WEB ASSEMBLY OPERATING SYSTEM FOR IOT DEVICES.



**UNIVERSITY OF NAIROBI**

**WEB ASSEMBLY BASED OPERATING SYSTEM FOR IOT DEVICES**

By James Kiarie

Registration Number: P15/ 1667/2019

Supervised by Dr. Wanjiku Ng’ang’a

November 2022

This project proposal has been submitted in partial fulfillment of the requirements of the Bachelor of Science in Computer Science as offered by the University of Nairobi.

# Acknowledgements

Dr. Wanjiku, for kind guidance.

Phillip Opperman, writer of “Writing an OS is Rust” blog. The blog singlehandedly made OS development to stop looking like a craft reserved for wizards.

Stephen Marz, writer of the “The Adventures of OS” blog. Took the OS development tutorials beyond just writing “hello world”.

Lin Clark, for simplifying complex webassembly jargon using mortal language... and simple cartoon illustrations.

The writers of “Wasmachine: Bring IOT up to speed with Web Assembly OS” research paper. This was the bedrock of this project.

All the Rust, wasm and riscv teams for creating an awesome well\_thought\_out tech.

# Introduction

It is a dream of many a computer scientist to control matter. To make all matter all around us programmable.

One step towards this direction is through embedded programming. Embedded programming on its own might sound simple, all you have to do is to read the data sheet of a piece of hardware, abstract that data sheet using data structures and finally manipulate the exposed registers using MMIO programming. Deployment looks simple too, sealing off the programming jack pin sounds enough.

But now, things have changed. Things have become complex. Both the hardware and software are more complicated. For example, reading the hardware data sheet is not enough, you have to understand the ISA, the corresponding hardwired security implementations and additional compatible circuit extensions. Embedded software now deals with network connectivity... Bringing in a whole set of cybersecurity modules and cloud integration drama.

Now the IoT architecture roughly looks like this [1] :

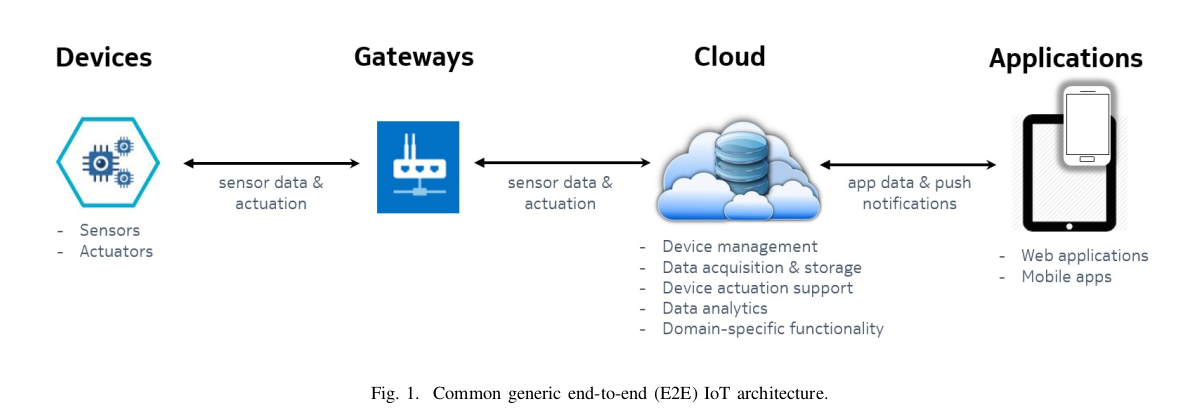
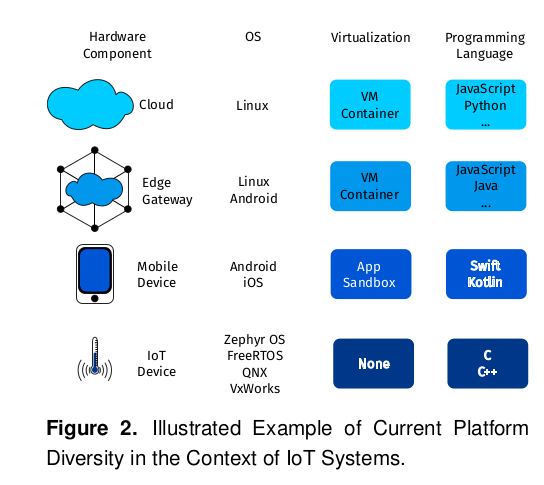


Figure 1 Image borrowed from Taivalsaari, On the development of IoT systems [1]

As a result of the above IOT architecture, building an average IOT system requires the development team to use a lot of unassociated technologies. For example; Assembly for the hardware, C/Rust for embedded programming, distributed programming for building the immediate network infrastructure, docker and kurbenetes for implementing microservices over the cloud, web languages to build a website that acts as a remote interface to the embedded devices... And probably kotlin for a mobile app that interacts with the embedded devices.

Here is a rough illustration of the different technologies used [2] :



This complexity can be summarized to three causes ; One, heterogeneity of the devices used in terms of hardware and runtime implementations. Two, inconsistency of the communication protocols used between the heterogeneous devices and finally, the necessity to use specialized tools and programming languages.

Below are the solutions as proposed in the paper by Mikkonen [2].

To solve the problem of heterogeneity of devices, Mikkonen proposes a universal API that describes the abstraction and interaction with known generic hardware devices. For example, the API should specify how to abstract and interact with cameras or heat sensors.

To solve the inconsistency of communication protocols used between heterogeneous devices, Mikkonen proposes a standard set of communication protocols to be specified for each known device-device interaction. The paper was in support for the use of existing web protocols for simplicity and easy adoption.

The above two problems have been partially solved by initiatives such as Web of Things Standards (WoT) and Open Communication Foundation (OCF).

The third problem ; having too many tools and programming languages involved in development, causes two kinds of complexity. One, It makes it hard for a small team to properly master and keep up with the tools involved. Two, It makes it hard to migrate code from one node to another within this distributed network of IOT devices. Code migration portability is important because of reasons like code re-usability or code maintainability. However, the major reason why code migration portability is important is because current IOT favors edge computing (fog computing). In edge computing, it is expected that at some point, the edge device may run the algorithms that are usually ran in the cloud (for example a data processing algorithm). If the data processing algorithm found in the cloud node is written in python, does that mean that the developer has to re-write that algorithm in C/Rust just for it to run on the edge device?

This code portability problem can be solved through two methods : One, finding a programming language or tool that gets explicitly and solely used through out the project OR find a way to package the programs written in different languages into a portable package.

Here is an image showing only one programming language being used versus many languages:

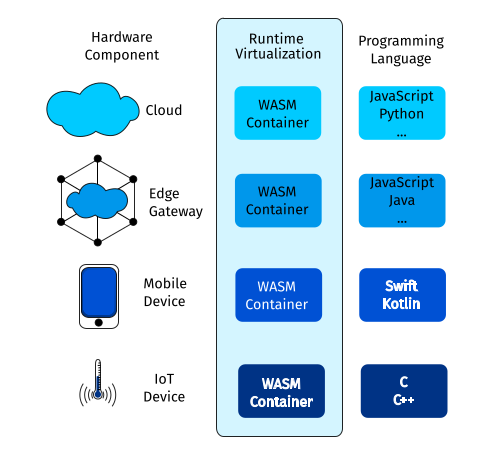
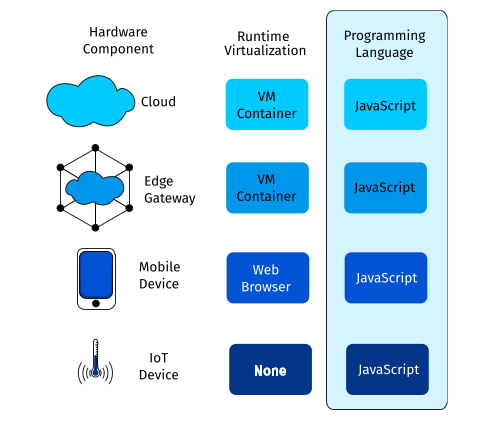


Figure 2 Forced to use 1 language (Left)

Figure 3 Multiple languages used but the packaging is constant (Right)

Solution one is unfavorable because each language or tool has its own special purpose, It is not okay to use a single language to solve every problem. For example, using Javascript to program micro-controllers using tools like Espruino is feasible, however this decision might cause inefficiencies. Javascript is an interpreted language, the JIT compiler continuously runs as the program runs, this might guzzle too much energy. Moreover, if you derail off the JIT’s happy path, the performance of the program might become unpredictable.

So developers opt to use Javascript to make the User-facing apps and use C/C++/Rust to write code for the devices. Python and Julia are preferred for writing the data processing algorithms.

Solution two is considerably favorable ; developers use the languages they are comfortable in, languages that were purposefully made for those jobs; but all the programs get compiled to web assembly binary. In a sense, all programs have been written in one language, targeting one machine.

This solves the code migration problem by making the resultant programs portable ; each program can run on any machine, as long as that machine can run a webassembly runtime. The portability is achieved by packaging the programs as containers or setting up compatible virtual machines on top of the incompatible execution environments.However, solution two does not reduce the number of technologies being used simultaneously. Developers still use a combination of many languages and tools.

It is hard to build a one-size-fits all tool; at least solution 2 acknowledges that, unlike solution 1.

Solution two is majorly implemented through the use of container technologies like Docker. There have been propositions and early implementations of using web assembly containers instead of mainstream solutions like Docker. This is because web assembly containers are comparably smaller in size and the corresponding wasm runtimes also have a small memory occupancy.

But if we look at the comparison objectively, Docker has the same properties as wasm functionality-wise. The true strength of wasm lies in the implementation of a universal host system API ; WASI (Web Assembly System Interface).

