic! UART writer is not a priority [25]. Afterwards, it can replace the call to **kernel::debug::panic_blink_forever** with **kernel::debug::panic**. Remember to keep it simple and efficient for the best results [25].

### Board Cargo.toml, build.rs

To ensure a seamless process, generating a high-level manifest, Cargo.toml, is crucial for every board crate. This can be done by duplicating an existing board's manifest and modifying the board name and author(s) as necessary. Moreover, Tock includes a build script, build.rs, that should be included. This build script helps to include a dependency on the kernel layout, making the process more straightforward [25].

#### Board Makefile

There is a Makefile in the root of every board crate [25]. At a minimum, the board Makefile must include:

When building the kernel, it is essential to include rules in the board Makefile to get code onto the board [25]. This process will vary depending on the specific board used, but Tock provides two commonly used targets:

- ”program”: This is used to load code onto boards with a bootloader or other support IC [25]. During regular operation, users can plug in the board and type ”make program” to load the code.
- ”flash”: This is used for more direct loading, typically done through a JTAG or similar interface [25]. Often, this requires external hardware, although some development kit boards have an integrated JTAG on-board, making external hardware unnecessary.
- ”install”: This should be an alias for either ”program” or ”flash”, depending on the preferred approach for the specific board [25].

#### Board README

For each board, it is necessary to include a README.md file at the top level of the crate [25]. This file should contain the following:

- Links to information on the platform and how to acquire it. If multiple versions of the platform exist, specify the version used for testing.
- An overview of how to program the hardware and any additional dependencies required [25].

### 4.1.4 Tockloader

Tockloader is a Python-based application utilised to program Tock onto various hardware platforms. Upon successful installation, this tool provides a plethora of commands for efficient management of applications on a board [29]. Among the key commands available are:

- Tockloader install - Load Tock applications onto the board [29].

- Tockloader update - Update an application that is already on the board with a new binary [29].
- Tockloader uninstall application name - Remove the specified application from the board [29].

Tockloader conveniently includes a list of hardcoded parameters for a diverse range of boards, which can be accessed through the "**list-known-boards**" command. However, if the desired platform is not present, one can simply specify the necessary parameters through command line options and provide a board name. Alternatively, it is recommended to file an issue on the Tockloader repository so that the development team can update the tool to include pertinent information about the selected platform [29].

### 4.1.5 Adding a Platform to Tock Repository

When merging a board into the main Tock repository, specific guidelines must be followed [25].

- The board's hardware should be easily accessible and available for purchase online.
- The platform should have console support for debug!() and printf(), timer support, and GPIO support with interrupt functionality.
- The contributor should also be prepared to maintain the platform and assist with testing for future releases.

### 4.1.6 Related work

As of October 2023, Tock OS now officially supports several architectures including Cortex-M, Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, RISC-V, and RV32I. Additionally, Tock provides support for various hardware platforms, which can be found in the boards directory of the Tock repository [28].

To better organize the boards, Tock has divided them into three tiers based on their level of support.

- Tier 1 consists of the most feature-complete and well-tested boards, which are used regularly by core team members and highly engaged contributors. These boards are showcased in the Tock Book [28].
- Tier 2 includes platforms with reasonably regular use, but may have incomplete peripheral support or known issues documented in the release notes. Some Tier 2 boards are variations of Tier 1 boards and are in good shape but not heavily tested [28].
- Tier 3 boards are new or highly experimental and only promise to support the minimum platform requirements listed in the Porting documentation [28].

**RISC-V**

Tock exhibits robust backing towards the RISC-V architecture, albeit lacking in tier 1 or 2 support for any individual RISC-V board. For those seeking to employ Tock on RISC-V, several alternatives are available.

- ESP32-C3-DevKitM-1. This board is under active development to move to Tier 2 support [28].
- For an entirely virtual platform on QEMU, the QEMU RISC-V 32-bit virt platform board can be used. This can be quickly started and run on a host computer [28].
- For a simulation environment one can use Verilator with OpenTitan Earlgrey on CW310 or Verilated LiteX Simulation [28].
- For an FPGA setup one can use OpenTitan Earlgrey on CW310 or LiteX on Digilent Arty A-7 [28].

***Table 4.1*** *Boards with tier 1 support from Tock OS [28].*

| **Board** | **Architecture** | **MCU** | **Interface** | **App deployment** | **QEMU support** |
|---|---|---|---|---|---|
| Hail | ARM Cortex-M4 | SAM4LC8BA | Bootloader | tockloader | No |
| Imix | ARM Cortex-M4 | SAM4LC8CA | Bootloader | tockloader | No |
| Nordic nRF52840-DK | ARM Cortex-M4 | nRF52840 | jLink | tockloader | No |
| Nano 33 BLE | ARM Cortex-M4 | nRF52840 | Bootloader | tockloader | No |
| BBC Micro:bit v2 | ARM Cortex-M4 | nRF52833 | openocd | tockloader | No |
| Clue nRF52840 | ARM Cortex-M4 | nRF52840 | Bootloader | tockloader | No |

***Table 4.2*** *Boards with tier 2 support from Tock OS [28].*

| **Board** | **Architecture** | **MCU** | **Interface** | **App deployment** | **QEMU** |
|---|---|---|---|---|---|
| Nordic nRF52-DK | ARM Cortex-M4 | nRF52832 | jLink | tockloader | No |
| Nordic nRF52840-Dongle | ARM Cortex-M4 | nRF52840 | jLink | tockloader | No |
| Particle Boron | ARM Cortex-M4 | nRF52840 | jLink | tockloader | No |
| ACD52832 | ARM Cortex-M4 | nRF52840 | jLink | tockloader | No |
| ST Nucleo F446RE | ARM Cortex-M4 | STM32F446 | openocd | custom | WIP |
| ST Nucleo F429ZI | ARM Cortex-M4 | STM32F429 | openocd | custom | WIP |
| STM32F3Discovery kit | ARM Cortex-M4 | STM32F303VCT6 | openocd | custom | WIP |
| STM32F412G Discovery kit | ARM Cortex-M4 | STM32F412G | openocd | custom | WIP |
| Pico Explorer Base | ARM Cortex-M0+ | RP2040 | openocd | openocd | No |
| Nano RP2040 Connect | ARM Cortex-M0+ | RP2040 | custom | custom | No |
| Raspberry Pi Pico | ARM Cortex-M0+ | RP2040 | openocd | openocd | No |
| SparkFun RedBoard Artemis Nano | ARM Cortex-M4 | Apollo3 | custom | custom | No |
| SparkFun - expLoRaBLE | ARM Cortex-M4 | Apollo3 | custom | custom | No |

***Table 4.3*** *Boards with tier 3 support from Tock OS [28].*

| **Board** | **Architecture** | **MCU** | **Interface** | **App deployment** | **QEMU support** |
|---|---|---|---|---|---|
| WeAct F401CCU6 Core Board | ARM Cortex-M4 | STM32F401CCU6 | openocd | custom | No |
| SparkFun RedBoard Red-V | RISC-V | FE310-G002 | openocd | tockloader | Yes (5.1) |
| SiFive HiFive1 Rev B | RISC-V | FE310-G002 | openocd | tockloader | Yes (5.1) |
| ESP32-C3-DevKitM-1 | RISC-V-ish RV32I | ESP32-C3 | custom | custom | No |
| BBC HiFive Inventor | RISC-V | FE310-G003 | tockloader | tockloader | No |
| i.MX RT 1052 Evaluation Kit | ARM Cortex-M7 | i.MX RT 1052 | custom | custom | No |
| Teensy 4.0 | ARM Cortex-M7 | i.MX RT 1062 | custom | custom | No |
| Digilent Arty A-7 100T | RISC-V RV32IMAC | SiFive E21 | openocd | tockloader | No |

### Tock Development status

As of November 2023, Tock has advanced to **version 2.1.1**, boasting over 100 contributors working on the project tirelessly. Tock introduces fresh features, stability updates, and bug fixes with each new version release to elevate its performance.

**Tock 1.0** was deliberately engineered to prioritise low power and binary stability, allowing applications compiled for the 1.0 system call interface to operate on any kernel claiming a 1.0 system call interface. It included support for various system call interfaces, including timers, debug consoles, LEDs, buttons, GPIO, and high-level sensor drivers like accelerometer, magnetometer, light, temperature, pressure, and more [20].