With WASI, we can package containers for any host system, be it an operating system or an Application like Word or VScode. As long as the systems API of that host system is loosely defined, you can interface a wasm module with it. WASI is just like POSIX, but it is meant for more than just operating systems.

It is better to choose tools whose design and purpose will last for a long time... Hopefully.

## Problem Definition

Human Problem :

Developers are enticed to package their Software into wasm modules so as to reap its benefits; portability, compactness and security. Wasm containers have the potential to be a defacto standard for deploying IoT apps. However, developers find it hard to deploy and run wasm containers on embedded devices at near native speed.

The wasm ecosystem is immature BUT quickly growing. Most of the available wasm tools are browser-focused, only a few projects focus on building wasm tools for the IOT space.

One tool that would be beneficial to have is a embedded system operating system that safely executes wasm containers on top of CPUs at near-native speed.

At the moment, developers who need such a tool have resorted to two solutions ;

On one hand, developers embed the wasm runtimes in their bare-metal program executables. Let us call this method Method\_A. This method is not favorable because it means that the developer will not be able to run multiple programs on the same riscv CPU. To run a new program, the developer has to adjust the code for the executable and re-compile it. Having an OS enables one to run multiple isolated processes on top of a shared hardware, without having to recompile the kernel. Furthermore, this process is unsuitable for modern IOT development because it makes remote code modification impossible.

On another hand, developers resort to running the wasm runtime on top of another operating system like Tock or Linux. Let us call this method Method\_B. This method allows swapping of programs and remote code modification. You get to enjoy the capability security system of wasm and use the extensible WASI interface. However this method is unfavorable because the layers of abstraction sacrifice out the execution speed of the wasm programs. The wasm programs run on-top of a wasm runtime that runs on-top of an OS.

### Technical Problem

Running wasm containers on top of a runtime that runs ontop of an OS is slow because the execution experiences two kinds of latencies ; System-call-normalization latency and context-switching latency.

The System-call-normalization latency is caused by the normalization process. The normalization process is the mechanism of converting a WASI function call to have the same function signature as the host kernel’s system calls. Normalization happens when there is a difference in the function prototypes of the WASI API syscalls and the Native syscalls.

System-call-normalization happens during the compilation of wasm files to the corresponding native binary files. When an AOT compiler is used, the normalization latency does not affect the execution speed of the binary file at runtime. If a JIT compiler or an interpreter is used, the System-call-normalization latency will affect the execution speed of the binary file at runtime.

Context switching latency occurs when the wasm executable calls for a system call function via WASI. This is because the CPU stops executing the wasm process and instead switches to the kernel process, just to execute a system call function. The switching between the two processes is costly but necessary, but this is made worse by the fact that the CPU has to additionally switch modes.

### Solution

The solution to the afore mentioned human problem is to build an embedded operating system that runs wasm containers at near-native speed.

The solution to the mentioned technical problems is to eliminate or reduce the intensity of the two latencies responsible for impeding the execution speed of wasm containers. The methods or reducing the two latencies has been discussed below.

#### Handling the System-call-normalization latency

The system-call-normalization latency can be completely eliminated by making the kernel export its native system calls using the prototypes used in the WASI API [3], this means that the webassebly runtime will not need to normalize functions.

#### Handling the Context-switching Latency

It is possible to completely eliminate the context switching expense by making the wasm modules part of the kernel code. Such that the CPU will only execute one process through out its up-time. But this solution requires the developer to compile the wasm modules together with the kernel code. This means that whenever the wasm app gets updated, the developer has to recompile the entire kernel. This is highly unmaintainable. This solution is suitable when performance is more important than maintainability. It is especially unsuitable in the IoT space where it is ideal to update device programs via the network.

Completely eliminating context switching comes at the cost of maintainability, for this reason, it becomes more viable to focus on reducing the cost of context switching instead of completely eliminating it.

Context switching from a user-level process to a kernel-level process is more expensive than context switching from a kernel-level process to another kernel-level process. This is because switching from user-level process to a kernel-level process requires more overhead : The CPU has to switch modes, re-map the Memory Management page tables and switch the execution stack. On the other hand, switching from one kernel-level process to another kernel-level does not require the CPU to switch modes and re-map the page tables ; although it may still have to perform a stack switch.

#### Chosen Solution

Based on the above discussion on the different methods of solving the latencies, the project went with building a kernel that :

1. Has the wasm runtime embedded in it, as part of the kernel code
2. Uses an Ahead-of-time compiler to compile the wasm modules
3. Exports its system calls using WASI function prototypes
4. Runs the loaded wasm containers in kernel mode.

### Objectives

For the project to be complete, the following research and implementation objectives had to be completed.

#### Research Objectives

1. Understand the different kernel designs and implementations
2. Understand the implementation aspects of wasm containers and wasm runtimes
3. Understand the IOT development infrastructure and figure out the problems faced in IOT development.
4. Understand the Docker technology and its downsides
5. Understand the relevance and usage of the Riscv CPU.
6. Research on how to integrate WASI system calls to the kernel

#### System Objectives

1. Build a kernel that supports user input and output
2. Build a kernel that has a virtual page management system
3. Build a kernel that supports a file system
4. Extend the minimal kernel to embed a web assembly runtime
5. Modify the extended kernel to export its system calls using WASI function prototypes

### Project Justification

Web Assembly programs execute slowly in embedded devices when compared to native programs. Native programs have good performance but with poor portability and security control. This project aims to implement an OS that makes web assembly programs run at near-native speed in embedded devices while still taking advantage of the capability security system of wasm.

The resultant operating system will be a proof of concept that wasm-based operating systems will be the default operating systems in devices used in the IOT infrastructure. The OS acts as a proof of concept of the wasmachine proposed in the paper [4] with the aim achieving an isomorphic IOT architecture [2].

### Scope

The resultant kernel will only run on top of a Qemu-emulated Riscv CPU. The kernel will not be able to run on other ISAs.

The resultant kernel will not have complicated but necessary features such as networking capabilities or sound and graphics. The kernel will only be limited to the functionalities specified in the System Analysis Chapter.

# Literature Review

The literature review focuses on the technologies used withing the IOT infrastructure and how they fit into the project. The conclusion section of the literature review outlines the summary of the literature review and how it affected the design and implementation of the kernel.

## The IOT development structure

IoT development requires a range of software technologies, from mobile development to cloud and analytics. The common end-to-end architecture for IoT systems consists of edge devices, gateways, cloud, and applications.

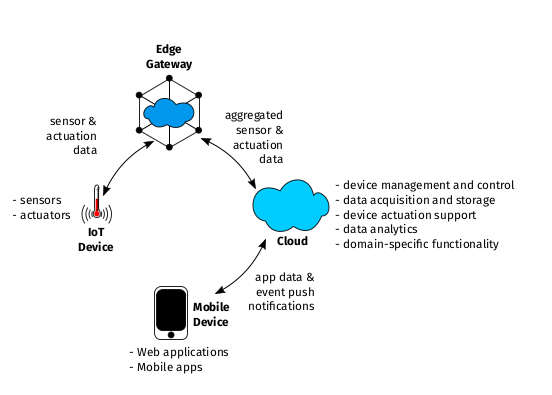


Figure 4 IOT architecture

IoT systems encompass embedded development, meaning that at the edge, the developer has to deal with a resource constrained environment. Embedded devices typically have low processing power, low memory and limited energy supply.

The EdgeGateway connects the edge devices to a local network or the internet and consequently a cloud platform. Edge devices or gateways are intermediary devices located closer to the devices/things, often at the edge of the network. They collect, pre-process, and filter data from connected devices, reducing latency and network traffic by performing local data processing and analysis. Edge devices may also provide additional functions like protocol translation and security features like monitoring.

Cloud services form the backend infrastructure that stores, processes, and analyzes the data collected from IoT devices. They enable centralized management, real-time monitoring, and remote control of connected edge devices.

The Web and Mobile Apps provide a way to interact with both the cloud and edge devices, they provide a form of user-interface to the backend. IoT app development is dominated by mobile and web development frameworks more than desktop frameworks like Tauri or Electron.

Under this architecture, the backend services are packaged as independent micro-services; this helps in making the backend software more modular. The micro-services get wrapped as containers. A popular container choice is Docker. To mange the containers, developers employ container orchestration software such as Kurbenetes.

The bigger dream of IOT development is to build a programmable world, ubiquitous technology. To achieve this vision, fog computing and edge computing gradually emerged. These two technologies have no clear definition, they roughly propose that computing tasks should be transferable across many nodes. The main point being that the cloud should not be the only place where core processes are executed. This proposition is achievable because the processing power, storage and energy usage found in embedded environment has improved; we now have advanced CPU chips and high-volume storage components in the embedded chips.

The challenge that arises is that the programming languages and tools used to write and execute the data processing applications found in the cloud might be not be supported by the runtime software found in the edge devices. As a result the software portability required by the fog computing proposal becomes hard to implement. As mentioned in the report introduction chapter, the solution to this is to : define communication standards, API standards and use an isomorphic wrapper(container).

## Web Assembly

Web assembly is an intermediate language for a virtual stack CPU. This intermediate language can be represented in assembly form or binary form. In other words, it is a language made for a virtual ISA (Instruction set architecture)... And it so happens that the architecture is stack-based.

Quick detour... Abstraction is an important concept in Computer Science, if a problem has too many details or multiple implementations... Just abstract it in simple wrappers and let the dirty implementations stay under that wrapper . For example, in the past, there were so many Operating systems cropping up. Every kernel developer came up with unique ways of implementing system calls. To simplify things, developers agreed to create a common System call abstraction, POSIX. Under this abstraction, they defined standard system call function signatures. From there on, developers had to use those agreed upon signatures but underneath those prototypes, they had the freedom to implement the functions in their own way.

[img]

Back to webassembly. Webassembly employs the same abstraction technique. It abstracts an execution environment. Core webassembly abstracts a CPU. Combining Wasm+WASI abstracts the CPU + the runtime running ontop of that CPU. That runtime could be an OS or another application — it could be anything that has an exposed API.

[img]

Being that core webassembly abstracts CPUs, the wasm bytecode can be compiled to suit specific CPU ISAs. This makes wasm bytecode to be Architecture-independent.

Using the WASI API, web assembly abstracts any underlying system, be it an operating system or an ethereum system. As long as you use the WASI prototypes in your wasm bytecode, that code can be compiled for that specific implementation of the target system. This makes wasm platform-independent.

All high level programming languages need to get compiled to the CPU target architecture. Using the LLVM compiler, a developer can compile any high level language to target the wasm virtual machine. This will in turn generate a .wat or .wasm file. A .wat file stands for (webAssembly Text). This .wat file contains human readable bytecode. The .wasm file contains machine code meant for the wasm virtual CPU.

The webassembly file is known to be compact and memory-efficient. This is greatly attributed to the fact that the bytecode targets a stack-based system instead of a register-based system thereby reducing the number of arguments required for each assembly instruction.

## WebAssembly Runtimes

A runtime may mean a lot of things. In the context of this section, the word runtime means a piece of software that provides and manages resources needed by another running process. For example, if a person plays a video game on a windows operating system, the windows operating system can be referred to as the runtime for the video game.

A web-assembly runtime is a piece of software that provides and manages resources needed a wasm program. The web-assembly runtime interfaces with the underlying host system. For example, the run-time might interface with a kernel or a bare metal hardware.

[img]

A wasm runtime acts as the intermediary between the host system and the wasm modules. It performs the following functions :

**Module loading and validation**: The runtime locates the location of the wasm module, reads it and stores the content in a read-only buffer. It then validates the WebAssembly module, ensuring that it adheres to the WebAssembly specifications and security requirements. It verifies the module's structure, type signatures, and validates its bytecode.

**Module compilation :** For the webassembly modules to get executed by the underlying CPU, it needs to be compiled to native machine code. The Wasm Runtime contains a compiler specifically for this job. Note that the compiler might be of any kind depending on the runtime’s specific implementation. For example, the compiler might be an Ahead-Of-Time compiler, a Just-In-Time compiler or even a combination of both.

**Import and export handling**: WebAssembly modules can import functions and data from the host environment, such as JavaScript. The runtime facilitates the interaction between the WebAssembly module and the host environment by handling imports and exports. It resolves the dependencies and connects the module's imports to the corresponding functions and data in the host environment.

[img]

**Memory management:** The runtime handles the memory management for the WebAssembly module.By interacting with the underlying operating system, It initiates commands that allocate and manage linear memory, which is a contiguous block of memory accessible to the module. The runtime provides functions for allocating, resizing, and manipulating this memory.

**Execution:** By interacting with the underlying OS, the runtime executes the WebAssembly module's bytecode instructions. It provides an execution environment where the module can run, interpreting or compiling the bytecode to machine code for efficient execution. The runtime manages the execution stack and handles control flow, including function calls, loops, and branches.

**Capability-Based Security :** By default, wasm modules are sandboxed, they cannot invoke any system calls or host functions. To do so, they reference the WASI functions in their bytecode OR they explicitly import host functions. Each function call made by the wasm module gets inspected and validated by the wasm runtime. The Runtime checks if the function affects only the accessible resources, For example, if the wasm module is given access to file “x” only, the runtime will flag an error when the module tries to access any other file apart from file “x”.

**Garbage collection:** Some Wasm runtimes incorporate garbage collection mechanisms to automatically reclaim memory that is no longer in use by the WebAssembly module. Garbage collection helps manage memory resources efficiently and prevents memory leaks.

**Interoperability:** Wasm runtimes often provide interfaces or APIs that allow the WebAssembly module to interact with the host environment. This enables communication between the WebAssembly module and the surrounding application or system. For example, it may provide functions to access the file system, network, or other platform-specific capabilities.

**Performance optimization**: The runtime may employ various techniques to optimize the execution of WebAssembly modules. This can include just-in-time (JIT) compilation, where the bytecode is dynamically compiled to machine code for improved performance. Runtimes can also utilize ahead-of-time (AOT) compilation to generate optimized machine code before execution.

Some of the runtime examples include : wasmtime, wasmer and wasmi. There are tens of runtimes listed on github [5].

### Security Capabilities of Wasm and Wasm Runtimes

WebAssembly (Wasm) and Wasm runtimes provide several security capabilities to ensure the safe execution of code. Here are some of the key security features:

**Sandbox Execution:** Wasm code runs within a sandboxed environment, which isolates it from the host system and other code. This sandboxing prevents malicious code from accessing sensitive resources or interfering with the underlying system. The mechanism for implementig this sandbox execution is entirely up to the wasm runtime.

**Memory Safety:** Wasm enforces memory safety by using a linear memory model with bounds checking. This prevents buffer overflows and other memory-related vulnerabilities that can lead to security exploits. Wasm runtimes ensure that memory accesses stay within the defined bounds, preventing unauthorized access to data. A wasm module cannot directly access memory that is not withing the allocated linear memory.

**Validation and Verification**: Before execution, Wasm modules are validated and verified by the runtime. This process ensures that the module adheres to the Wasm specifications, has correct type signatures, and is free from structural errors. Invalid or malicious modules are rejected, preventing potential security risks.

**Sandboxed APIs:** Wasm runtimes provide a set of sandboxed APIs that allow controlled interaction with the host environment. These APIs provide limited access to system resources, such as file I/O, networking, or graphics, while preventing direct access to sensitive operations or resources. This helps mitigate potential security threats by enforcing access restrictions. This sandboxed APIs ensure that a wasm module only gets to use certain functions only. For example, you can refuse to provide the networking API to a calculator app. This forms the basis of capability-based security.

**Controlled Imports and Exports**: Wasm modules can import and export functions, but the runtime controls these interactions. Import functions are explicitly linked to trusted and safe host functions, preventing unauthorized access or execution of arbitrary code. Exported functions are also subject to runtime enforcement, ensuring that they are used appropriately and securely by the host environment.

### Porting of webAssembly Runtimes to No-std environments

Majority of the wasm runtimes were built to run on the browser. But there have been many upcoming runtimes that have been built to run off-browser. For example wasmtime, wamR, Wasmi and Wagi. A good comparison of the different runtimes has been done by Appcypher[5]

Some of them have been implemented as standalone command-line programs while some have been implemented as language-embeddable libraries.

Wasmtime was a compelling option being that it supports AOT compilation. Wasm3 was embeddable in a no-std environment but it is written in C and it has an Interpreter instead of an AOT compiler. Having an AOT compiler is crucial. An interpreter or a JIT compiler introduces unpredictability in performance and resource inefficiency; they waste energy, time and memory.

Wasmi proved to be the best candidate. It was the only embeddable runtime that easily ran in a no-std environment. It can easily be imported as a rust-library crate. Porting wasmtime proved to be too hard a hill-climb. Wasmtime is tightly coupled to the operating systems that it runs on. The downside is that wasmi uses an interpreter instead of an AOT compiler.

### WASI and POSIX

WebAssembly System Interface (WASI) is a system interface that has been developed to enable WebAssembly (Wasm) applications to run securely and efficiently across different platforms and operating systems. WebAssembly itself is a binary instruction format designed for safe and efficient execution on web browsers and other environments. WASI extends the capabilities of WebAssembly beyond the web, allowing it to be used in a wide range of contexts, including server-side applications, edge computing, and more. [6][7][8].

For example, wasi-nn is an extension to WASI that provides neural network inference capabilities. It enables WebAssembly applications to perform machine learning inference using standardized APIs, making it easier to deploy and execute machine learning models

One of the primary goals of WASI is to provide a consistent runtime environment for WebAssembly applications across different platforms, architectures, and operating systems. This means that developers can compile their code once and have confidence that it will run consistently across various execution environments.

POSIX on the other hand is a System Interface description purposefully built to abstract Operating system functionalities. POSIX defines a standard set of interfaces, functions, and behaviors that operating systems should provide if they want to be POSIX-compliant. The goal is to ensure application portability across different Unix-like operating systems.

**Advantages of POSIX in IoT:**

1. Compatibility: Many existing IoT devices and systems might already have POSIX support, making it easier to develop and port applications to these devices.
2. Familiarity: Developers who are experienced with POSIX programming might find it more comfortable and familiar to work with this standard, as opposed to learning and adopting new technology like WASI.
3. Legacy Systems: IoT deployments often involve older devices that lack the resources or compatibility for new technologies. POSIX support could cater to such legacy systems.

Advantages of WASI in IoT:

1. Security Isolation: One of the key benefits of WASI is its strong security isolation. It restricts applications from accessing the underlying system in ways that could pose security risks. This is crucial for IoT devices, where security vulnerabilities can have serious consequences.
2. Portability: While POSIX provides compatibility across various Unix-like systems, it might not cover the entire IoT landscape, which includes diverse devices with different operating systems or even bare-metal environments. WASI containers offer a more consistent runtime across a wider range of systems.

Example: Let's say you have an IoT device that uses a specialized real-time operating system (RTOS) that doesn't fully adhere to POSIX standards. Porting a POSIX-based application to this device might require significant modifications due to differences in APIs and behaviors.

While POSIX provides portability within the scope of Unix-like systems, WASI extends this portability to a wider range of environments.