Additionally, **Tock 1.0** addressed issues, such as bug fixes that remedied pin misassignment on the new imix board [20] and power scaling issues in sam4l, and addressed bugs with the SAM4AL SPI slave implementation [20].

The landmark release of Tock, **version 1.3**, had two pivotal goals. Firstly, it aimed to make Tock architecture agnostic, and secondly, it sought to incorporate an interface that enabled the sending and receiving of UDP packets [21]. To achieve the former objective, several changes were implemented, including the elimination of support/arm.rs from the kernel crate [14], the removal of architecture-specific stack/register handling from processes [15], the relocation of architecture-dependent syscall code to arch/cortex-m [15], the eradication of Arm register specifics from panic print [16], and the addition of support for a universal MPU interface [17].

The following software release, **Version 1.5**, boasts many new updates to improve user experience. These include better process handling, RISC-V development, support for new boards, and updated components [22].

One of the most notable features in this update is the addition of generic components, which allows capsule authors to create components for new sensors. This makes it much simpler and more trustworthy to chain callbacks together on any board with that sensor [23].

Another essential addition to **Version 1.5** is support for RISC-V Physical Memory Protection (PMP). The MPU implementation on RISC-V allows for allocating sections of memory for apps, enabling userspace apps to run. Tock uses the Physical Memory Protection (PMP) Top of Range (TOR) alignment to avoid alignment issues with Naturally aligned power-of-two region (NAPOT). This also results in less wasted memory, as using NAPOT requires the address to be aligned to the size [22].

Moreover, **Version 1.5** includes capsules for low-level debugging, log storage, and the l3gd20 3-axis gyro and temperature sensor [22].

The **Tock 2.0** system call interface has undergone significant changes resulting in a more defined resource sharing mechanism between processes and the kernel. The updated system calls now allow up to four registers of values to be returned to userspace. Capsules are now subject to additional restrictions in implementing subscribe and allow, which are now checked within the kernel. [24].

Regarding app requirements, **Tock 2.0** mandates that each app include a Tock Binary File header specifying the minimum kernel version needed. The process loader automatically checks this header and rejects any apps not explicitly compatible with **Tock 2.0**. Furthermore, all capsules must store the process state in a grant; each capsule can only have up to one grant.

The update also involved significant changes to the chip and platform traits in the kernel. The chip now only includes functions closely tied to microcontrollers, while the platform has been divided into distinct and well-defined traits. The kernel crates have been reorganized, and kernel exports are now structured more clearly.

Finally, **Tock 2.0** supports new platforms like the Nano Connect, BBC Micro:bit v2, Teensy 4.0, and Raspberry Pi Pico, among others.

Ongoing efforts are being made towards the unveiling of **Tock 3.0**, with no definitive date of release disclosed as of November 2023.

# 5 Ballast Architecture

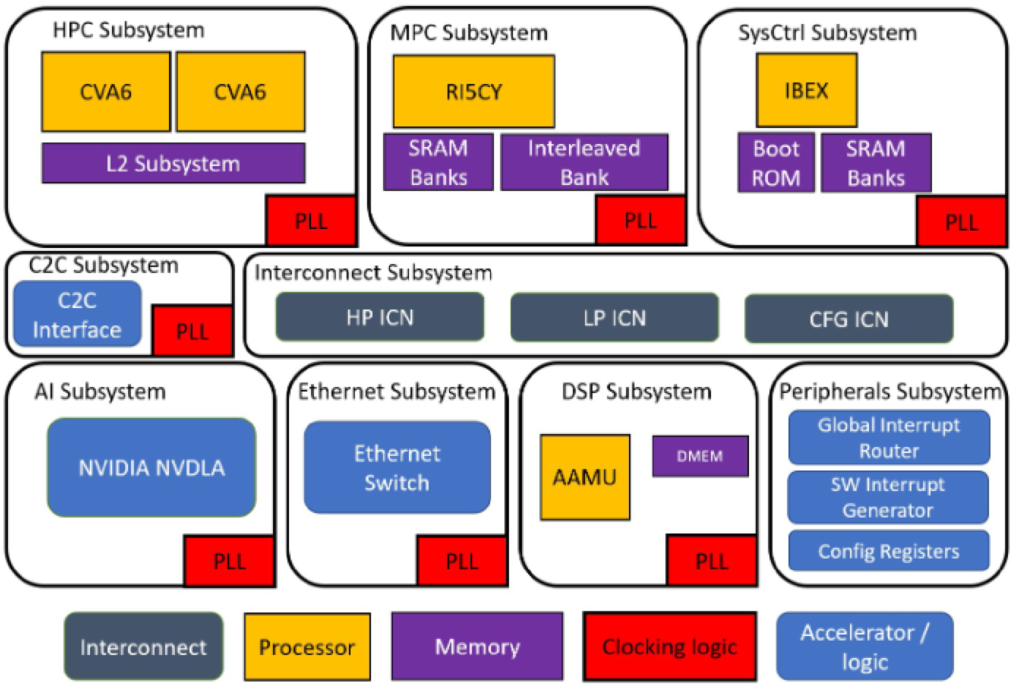


***Figure 5.1*** *Ballast High-level Overview*

Figure 5.1 shows a high level block diagram of the Ballast Architecture [9]. The system is comprised of three processor subsystems based on RISC-V architecture. The first subsystem, high-performance computing (HPC), is a multi-core subsystem that can run Linux and comes with two CVA6 64-bit processor cores [9] [30]. The second subsystem, medium performance computing (MPC), is a single-core subsystem that utilises the PULP-platform's PULPissimo microcontroller. Lastly, the System Control CPU (SysCtrlCPU) subsystem is also based on the PULPissimo microcontroller but uses the smaller Ibex-core, while the MPC uses the RI5CY processor core. The SysCtrlCPU subsystem functions as the Bootstrap Processor (BSP) of the system, while the other subsystems function as application processors [9] [30].

The other subsystems on the chip support the operation of these three subsystems. The Digital Signal Processor (DSP) subsystem is a custom-designed co-processor for the RISC-V-based processors on the SoC. The Aamu-core of the subsystem is a Transport Triggered Architecture (TTA) processor [11], an alternative to the more common operation-based processor architecture used in CPUs. The Aamu-core was generated using the TTA-based co-design environment (TCE) toolchain.

The Ethernet subsystem allows for ethernet connectivity, while the chip-to-chip (C2C) subsystem provides an interface for communicating with an external chip. The AI subsystem acts as an accelerator for AI applications and is built on the open-source NVIDIA deep learning accelerator (NVDLA) [10].

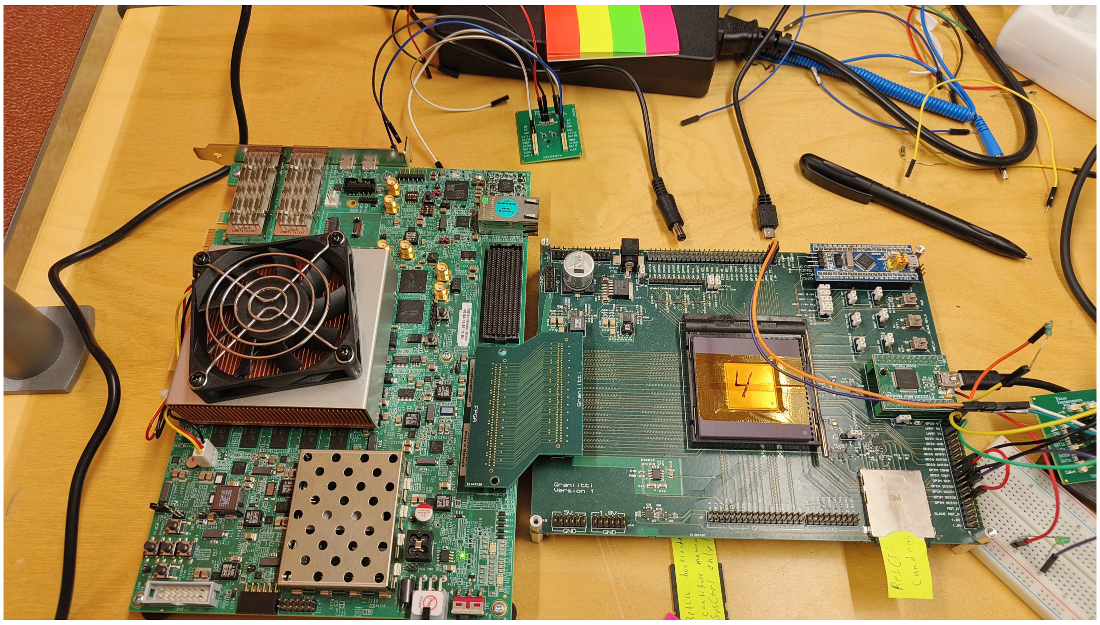


***Figure* 5.2** *Granitti Board with Ballast MPSoc*

Figure 5.2 shows the Ballast MPSoC Connected to the Granitti board. The board was custom designed by the SoC Hub team to debug the different subsystems in Ballast.