### Kernel Designs

The kernel designs vary, there are no definite designs. For example, one might say that kernel A is a micro-kernel because the kernel services run as user processes, while another person might say that the same kernel is a monolith because its drivers are implemented as part of the core kernel. Naming designs is subjective.

Due to the subjectivity, this chapter describes analyzed kernels. Whether they are real-time Operating systems, monoliths or micro-kernels or any other categorization , that is completely up to the reader.

#### Tock

The Tock Operating System is an open-source, event-driven operating system designed primarily for low-power, embedded systems and Internet of Things (IoT) devices. It is specifically engineered to address the challenges posed by constrained hardware resources, real-time requirements, and security considerations. [9][10]

Here are some key design aspects of the Tock Operating System:

**Event-Driven Architecture**: Tock uses an event-driven architecture, where applications and kernel components respond to asynchronous events, such as sensor readings or timer expirations. This approach allows for efficient use of resources, as components are activated only when necessary

**Multithreading and Isolation:** Tock supports multithreading, allowing multiple tasks to run concurrently. Each task has its own stack and memory space, which helps prevent one task from interfering with another. This isolation improves system reliability and security.

**Small Kernel Size**: Tock's kernel is designed to be compact, which is important for resource-constrained devices. This small kernel size minimizes memory overhead and leaves more resources available for applications.

Applications are loadable, one need not compile the application together with the kernel. Tock seperates the Apps from the kernel using the Memory Protection Unit; to this regard, Tock assumes that the host system has an MPU. But if the underlying host architecture does not have an MPU, one can configure the kernel to use a virtual MPU.

##### Tock architectural Design

[img] : Tock system

The drivers (both low and High) are not part of the Kernel. The Apps and Services are also not part of the kernel. The Apps are programs that users can directly interact with.

The services are programs that continuously run in the background and are meant to be summoned by other software. They are not typically used by users. They can be things like a server.

The low\_level drivers are pieces of software that come pre\_witten inside the hardware. They can directly interact with the hardware. These drivers expose a Hardware Interface Layer to the Kernel.

These low level drivers are Board specific and device specific. The kernel needs to find a way to abstract this. The kernel takes this low level and specific Interface and exposes it to the Higher level drivers using a kernel\_defined\_standard Interface.

The capsules (ie. High\_level\_Agnostic\_drivers), provide APIs that can be imported to be part of the SYSCALL interface.

Now, the drivers need to access registers directly. But this requires the Driver\_writers to use unsafe RUST.

The kernel is considered memory safe because the compiler did all the memory checks at compile time. The compiler does not check memory safety within unsafe rust blocks. So the tock kernel developers were compelled to reduce the number of unsafe blocks in any kernel Rust code.

The problem becomes that all drivers will end up having unsafe code. This is because registers are accessed using raw pointers. Tock solves this problem by defining a register\_interface below the low\_level\_drivers.

This register interface provides consistent safe functions for accessing memory. This safe functions are unsafe functions that have been wrapped with safe code. This ensures that all drivers do not get to have the necessity to declare any unsafe block. They only have to call the memory\_interface safe functions. Tock does not allow any unsafe code in the Capsules.

Functions of the Tock Kernel :

1. Schedule the Apps and services. Who gets to use the CPU?

2. Memory Management : Allocate/deallocate/track memory needed by the Apps and Services

3. Provides mechanisms for Inter-process communication

4. Providing an interface for the Apps and Services

5. Providing an interface to drivers (High\_level\_drivers AND low\_level\_drivers)

##### **System Calls**

Tock uses both synchronous and asynchronous system calls.

Synchronous calls for simple tasks that take negligible time : For Example getting the number of GPIO pins (General Purpose I/O pins).

Asynchronous calls for operations that might take some time : E.g. reading and writing to a peripheral device.

##### **The Problem of Dynamic Memory Allocation**

[undone]

#### Docker

[undone]

#### Kurbenetes

[undone]

## Conclusions from the Literature Review

It was better to implement a monolithic kernel because it would be easier to design and implement. Considering that the applications run in isolated wasm environments within the wasm runtime, it would be an overkill to further isolate them to the user address space(micro kernel architecture). Furthermore, the concept of the project is to run user processes in kernel mode.

Wasm containers seem more promising than Docker containers in terms of resource efficiency and portability. The wasm containers are smaller and the wasm runtimes themselves have small memory footprints.

Sticking to WASI over POSIX will reduce the latency caused by the function-normalization. Furthermore, the operating system will be more extensible to different fields if need be.

Just like the Tock system, I/O system calls and other heavy operations will be done asynchronously while quick operations will get done synchronously. This will ensure that the CPU does not stay idle as it waits for responses from the I/O devices.

The fact that Kurbenetes has support for wasm containers further encourages the notion that maybe in the future, wam containers will become the norm.

The problem of dynamic memory allocation may have an impact in real-life critical embedded systems but considering that this is a learning project, a system that implements the proposed heap allocator would be an overkill and hard to implement. The Heap estimations would also be hard to gauge.

The project will use the Wasmi runtime because it supports no-std capabilities. Wasmi is easy to port to a bare metal environment.

The kernel will be non-POSIX compliant. Using the C library would be hard because the kernel itself will not have the ability to import all of the essential system calls needed by the C library

# Methodology

This section describes the process used in actualizing the project

Being that the project was an uncharted territory for the implementors, there was no clear plan at the very beginning. The plan only became clearer as the research and half-implementations were undertaken.

The process was divided into six phases :

1. Knowledge and skill building
2. Research
3. System Analysis
4. System Design
5. Implementation Phase
6. Support Phase (Testing and Integration)

## 1. Knowledge and Skill building

Under this phase, the developers were required to get familiar with the business domain and the technologies involved. The business domain being **the provision of development tools** for the IOT development.

Knowledge and skill building boiled down to this : Reading the necessary research papers associated with the project, familiarizing with the technologies involved, implementing any modules within reach and repeating the loop.

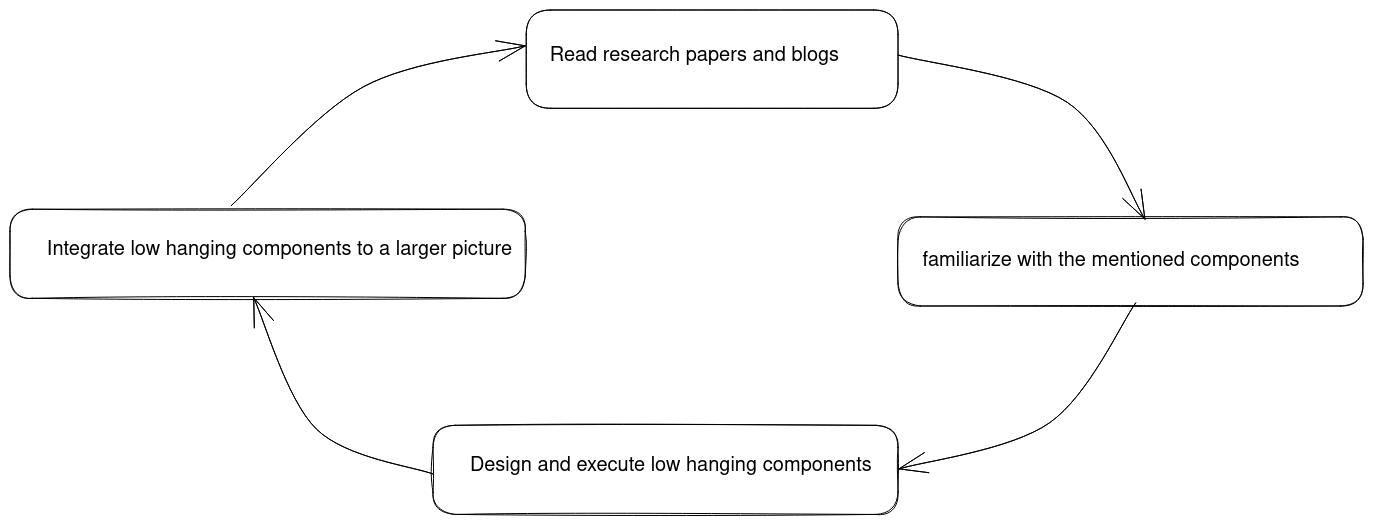


Figure 5 Methodology

## 2. Research Methodology

The research methodology involved looking for scholarly documents concerning the involved subjects and understanding the applicable components. This mainly touched on the web assembly and IOT infrastructure topics. The key papers have been referenced in the appendix section of the report.

As for the Rust and Riscv knowledge, the official documentation and online tutorials were sufficient. The books used have also been attached in the Appendix.

The Kernel design and implementation knowledge was extracted from three blog tutorials and referenced text books in the appendix:

ChatGPT was also used as a guide during research, It helped save days of googling or reading textbooks and datasheets with hope of finding a specific line of knowledge. Sometimes it was inaccurate but reading official documentations about datasheets or softwares clarified issues quickly.

To help retain the knowledge gained, the Feynman technique was applied, the implementer wrote a tutorial [11 ]as they researched and implemented the project.

No questionnaires or interviews were conducted. This project was not aimed to solve anyone’s problems in particular. The project was aiming to create a cool tool.

## 3. System Analysis

System analysis involves understanding the business requirements and determining which features and functionalities would be needed to satisfy those business requirements in a feasible manner. Feasible in terms of time, knowledge, money and human energy.

Considering operating systems are part of an established domain, the business requirements were pretty clear. For example : The operating system should be able to support a hard-disk and a filesystem, the OS should support multiple processes, the kernel should export certain system calls. The business requirements are **endless** in this case. The designs of the internal implementations are endless and with different levels of difficulty.

With regards to the above statements, the question for system analysis changed from “what business requirements need to be satisfied?” to “what can the developers implement within the specified time?”

Here is a fact : Only a person who has done a certain project can estimate their capabilities in doing that project. If you have never swam, can you estimate how far you can swim? If you have never built a block driver, can you estimate the feasibility of implementing a block driver?

The choice of the functionalities to be chosen became this : “implement any features that you can implement, as long as you eventually implement some system calls in webassembly”.

The bottom requirement became :

1. To Implement a bare kernel (however far the implementer can)
2. To Embed a wasm runtime
3. To Export at least one syscall in WASI notation

**Functional requirements :**

Under this segment, the minimal kernel stands for a kernel that has not integrated the wasm runtime in its code. The term extended kernel describes the result of integrating the minimal kernel with the wasm runtime.

1. The minimal kernel should provide a terminal user interface as its standard output and the keyboard as the standard input.
2. The minimal kernel should have a page grained memory allocation system
3. The minimal kernel should have a byte-grained memory allocation for both kernel and user space heaps
4. The minimal kernel should have a memory management unit that runs in the Riscv Supervisor mode
5. The minimal kernel should be able to handle both external and internal interrupts with the help of the CLINT and PLIC
6. The minimal kernel should support user processes
7. The minimal kernel should export both system calls falling under FileSystem access, memory allocation and process handling.
8. The minimal kernel should provide persistent storage by providing a filesystem that runs atop a hard-disk.
9. The extended kernel should export its system calls using the WASI syscall prototypes
10. The extended kernel should be able to load wasm containers and execute them as distinct processes.
11. The extended kernel should provide a simple capability based interface for inspecting the capabilities exposed to a user process.

## System Design

System design involved sketching out the possible high-level implementations. It ranged from drawing block diagrams, dataflow diagrams and pseudo-code flowcharts.

The tools used were mermaid.js, excalidraw and mdbook.

The Designs made have been included in the Implementation section.

Just like the system analysis, a person can only design what they know. The designs became clearer with time as the implementer half-implemented the low hanging components. With time the designs became clearer, almost days before the final presentation of the project.

For example, the UART design might seem independent at first, but it you reach the section of trap handling, you realize that the PLIC and the UART are quite intertwined. Their memories are also getting handled by the MMU, a component that you only realize is needed later on.

The above problem introduced the problem of re-drawing components. Today the designer may take an hour designing how the UART works. Two weeks later the designer trashes the previous image because it is incompatible with the PLIC. To solve this problem, the designer resorted to using mermaid.js, a tool that uses code to draw designs. Excalidraw made it easier to modify the images.

### The Mini-Kernel

Below is the overview of the components that interact with the Kernel.

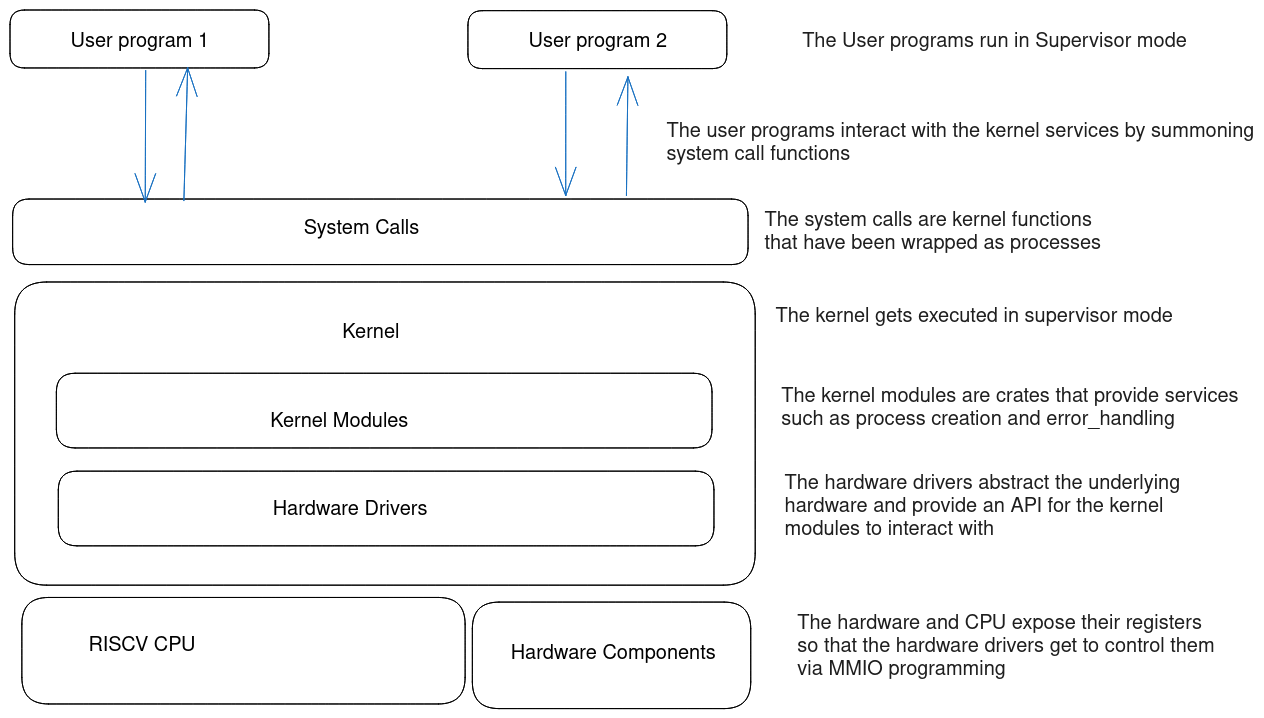


Figure 6 The kernels’ environment

### The Hardware Components

The hardware components in the system include the following :

1. The hard-disk
2. The RAM (Random Access Memory)
3. The CPU (4 cores)
4. The MMIO memory region
5. The UART converter
6. The Core Local Interruptor (CLINT)
7. The PLIC (Platform Level Controller)
8. The Console
9. The Keyboard

The functions done by hardware drivers mostly fall under three categories : The functions might be controlling the behaviour of the hardware by changing the status of different registers OR the driver might just be abstracting the hardware

Not every hardware component has a a driver, this is because Rust code can directly access the raw memory addresses of the component registers without the need of setting up communication configurations. For example, the CLINT does not need a driver.