The top peripheral subsystem grants shared peripheral access to all cores and continuous memory space. The Interconnect subsystem furnishes connectivity within the chip with three distinct interconnects for high-performance (HP ICN), low-performance (LP ICN) and configuration (CFG ICN) applications. The Ballast ASIC subsystems utilise phase-locked loop (PLL) circuits to produce the required clock frequencies on the chip. PLLs are frequency synthesisers that generate frequencies from one or multiple source frequencies. They are a favourable solution for ASIC systems because they can be made as part of the IC, reducing the amount of external noise the signals can encounter and providing GHz-level frequencies.

### 5.0.1 PULP-platform

The PULP project is a joint effort between the University of Bologna's Department of Electrical, Electronic and Information Engineering and ETH Zürich's Integrated Systems Laboratory. It began in 2013 with the intention of developing computing solutions that achieve optimal performance while consuming minimal energy [9] [30].

The PULP platform is an open-source system constructed on the RISC-V instruction set architecture. It aims to enhance energy efficiency and performance by integrating data- and thread-level parallelism with near-threshold voltage computing. This technique operates transistors near their threshold voltage level, which helps to decrease power consumption.

The platform offers designs for both small IP blocks, like RISC-V CPUs, peripherals, interconnects, and hardware accelerators, as well as complete single or multi-core systems.

### 5.0.2 PULPissimo

The PULPissimo is a microcontroller platform that uses 32-bit RISC-V technology. It can be designed to use either the RI5CY-core, a 4-stage pipelined CPU with an optional floating point unit, or the Ibex-core, a 2-stage CPU optimized for controlling tasks. The customization option is used on the Ballast SoC where the MPC subsystem uses the RI5CY-core, and the SysCtrlCpu subsystem uses the Ibex-core.

The PULPissimo (Fig  5.3) design comprises five functional subsystems: the fabric controller (FC), peripherals, I/O DMA ( DMA), interconnect, and memory. The architecture is divided into three modules: the Padframe, the Safe Do-

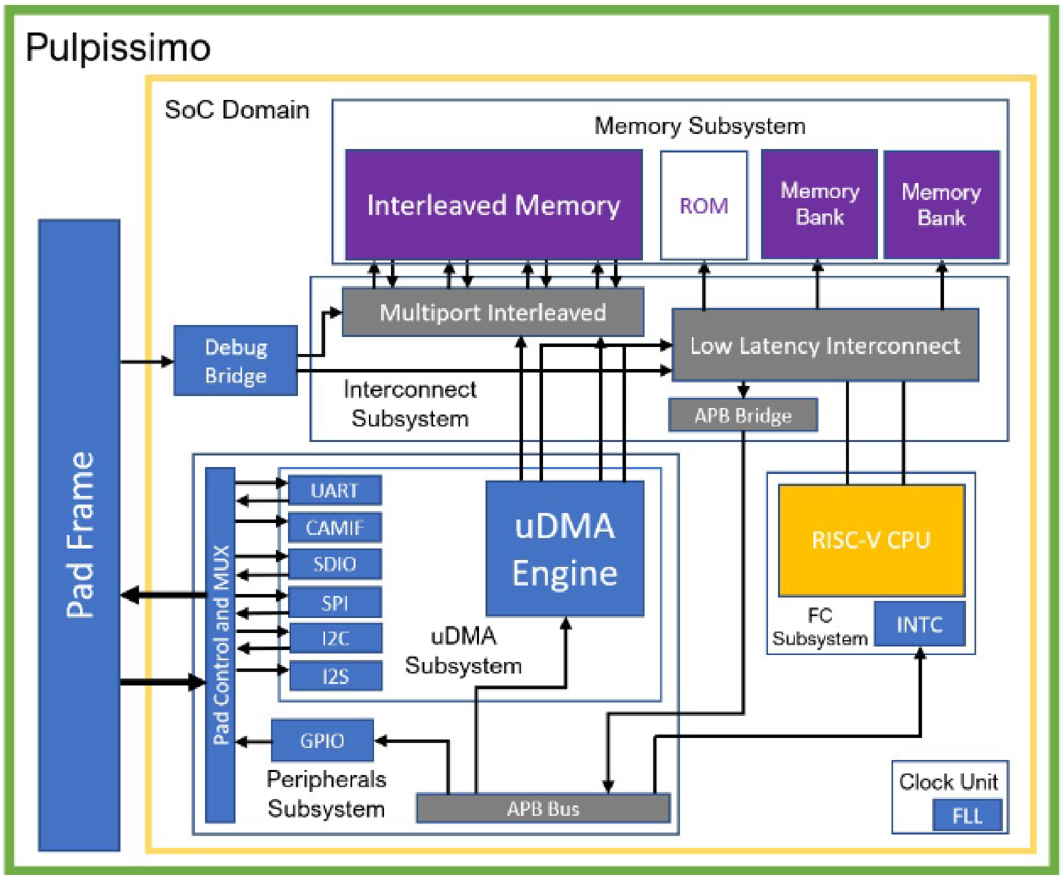


***Figure 5.3*** *PULPisimo SoC block diagram*
[9] [30]

main, and the SoC Domain. This segmentation allows for creating individually controllable power domains in an ASIC implementation. The Padframe contains technology-independent wrappers for the SoC's I/O pads, while the Safe Domain contains modules that must remain powered on, regardless of the SoC's state. Lastly, the SoC Domain includes all five functional subsystems.

The FC subsystem houses the RISC-V CPU and an interrupt controller connected to the advanced peripheral bus (APB) interface. Other hardware accelerators that the PULPissimo may have are also located in this subsystem to ensure consistent power domain control. The memory subsystem contains all internal memory for the system on a chip (SoC). The interconnect subsystem connects the memory, external debug bridge, FC, peripheral, and the DMA subsystem.

The DMA subsystem is an autonomous direct memory access (DMA) subsystem within the peripherals subsystem designed to manage data transfers between peripherals and memory, reducing the workload of the CPU. The SoC supports multiple peripheral interfaces, including SPI, UART, and JTAG, which are multiplexed into a smaller number of available physical ports using dedicated control registers.