### The UART

#### Overview

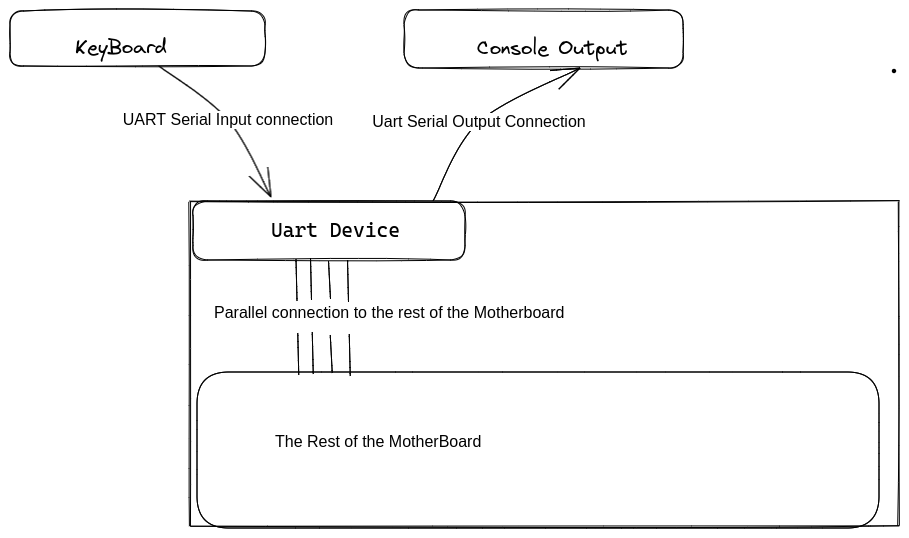


Figure 7Interaction of the UART with other components

#### Interaction with Keyboard – high

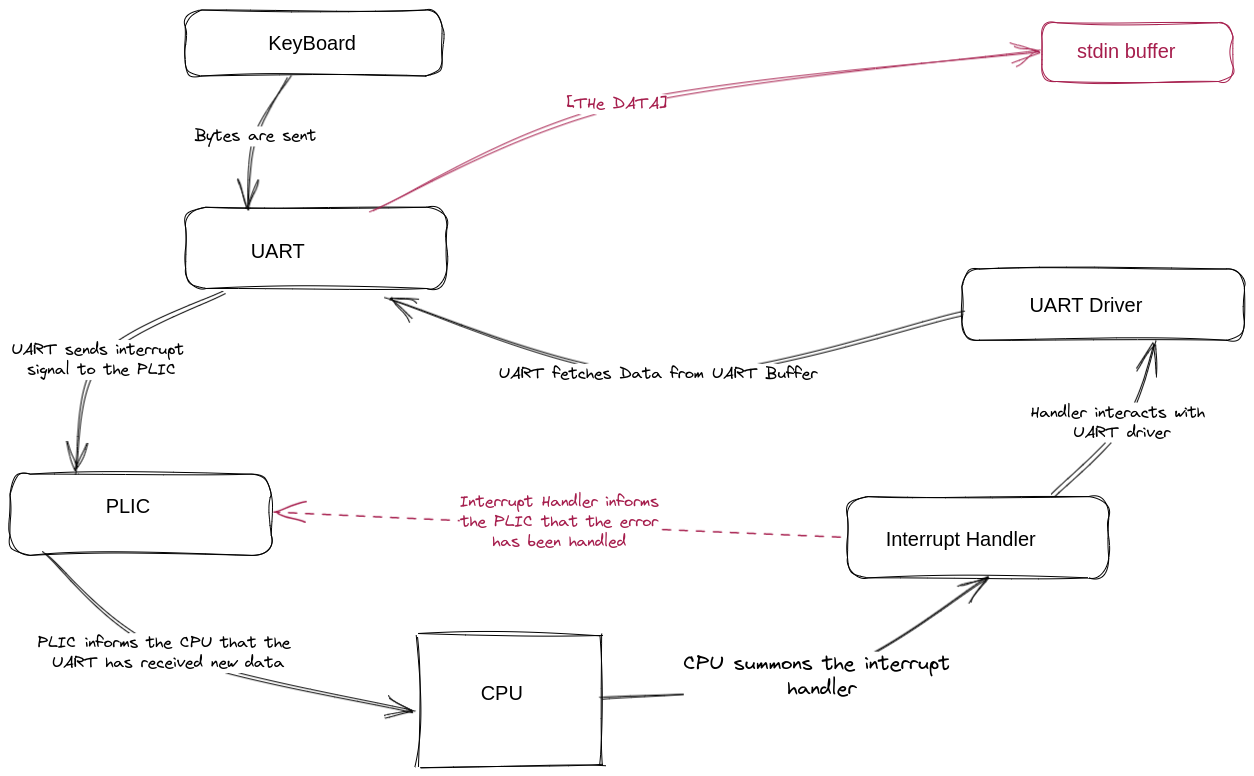
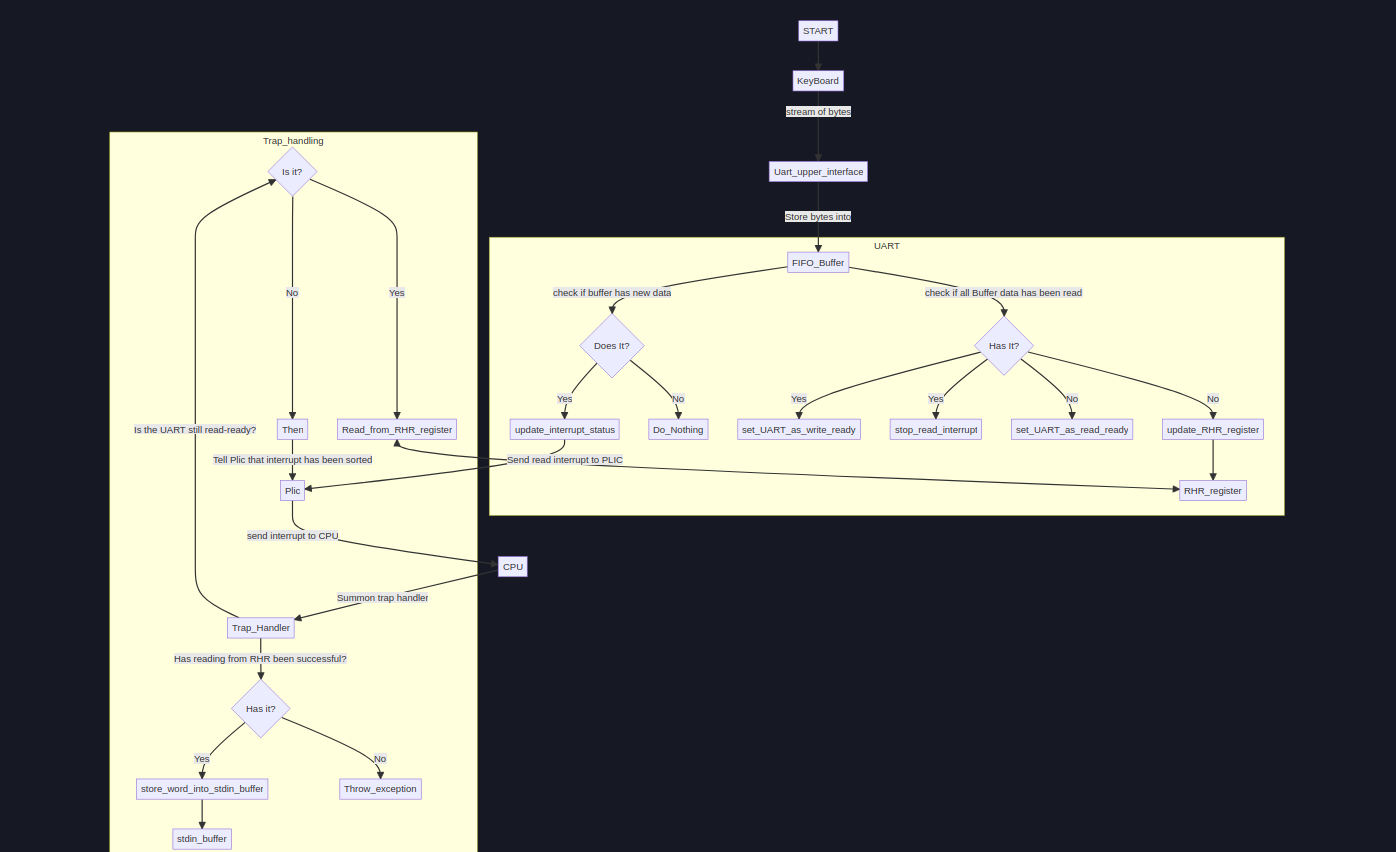