### 5.0.3 Ibex-core

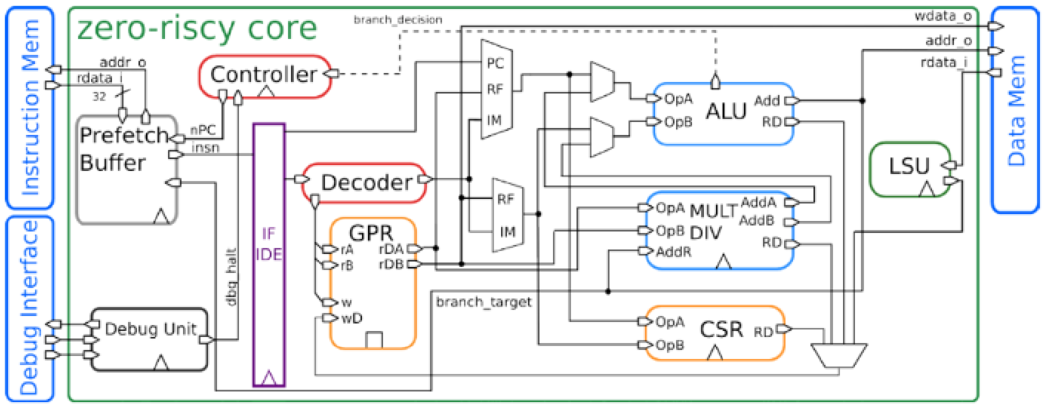
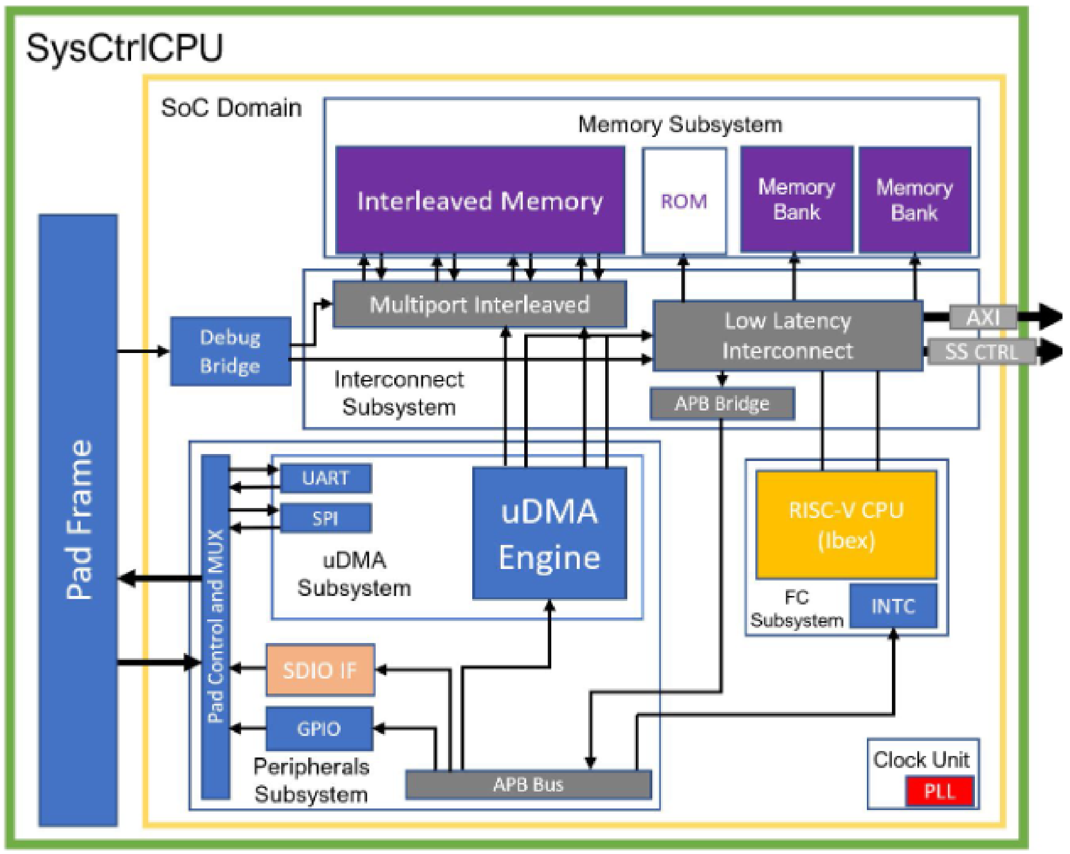


***Figure 5.4** Ibex Core*
[9] [30]

The Ibex is a processor core (fig 5.4) optimised for embedded devices that supports a base integer instruction set, reduced register extension (E), and M and C extensions. It can implement either 32 or 16 32-bit general purpose registers (GPRs) depending on the use of the E extension. The IF/IDE register bank separates the two pipeline stages between the fetch and decode unit. Due to its Harvard architecture, the Ibex implements some control and status registers (CSRs) defined in the RISC-V specification and has separate instruction and data memory interfaces.

The Ballast SoC is controlled and booted by the System Control CPU ( 5.5). The Ballast development team created the SysCtrlCPU by customising the PULPissimo SoC, removing unnecessary peripherals and using the Ibex-core instead of the default RI5CY-core. The removed interfaces include I2C, I2S, and the camera interface. Instead, an AXI interface, subsystem control signals, and a custom SDIO interface were added. These interfaces provide high-bandwidth connectivity within the Ballast ASIC, reset, clock, and PLL signals for other subsystems on the ASIC, and an external memory space accessible for the CPU immediately as instruction execution starts, respectively. The custom SDIO interface includes a hardware state machine to perform an initialisation routine of an external SD card.

### 5.0.4 System Control CPU



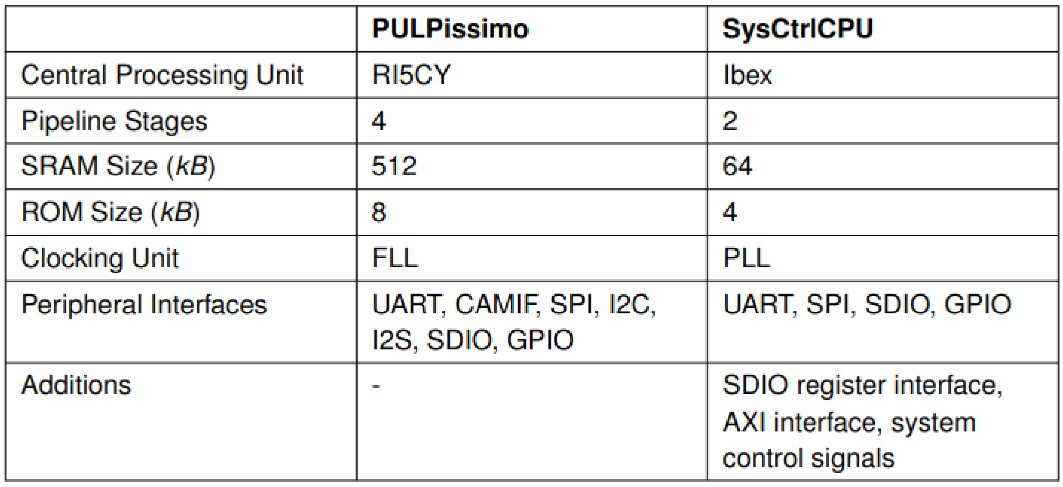
[9]

[30]

***Figure 5.5*** *System Control CPU*

Multiple memory-mapped registers have been introduced to enhance control over the Ballast SoC. These include 32-bit registers for PLL control, status functions for nine ASIC subsystems, and three registers with 8-bit clock controls. Additionally, two 32-bit registers were added for subsystem reset and clock enables, aiding in power analysis. Lastly, a 16-bit clock divider control register has been incorporated for top peripherals. Size reductions were made to most memory on the SysCtrlCPU from the PULPissimo.

The L2 static random-access memory (SRAM) on the PULPissimo has been reduced from 512 KB to 64 KB, and the size of the boot ROM has been reduced from 8 KB to 4 KB. Additionally, the system's memory map has been scaled down, and the regions for the frequency-locked loop (FLL) and optional hardware accelerator have been removed.



| | **PULPissimo** | **SysCtrlCPU** |
|---|---|---|
| Central Processing Unit | RI5CY | Ibex |
| Pipeline Stages | 4 | 2 |
| SRAM Size (*kB*) | 512 | 64 |
| ROM Size (*kB*) | 8 | 4 |
| Clocking Unit | FLL | PLL |
| Peripheral Interfaces | UART, CAMIF, SPI, I2C, I2S, SDIO, GPIO | UART, SPI, SDIO, GPIO |
| Additions | - | SDIO register interface, AXI interface, system control signals |

***Figure 5.6*** *Comparison of PULPisimo and System Control CPU* [9] [30]

### 5.0.5 Ballast Toolchain

In order to explore the feasibility of porting Tock OS on Ballast, a Rust-based toolchain is required, along with additional tools to generate necessary code. Rust has plenty of tools to help with software development, ranging from build management tools to linter and code formatting. It also has embedded specific tools, such as svd2rust, for creating peripheral access crates.

### Cargo

The Cargo package manager for Rust facilitates the declaration of dependencies and ensures a consistent build for Rust packages. This is achieved by utilising two metadata files containing package information, the acquisition and construction of the package's dependencies, and the invocation of Rustc or a suitable build tool with the appropriate parameters to complete the package build. Additionally, Cargo introduces a set of conventions that streamline working with Rust packages [34].

Cargo normalises the required commands for program and library builds, resulting in a simplified building process. Instead of invoking the rust compiler directly, a user can use a general command, such as ”cargo build,” to allow Cargo to handle the construction of the correct rustc call [34]. Cargo also automatically retrieves any dependencies defined in the project from a registry and prepares them for incorporation into the build as needed [34].

### Rust-clippy

Clippy is a helpful tool that assists users in writing more idiomatic Rust code by catching common mistakes [5]. It offers various lint categories, each providing solutions on how to fix the issues they detect. Some of the lint categories include:

- **Clippy:correctness** which, when triggered, aborts compilation since the code is either wrong or useless and requires fixing [6].
- **Clippy:suspicious** which, by default, warns the user about potentially suspicious code that should be fixed unless intentionally written that way [6].
- **Clippy:complexity** which offers suggestions on simplifying code, focusing on making it more readable while preserving semantics [6].
- **Clippy:perf**, which suggests increasing performance by writing code in a way that is easier for the optimiser to handle. This warning applies to code that the compiler cannot trivially optimise [6].

### Rust-analyzer

Rust-analyzer is a highly regarded open-source implementation of a language server that caters to the Rust programming language [2]. It is optimised to be compatible with various code editors that support the Language Server Protocol (LSP), thereby providing a more robust Rust development experience [2] [7].

Although Rust Language Server (RLS) and rust-analyser act as language servers for Rust, rust-analyser is renowned for its superior speed and broader range of features [2] [7]. While RLS was initially the official language server for Rust, rust-analyser has emerged as the new standard and is often the preferred choice for developers. As of 2022, the Rust Language Server (RLS) is gradually being phased out in favor of rust-analyser [2] [7].

### Rustfmt

An application designed to align the syntax of Rust code with established style conventions [35].

### svd2rust

svd2rust is a handy command line tool that converts System View Description (SVD) files into crates with a type-safe API [36]. This API is ideal for accessing the peripherals of a device with ease. An SVD file, which is an XML file describing the hardware features of a microcontroller, provides details on all available peripherals, as well as the associated registers' memory location and function [36].

### Kactus2

Kactus2 is a graphical toolset utilised to design embedded systems, primarily SoCs based on FPGAs [3]. Its key objective is to enhance development productivity by enabling the reuse, exchange, and integration of Intellectual Properties (IPs). The development and maintenance of Kactus2 are performed by the System-on-Chip

Research Group at Tampere University [4].

Kactus2 boasts numerous features, including the capability to package IPs for reuse and exchange, import existing IPs as IP-XACT components, generate HDL module headers for new IP-XACT components, utilize IP-XACT files from any standard compatible vendor, reuse IPs in designs, establish interconnections among wires and busses, create hierarchical HW designs, generate multilevel hierarchies, configure component instances in designs, including the sub-designs, utilize generator plugins to create HDL with wiring and parameterisation and integrate HW and SW [4].

Moreover, Kactus2 enables users to use a memory designer that allows previewing memory maps and address spaces. Users can package software to IP-XACT components and map them to hardware while generating makefiles that build executables with rules defined in IP-XACT components [3] [4].

**OpenOCD**

The Open On-Chip Debugger (OpenOCD) is designed to provide debugging, in-system programming, and boundary-scan testing for embedded target devices [12]. Accomplishing this relies on a debug adapter - a small hardware module that facilitates the proper electrical signalling required to debug the target. Debug adapters are necessary because the debug host, where OpenOCD is run, typically lacks native support for this signalling or the necessary connector to link to the target [12].

These adapters support one or more transport protocols, each utilising different electrical signalling and messaging protocols. Debug adapters come in various types and are often referred to by different names, with differences in product naming. They may be packaged as discrete dongles, sometimes called hardware interface dongles. Alternatively, some development boards integrate them directly, allowing the board to connect directly to the debug host via USB and, in some cases, power it over USB [12].

# 6 Feasibility of porting

To determine the feasibility of porting, it was essential to first establish a functional toolchain. Thanks to the assistance of Kactus2 and svd2rust, this was successfully accomplished. Kactus2 produced an SVD file from IPXACT, which was then passed on to svd2rust. This tool generated the Peripheral Access Crate (PAC), a secure and direct interface to the chip's peripherals [38]. This allows users to customize every detail to their exact requirements. Typically, the PAC is only necessary if the higher layers don't suffice or during development. The svd2rust file output was then formatted using the rustfmt tool, dividing the single file into multiple files corresponding to each peripheral.

### 6.0.1 Test setup

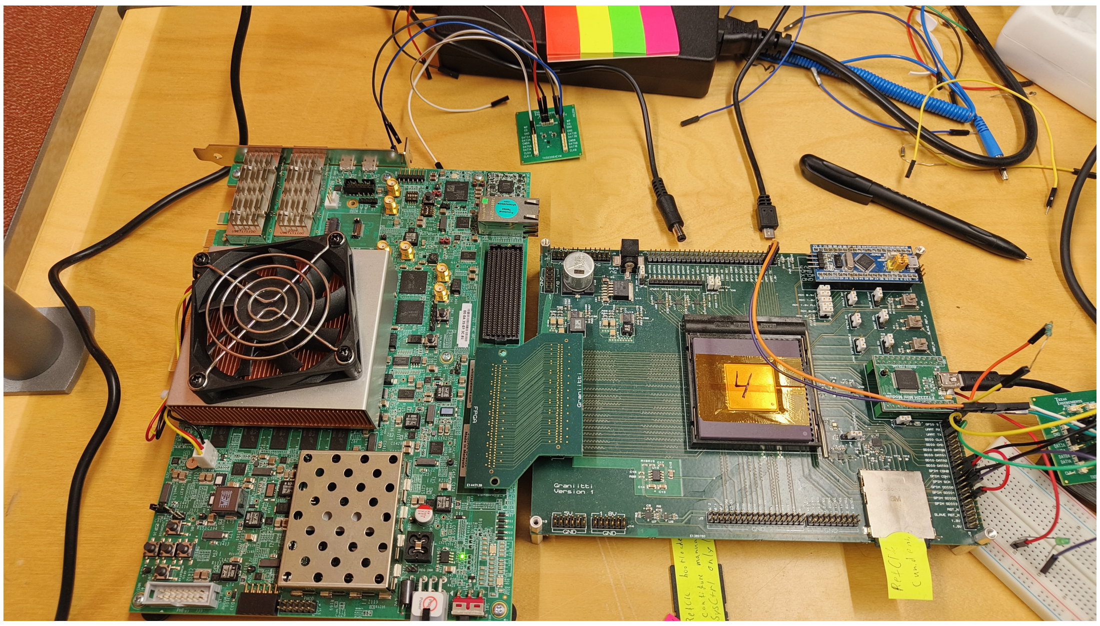


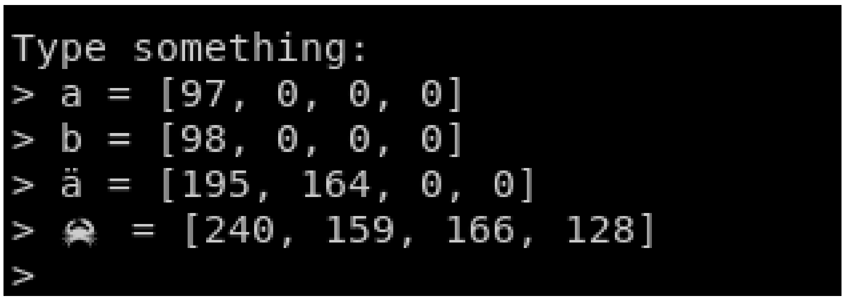
***Figure 6.1*** *Granitti Board with Ballast MPSoc setup for UART testing*

In order to conduct a thorough system evaluation, as seen in 6.1 the Ballast MPSoC was linked to the Granitti board, which boasted an FTDI chip and an RS232 board for UART communication. Subsequently, it was linked to the host

PC. The host PC was equipped with an openOCD server and GDB, which were activated using the following commands.

**sudo openocd –f config.cfg**
**riscv64gc -unknown -none-gdb executable**

A rudimentary UART transmission and reception application was developed and implemented to verify the connectivity upon successfully generating the PAC. Subsequently, the application, as mentioned earlier, was flashed onto the device to conduct a comprehensive test. Figure 6.2 demonstrates the application.