Figure 8Keyboard interaction with the UART

#### Interaction with keyboard - low



### Memory Design

#### Overview of memories being handled:

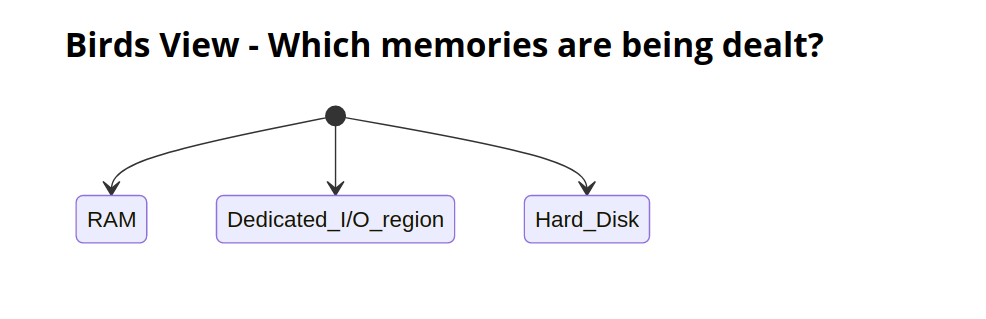


Figure 9 The memories being abstracted

Figure 10 How the Ram gets abstracted

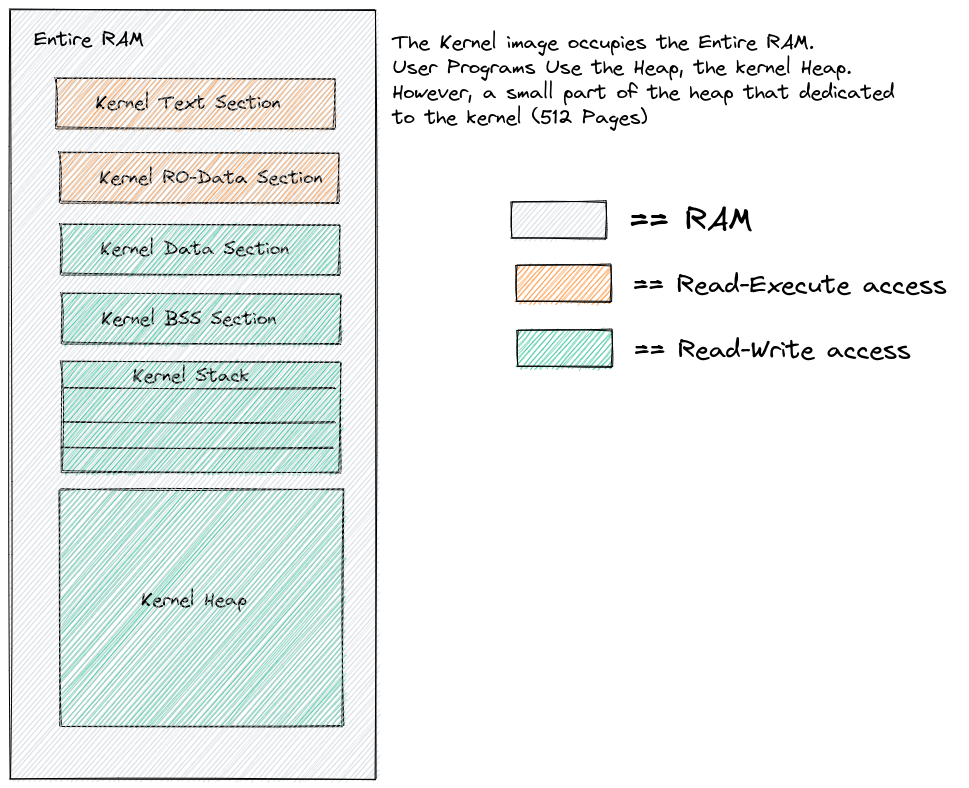


Figure 11 How the Heap gets abstracted

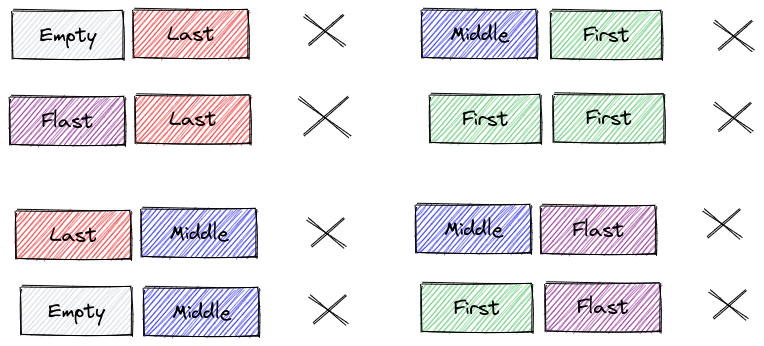
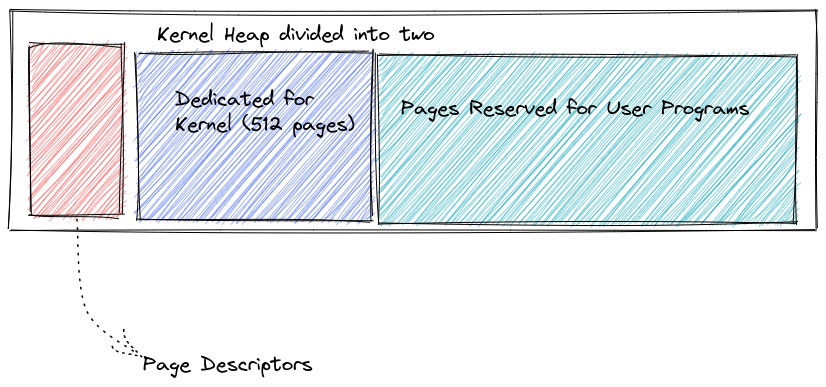
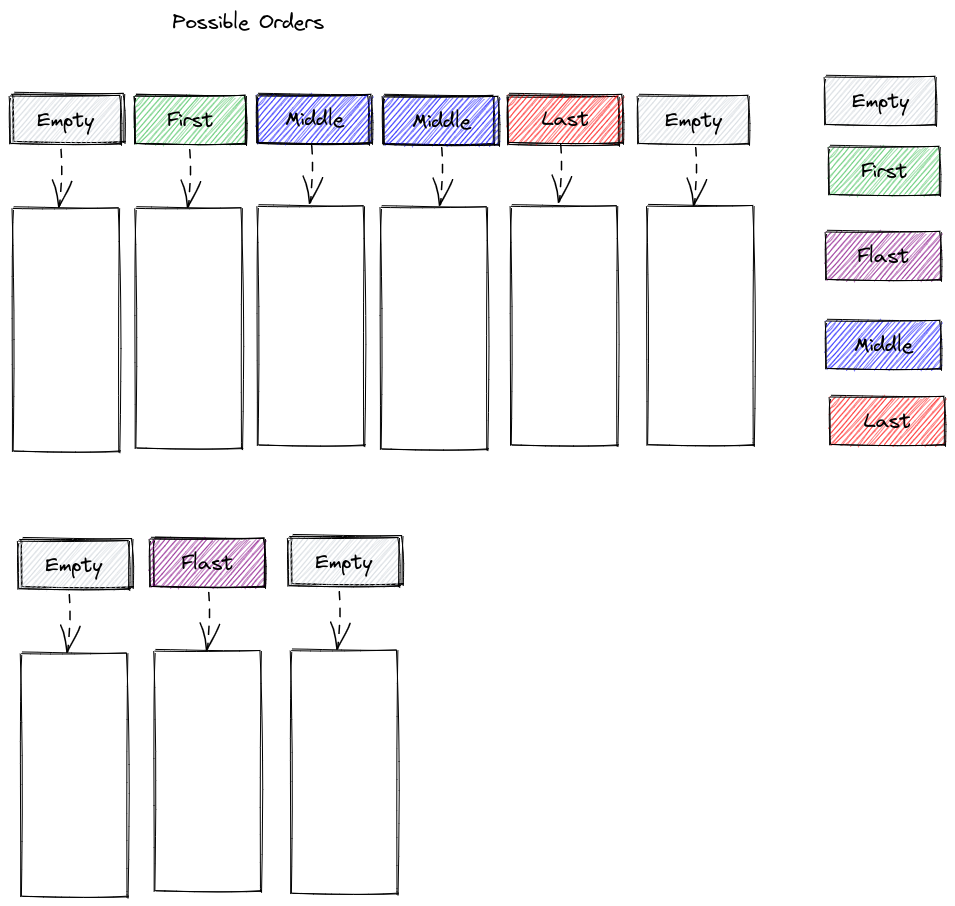
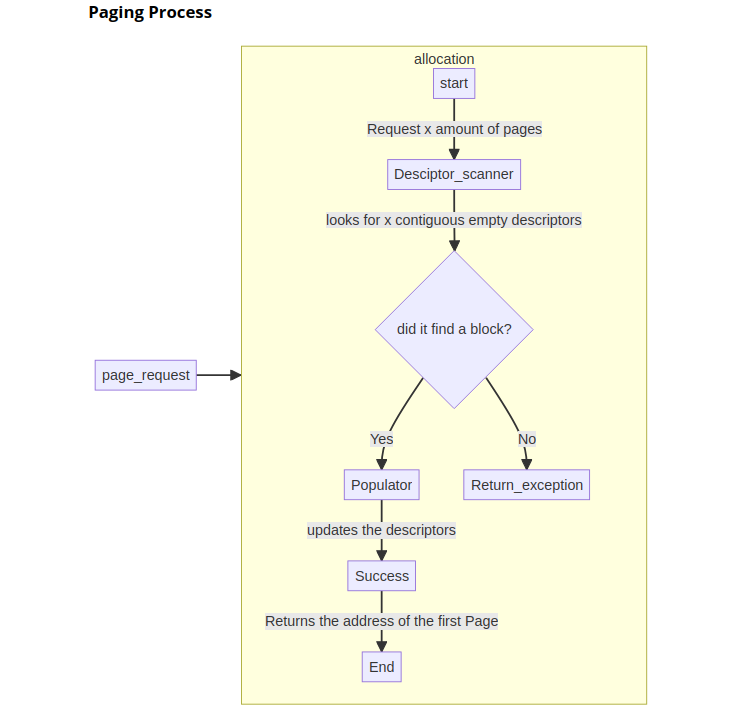


Figure 12 Descriptor Ordering

Figure 13 Abstraction of heap as pages and descriptors

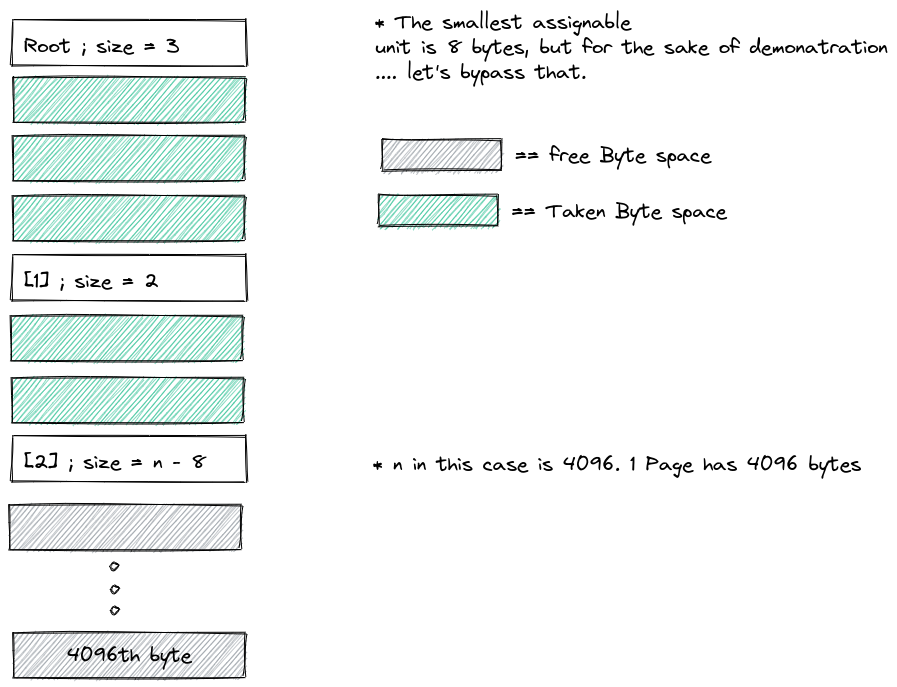


#### Page allocation



### Byte Allocation

Byte allocation happens within the Page. A linked List is used to keep track of which bytes have been allocated



### Virtual Memory Management

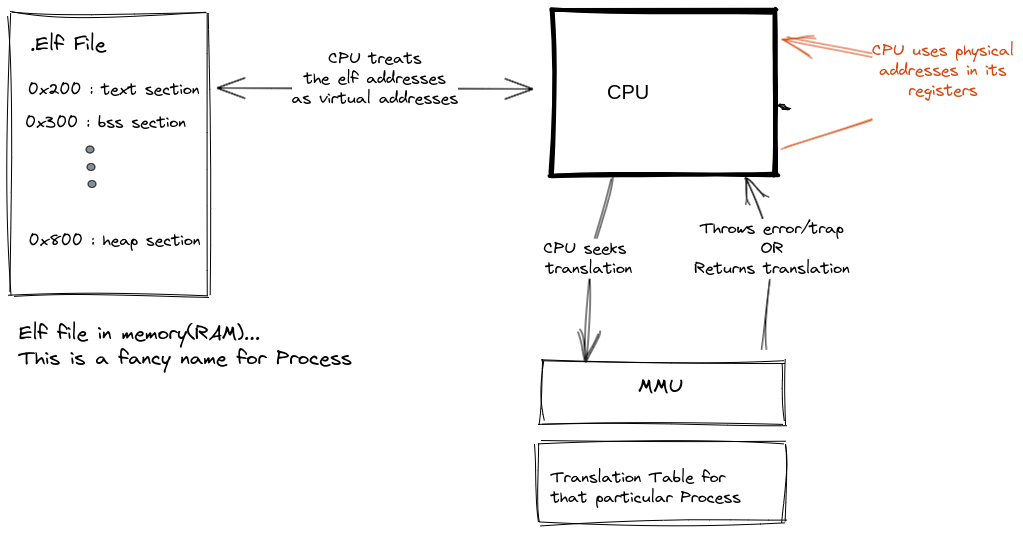


Figure 14 The translation of virtual addresses

# System Implementation

This section discusses how different components were implemented.