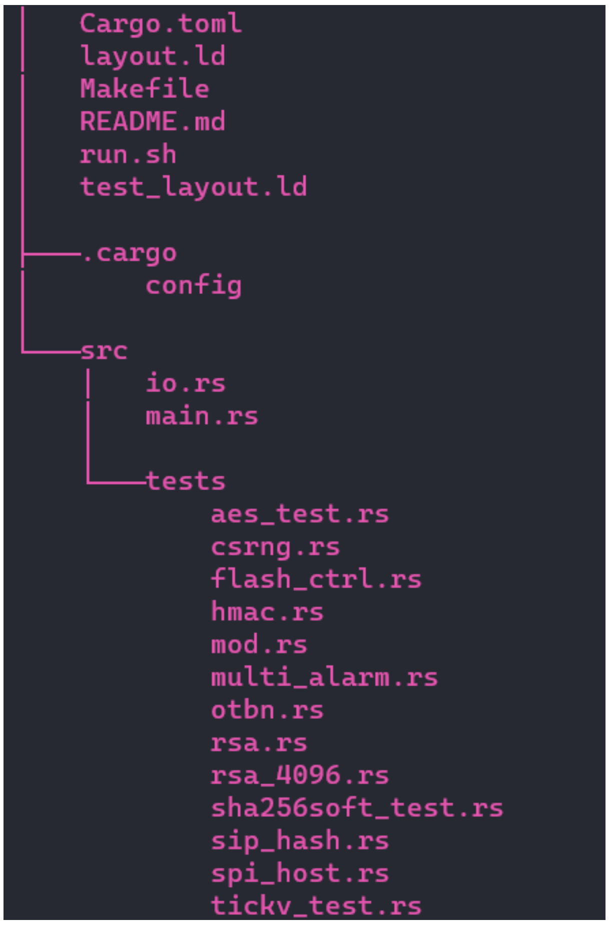


```
Type something:
> a = [97, 0, 0, 0]
> b = [98, 0, 0, 0]
> ä = [195, 164, 0, 0]
> 🦀 = [240, 159, 166, 128]
>
```

***Figure 6.2*** *UART transmission and reception application*

A comprehensive investigation was carried out to analyse the implementation of existing Tock OS ports. The study was centred on two platforms, namely BBC Micro:bit v2 and OpenTitan [13]. The former boasts tier 1 Tock OS support, a functional operating system port, and uses an Arm Cortex-M4. The latter is an open-source secure silicon ecosystem that provides silicon IP and complete top-level designs for various applications. OpenTitan features a discrete, secure microcontroller [13] and an integrated secure execution environment supporting Root of Trust functionality, secure boot, and DICE attestation. It was created in collaboration with lowRISC and leverage a RISC-V architecture [8].

After conducting research, it was noticed that the CVA6 RISC-V shared similarities with the RISC-V architecture found in the Ballast SoC's High-Performance Computing (HPC) subsystem. As Tock OS necessitates an MPU, HPC was the sole subsystem featuring one. Running Tock OS on a system lacking an MPU would require excising MPU-specific code from the kernel. However, as this research aimed to utilise Tock OS unaltered, HPC was deemed the appropriate choice. Consequently, a test plan was devised to assess the feasibility of operating a minimum viable version of Tock OS on the SoC.



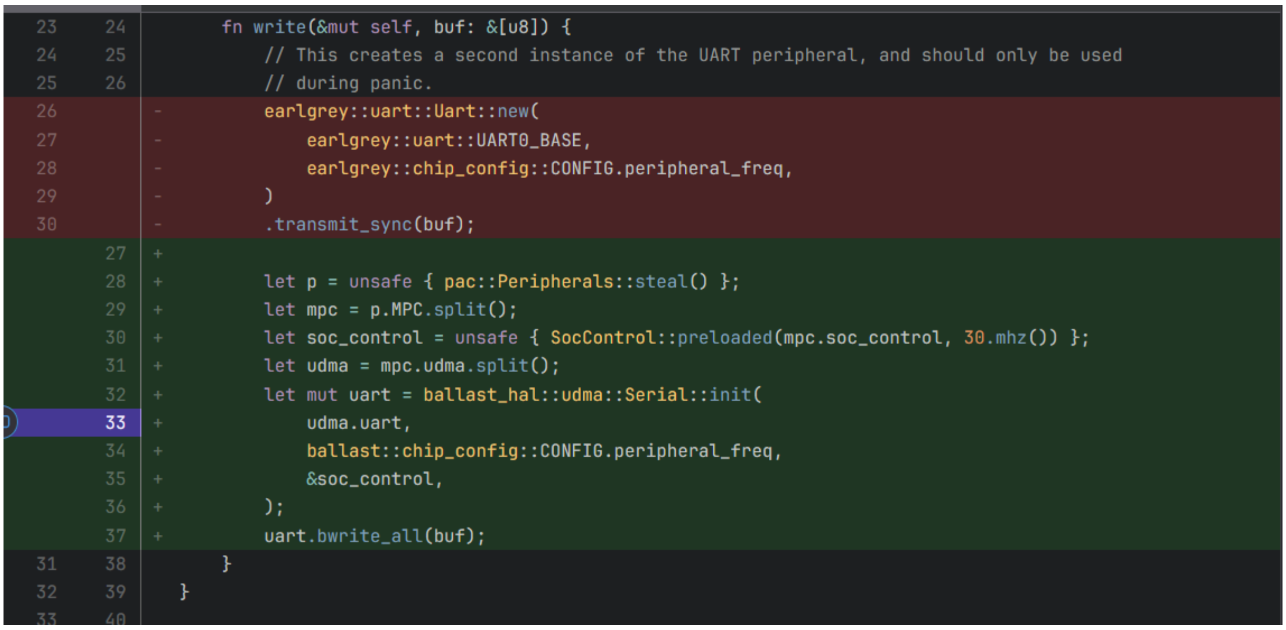
```
│   Cargo.toml
│   layout.ld
│   Makefile
│   README.md
│   run.sh
│   test_layout.ld
│
├───.cargo
│       config
│
└───src
    │   io.rs
    │   main.rs
    │
    └───tests
            aes_test.rs
            csrng.rs
            flash_ctrl.rs
            hmac.rs
            mod.rs
            multi_alarm.rs
            otbn.rs
            rsa.rs
            rsa_4096.rs
            sha256soft_test.rs
            sip_hash.rs
            spi_host.rs
            tickv_test.rs
```

***Figure 6.3*** *Directory tree of Granitti board files*

In order to accomplish this, the repository required the inclusion of Ballast SoC specific code. After considering the similarities between the HPC subsystem and the Opentitan EarlGrey SoC, it was determined that Opentitan would serve as the foundation. Initially, the pertinent Granitti board code was added to the **tock/Repository/boards**, including files such as a Cargo.toml file to manage project details and dependencies, a layout.ld file (figure 6.6) to outline the board layout, a Makefile for code building, and board-specific files for alarms, encryption tests, and flash controller.

```
MEMORY
{
  rom    (rx)  : ORIGIN = 0x20000000, LENGTH = 0x30000
  /* Support up to 0x2009_0000 for apps
   * and 0x2009_0000 to 0x2010_0000 is for flash storage.
   */
  prog  (rx)  : ORIGIN = 0x20030000, LENGTH = 0x60000
  /* The first 0x650 bytes of RAM are reserved for the boot
   * ROM, so we have to ignore that space.
   * See https://github.com/lowRISC/opentitan/blob/master/sw/device/
    silicon_creator/lib/base/static_critical.ld
   * for details
   */
  ram    (!rx) : ORIGIN = 0x10000650, LENGTH = 0x10000 - 0x650
}
```

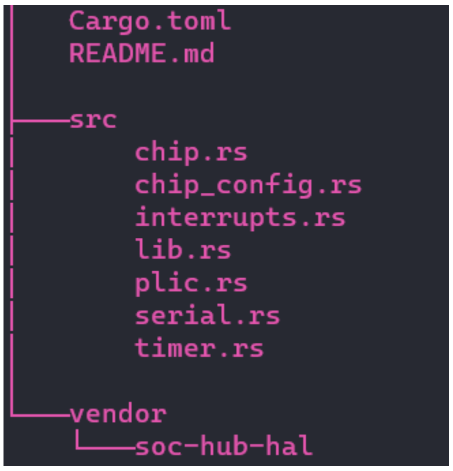
***Program 6.1*** *snippet from the layout.ld file of Granitti board.*



```
      fn write(&mut self, buf: &[u8]) {
          // This creates a second instance of the UART peripheral, and should only be used
          // during panic.
-         earlgrey::uart::Uart::new(
-             earlgrey::uart::UART0_BASE,
-             earlgrey::chip_config::CONFIG.peripheral_freq,
-         )
-         .transmit_sync(buf);
+
+         let p = unsafe { pac::Peripherals::steal() };
+         let mpc = p.MPC.split();
+         let soc_control = unsafe { SocControl::preloaded(mpc.soc_control, 30.mhz()) };
+         let udma = mpc.udma.split();
+         let mut uart = ballast_hal::udma::Serial::init(
+             udma.uart,
+             ballast::chip_config::CONFIG.peripheral_freq,
+             &soc_control,
+         );
+         uart.bwrite_all(buf);
      }
  }
```

***Figure 6.4*** *Diff showing the changes made for one of the Granitti board files*

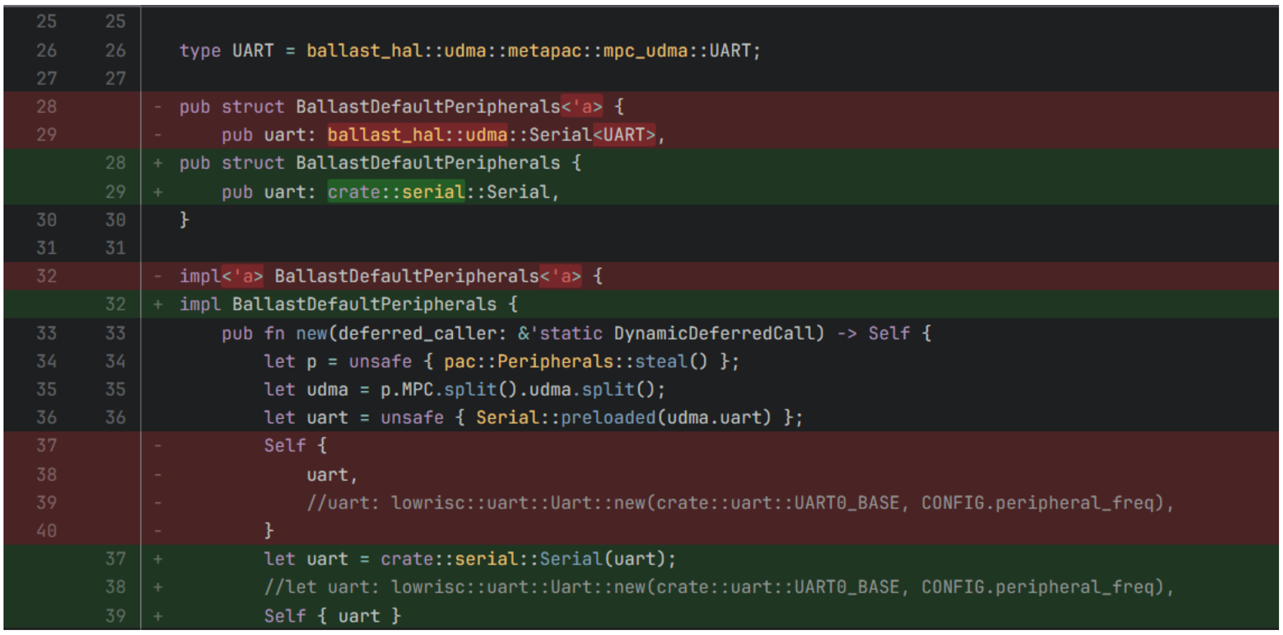
The next task was to incorporate the Ballast chip's unique code into **tock-/Repository/chips**.



```
│   Cargo.toml
│   README.md
│
├───src
│       chip.rs
│       chip_config.rs
│       interrupts.rs
│       lib.rs
│       plic.rs
│       serial.rs
│       timer.rs
│
└───vendor
    └───soc-hub-hal
```

*Figure 6.5 Directory tree of Ballast chip files*

This entailed adding Cargo.toml files for project dependencies, **chip.rs** to facilitate high-level setup and interrupt mapping, **chip_config.rs** to support chip-specific configuration, **gpio.rs** for handling GPIO instantiation, **i2c.rs** to configure i2c functionality, **interrupt.rs** to manage named interrupts for the chip, **lib.rs** for drivers and chip support, **plic.rs** for platform-level interrupt control peripheral driver, **timer.rs** to drive timers, and lastly **uart.rs** to handle UART functionality.



```diff

  type UART = ballast_hal::udma::metapac::mpc_udma::UART;

- pub struct BallastDefaultPeripherals<'a> {
-     pub uart: ballast_hal::udma::Serial<UART>,
+ pub struct BallastDefaultPeripherals {
+     pub uart: crate::serial::Serial,
  }

- impl<'a> BallastDefaultPeripherals<'a> {
+ impl BallastDefaultPeripherals {
      pub fn new(deferred_caller: &'static DynamicDeferredCall) -> Self {
          let p = unsafe { pac::Peripherals::steal() };
          let udma = p.MPC.split().udma.split();
          let uart = unsafe { Serial::preloaded(udma.uart) };
-         Self {
-             uart,
-             //uart: lowrisc::uart::Uart::new(crate::uart::UART0_BASE, CONFIG.peripheral_freq),
-         }
+         let uart = crate::serial::Serial(uart);
+         //let uart: lowrisc::uart::Uart::new(crate::uart::UART0_BASE, CONFIG.peripheral_freq),
+         Self { uart }
```

*Figure 6.6 Diff showing the changes made for one of the Ballast chip files*

# 7 Results

While running the test, it was determined that the operating system could not be executed on the HPC due to architectural differences. The test was plagued by multiple compiler errors, which, upon further investigation, were caused by Tock's insufficient support for 64-bit platforms.

Upon careful analysis, it became apparent that two key factors were hindering the feasibility of this research. Firstly, much code rewriting would be required to enable Tock OS to support 64 RISC-V architectures. Secondly, time constraints proved to be a crucial barrier, as the projected timeframe for such a comprehensive rewrite would exceed the scope of a master's thesis. The addition of support for Ballast MPSoC necessitated the modification of 41 files and writing approximately 4,500 lines of code (LOC). Incorporating support for a 64-bit architecture would require substantial additional work and time. As a result, it is not feasible to complete the project within the specified timeline.

### 7.0.1 Future work

With enough time, a 64-bit port can be achieved by updating the registers to 64-bit in all low-level startup and context switching codes. Any implementation supporting both register sizes is recommended to minimise the using cfg statements to avoid compiling different low-level assembly code depending on the RISC-V architecture. While code duplication should be avoided, balancing this with the risk of bug fixes only being propagated to one architecture is essential. [18]

To create a Tock OS port, the user must create a RISC-V 64-bit integer Architecture folder in the repository that follows RISC-V specifications. The next step is to use 64-bit load/stores in the architecture code. The following list of files in the repository must be modified for the porting process. [18]

- **kernel/src/config.rs**
- **kernel/src/memop.rs**

- **kernel/src/syscall.rs**
- **kernel/src/grant.rs**
- **arch/riscv/Cargo.toml**
- **arch/riscv/src/syscall.rs**
- **arch/riscv/src/csr/mcause.rs**
- **arch/riscv/src/csr/mod.rs**
- **arch/riscv/src/csr/satp.rs**
- **arch/riscv/src/lib.rs**
- **arch/rv32i => riscv/src/pmp.rs**
- **arch/rv32i => riscv/src/support.rs**
- **arch/rv32i/src/lib.rs**

After additional conversations with the Tock OS team and a Google engineer who also attempted a Tock OS 64-bit port, it was discovered that most of the porting process consisted of updating the code where **u32** was utilised to use **usize** instead. **"Places that assumed 32-bit did so formally by using u32, so changing to usize fixed most of the problems. Rarely did any code implicitly rely sizeof(usize) or alignof(usize) == 4."** (Lawrence Esswood, personal communication, January 2024).

This is because **usize** is pointer-sized, meaning its size depends on the architecture for which the program is being compiled. For instance, on a 32-bit x86 computer, **usize** equals **u32**, whereas on x86_64 computers, **usize** equals **u64**.

Changes were also made to the upcall traits, but wrapper APIs rarely used it directly; instead, they used the type that implemented it. So, the following changes were made to the implementation.

```
impl Upcall<AnyId> for core::cell::Cell<Option<(u32, u32)>> {
    fn upcall(&self, arg0: u32, arg1: u32, _: u32) {
        self.set(Some((arg0 as u32, arg1 as u32)));
    }
}
```

***Program 7.1*** *Before Refactoring*

```
impl Upcall<AnyId> for core::cell::Cell<Option<(u32, u32)>> {
    fn upcall(&self, arg0: usize, arg1: usize, _: usize) {
        self.set(Some((arg0 as u32, arg1 as u32)));
    }
}

impl Upcall<AnyId> for core::cell::Cell<Option<(usize, usize)>> {
    fn upcall(&self, arg0: usize, arg1: usize, _: usize) {
        self.set(Some((arg0, arg1)));
    }
}
```

***Program 7.2*** *After Refactoring*

This ensured that any existing APIs that wanted **u32s** could continue to have them, and new interfaces could use usize.

Before porting Tock to a new platform or microcontroller, it is imperative to conduct a thorough assessment to determine whether the platform is suitable for the intended purpose. Although no definitive criteria checklist exists, essential and generally expected criteria are typically evaluated.

The must-have criteria include memory protection support, typically in the form of the MPU on Cortex-M platforms or the PMP on RISC-V platforms, as well as at least 32-bit support. Since Tock is not designed for 16-bit platforms, this criterion is crucial. Additionally, the platform should possess sufficient RAM and flash to support userspace applications, with a minimum of 64 kB of RAM and 128 kB of flash generally considered sufficient.

Furthermore, the generally expected criteria require the platform to be 32-bit, although future support for 64-bit platforms may become available. Additionally, while a multi-core CPU is acceptable, Tock is designed to utilise only one core, and the platform should be single-core.

By adhering to these criteria, the transition of Tock to a new platform or microcontroller can be carried out effectively and efficiently, ensuring optimal performance and functionality.

It is advisable to involve the Tock OS team from the beginning of development. Their active developers can guide how to approach specific problems and implement features. This also ensures that the work adheres to their standards and meets quality requirements.

# 8 Conclusions

The present study aimed to investigate the feasibility of porting Rust-based secure operating systems onto a RISC-V MP System on Chip (SoC), focusing on implementing the operating system on a new RISCV SoC platform.

The study commenced with a detailed investigation of the Tock OS, encompassing its architecture, user space, kernel, hardware drivers, and security features. This was followed by exploring the Tock OS repository, project structure, and the development environment required to work on Tock OS.

Subsequently, a comprehensive breakdown of the requirements for a custom Tock OS port was presented, which included the hardware and software requirements, modifications in the codebase, related works, and boards supporting Tock OS. Additionally, the study highlighted some key features added during the Tock development history, along with some of the bugs encountered.

The target platform was introduced in the subsequent chapter, which included subsystems and the toolchain for the platform. The feasibility report elaborated on the approach to test the hypothesis and presented the results. The following chapter details the steps required to complete future work.

Overall, the hypothesis exploration was successful despite the unachieved end goal due to time constraints. The Ballast SoC toolchains worked well, and creating a PAC and using it to access various peripherals of the SoC proved successful. Most of the difficulties faced during the project were attributed to the lack of related work regarding 64-bit Tock OS and the absence of official support for 64-bit architectures during the research period.

The research yielded a comprehensive list of software requirements subsequently conveyed to the Ballast hardware team. The objective was to articulate the essential functionalities the hardware platform must possess to facilitate developers and operating systems in future versions of the MPSoC. The list encompasses a wide range of requirements, including support for 32-bit architectures with MPU, efficient management of interrupts (PLIC & CLINT(s)), provision of atomic instructions, standard embedded smclic (CLIC) compliant with RISC-V, and advanced RISC-V ACLINT, among other critical features.

In conclusion, the present study can contribute to exploring the potential of porting Tock on future SoC-Hub projects.

## References

[1] Bradford Campbell Amit Levy et al. "Multiprogramming a 64 kB Computer Safely and Efficiently". In: *Multiprogramming a 64 kB Computer Safely and Efficiently* (2017). URL: `https://tockos.org/assets/papers/tock-sosp2017.pdf`.

[2] Rust Analyzer. *Rust Analyzer*. URL: `https://rust-analyzer.github.io/`.

[3] System on Chip Research Group. *kactus2*. URL: `https://research.tuni.fi/system-on-chip/tools/`.

[4] System on Chip Research Group. *kactus2*. URL: `https://github.com/kactus2/kactus2dev`.

[5] Clippy. *Clippy Documentation*. URL: `https://doc.rust-lang.org/clippy/`.

[6] Clippy. *Clippy Lints*. URL: `https://doc.rust-lang.org/clippy/lints.html`.

[7] JetBrains. *Rust Analyzer*. URL: `https://www.jetbrains.com/help/fleet/using-rust-analyzer.html`.

[8] lowrisc. *lowrisc*. URL: `https://lowrisc.org/`.

[9] Antti Nurmi. *BootROM Development for a Novel Multiprocessor System-on-Chip*. 2022.

[10] NVIDIA. *NVDLA*. URL: `http://nvdla.org/`.

[11] OpenASIP. *TTA*. URL: `http://openasip.org/`.

[12] OpenOCD. *OpenOCD*. URL: `https://openocd.org/doc/pdf/openocd.pdf`.

[13] OpenTitan. *OpenTitan*. URL: `https://opentitan.org/book/doc/introduction.html`.

[14] Tock OS. *Architecture Agnostic Tock*. URL: `https://github.com/tock/tock/pull/962`.

[15] Tock OS. *Architecture Agnostic Tock*. URL: `https://github.com/tock/tock/pull/1113`.

[16] Tock OS. *Architecture Agnostic Tock.* URL: https://github.com/tock/tock/pull/1115.

[17] Tock OS. *Architecture Agnostic Tock.* URL: https://github.com/tock/tock/pull/1159.

[18] Tock OS. *RISC-V 64 bit support.* URL: https://github.com/tock/tock/issues/2332.

[19] Tock OS. "The Tock Book. (English)". In: (). DOI: https://book.tockos.org/getting_started.

[20] Tock OS. *Tock 1.0.* URL: https://github.com/tock/tock/releases/tag/release-1.0-2018-02.

[21] Tock OS. *Tock 1.3.* URL: https://github.com/tock/tock/releases/tag/release-1.3.

[22] Tock OS. *Tock 1.5.* URL: https://github.com/tock/tock/releases/tag/release-1.5.

[23] Tock OS. *Tock 1.5.* URL: https://github.com/tock/tock/pull/1338.

[24] Tock OS. *Tock 2.0.* URL: https://github.com/tock/tock/releases/tag/release-2.0.

[25] Tock OS. "Tock OS porting guide . (English)". In: (). DOI: https://github.com/tock/tock/blob/master/doc/Porting.md.

[26] Tock OS. *Tock Repository.* URL: https://github.com/tock/tock.

[27] Tock OS. "Tock Supported Architectures. (English)". In: (). DOI: https://github.com/tock/tock/tree/master/arch.

[28] Tock OS. "Tock Supported Boards. (English)". In: (). DOI: https://github.com/tock/tock/tree/master/boards.

[29] Tock OS. *TockLoader.* URL: https://github.com/tock/tockloader.

[30] PULP. "Pulp Platform Implementation". In: (). DOI: https://pulp-platform.org/implementation.html.

[31] Alexandru Radovici et al. "Embedded Systems Software Development". In: *Getting Started with Secure Embedded Systems: Developing IoT Systems for micro: bit and Raspberry Pi Pico Using Rust and Tock* (2022), pp. 1–521.

[32] Rust. *Packages and Crates*. URL: https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html.

[33] Rust. *Rust Traits*. URL: https://doc.rust-lang.org/book/ch10-02-traits.html.

[34] Rust. *The Cargo Book*. URL: https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html.

[35] rustfmt. *rustfmt*. URL: https://github.com/rust-lang/rustfmt.

[36] svd2rust. *svd2rust*. URL: https://docs.rs/svd2rust/latest/svd2rust/.

[37] SoC Hub Tampere. *Ecosystem for Finnish System-on-Chip design and co-creation*. URL: https://sochub.fi/.

[38] Rust Embedded Terminology. *PAC*. URL: https://docs.rust-embedded.org/discovery/microbit/04-meet-your-hardware/terminology.html.