## System setup

The kernel was written for Qemu-virt riscv machine. This compelled the developer to set up a cross-compiling toolchain. The Riscv-GNU toolchain[12] was initially used but the Rust Default toolchain proved more user friendly in the end. The Riscv GNU had installation difficulties at first, this was followed by configuration subtle tweaks and bugs. Moreover, it required the use of Makefiles which were very cumbersome.

The Rust toolchain however has an easier time installing and configuring. There is no need to write Makefiles, cargo automatically takes care of dependency and compilation dependencies. All the chain tools (linker, compiler, Assembler, debugger, binary file inspectors) are installed by default when you install the language compiler. In one command.

The Rust toolchain also supports RISCV assembly, you can write asssembly code within rust code. You can link assembly files and the toolchain will take care of combining them into object files.

The rust toolchain allows one to download compiler backends called targets. Like for the case of this project, the target installed was riscv64gc-unknown-none-elf .

The process of linking the different object files and determining the layout of the final ELF file was defined via a linker script. Configuration on the dependency management and compilation was done via cargo.

The toolchain setup was followed by making a binary file that could run on bare metal, without standard Library support. This involved setting up the right nightly compiler features and defining the error handling personality of the executable.

## The Bootloader

A typical bootloader essentially does the following operations:

1. Find the memory address of the Kernel program by searching through the external memory devices that have been plugged into the Motherboard.
2. Loads the Kernel image onto the RAM. Note that it only loads the required sections, it might not load the entire image.
3. Prepare the values found in the CPU registers to suit the execution of the kernel. For example making the Stack pointer to point to the stack of the kernel.
4. Transfering control to the kernel. This is done by making the program counter point to the entry\_point of the kernel

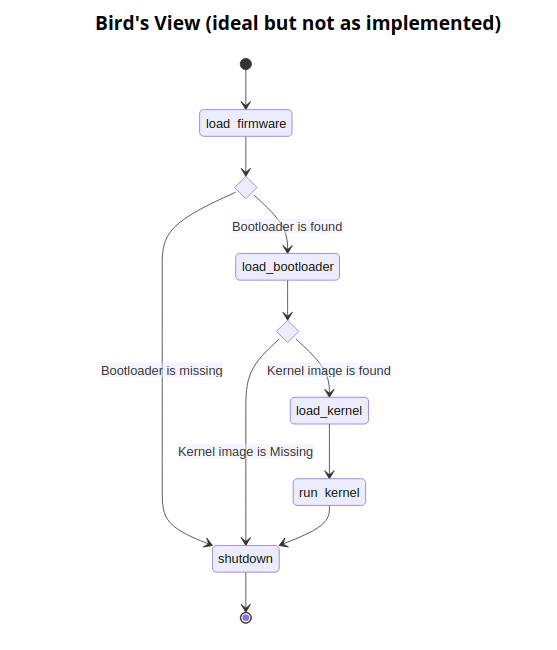
The Bootloader in this project is much simpler. It does not have to look for the kernel image in a secondary memory like a hard-disk, instead, the Kernel and the bootloader are one program.

I the case of this project, the boot program is the one that is responsible for :

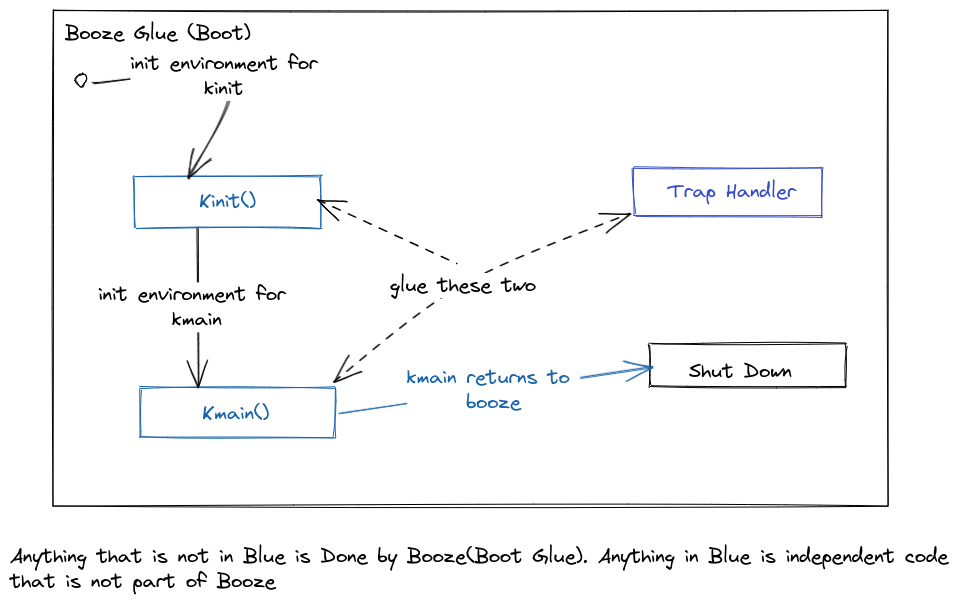
1. Setting up the environment for the kernel to run in machine mode
2. Transferring the control to kinit() (kernel running in machine mode)
3. Designating a specific place where the CPU will jump to if a trap occurred when kernel is in either machine mode or supervisor mode.
4. Initiating an environment for the kernel to run in supervisor mode.

The boot code is the one that contains the \_start function. \_start function is the entry point for the whole kernel image. So it is the first place the CPU points to in the RAM after the Qemu firmware has done its thing.

Here is a Bird’s view of the general boot process :



Instead of the boot code being a program that gets called once(as seen above), it becomes the glue code that gets called whenever kernel goes out of scope. It encompasses the entire kernel. It acts as glue code as seen below :



Below is a clearer demonstration how the boot code has been implemented :

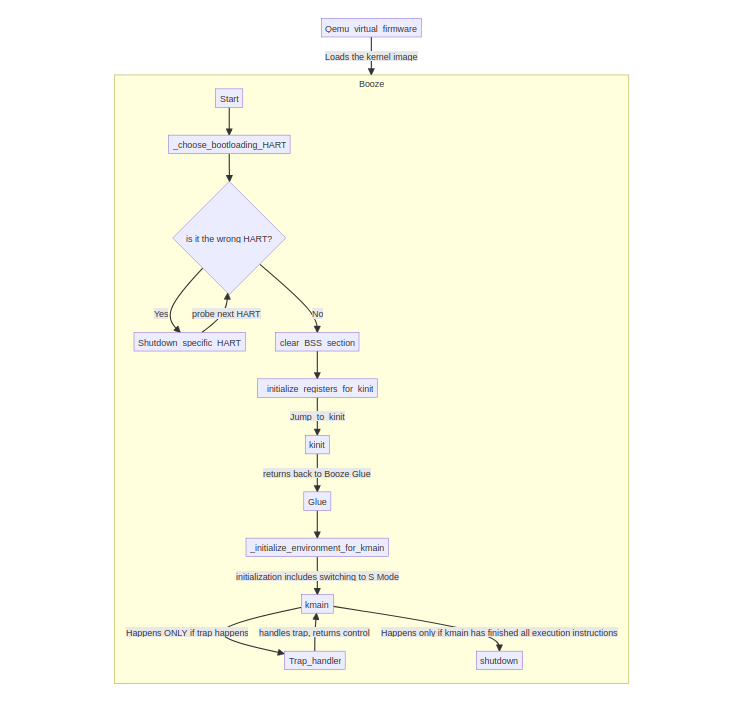


Figure 15 Rough Flow of the Boot code

## The UART

The UART device is a hardware device that stands in between a parallel connection and a serial connection. It converts a serial signal to a parallel signal and vice-versa.  
 It stands between the motherboard and the peripheral devices that use serial connections ; eg mouse, keyboard and console output.  
The UART driver is the software that controls and interacts with the physical UART device.

In the case of Qemu, the console output and the keyboard input use the same UART device. This is because the transmit\_out channel is connected to the console output AND the receive channel is connected to the keyboard.

The UART emulated in Qemu is the NS16550A UART chipset. We control the UART using MMIO programming. The Base address of the UARTs begins at 0x1000\_0000 and each UART device is given an offset of 0x100 (256 bytes)

The UART has 8 physical registers that can be interpreted as 12 logical registers... this is because some of the physical registers can be used differently under different contexts. For example The Buffer Register can be used as an input register when the UART is idle, but when the UART is not idle, the same register will be treated as an output register.

[**Why are we not using the USB driver/protocol?**](http://localhost:3000/general_theory_on_communication.html#why-are-we-not-using-the-usb-driverprotocol)

The USB also does the conversion of parallel signals to serial signals. The USB has higher transfer speeds than the UART connection. The USB can do 20 Gbps while the UART does around 1 Mbps!  
However, the project uses the UART because of two reasons :

Its simplicity in configuration. The UART is easier to configure when compared to the USB. This is a learning project, it is expensive to delve into the complex nature of USBs.

It uses less power than the USB. The project is targeting embedded systems; the less power the machine consumes, the better.

There is a trade-off the speed of the USB for the simplicity and power-efficiency of the UART.

**Implementation of the UART**

Implementing the UART driver involved initializing communication configurations and making the UART reads and writes to be interrupt driven.

## Memory Initialization

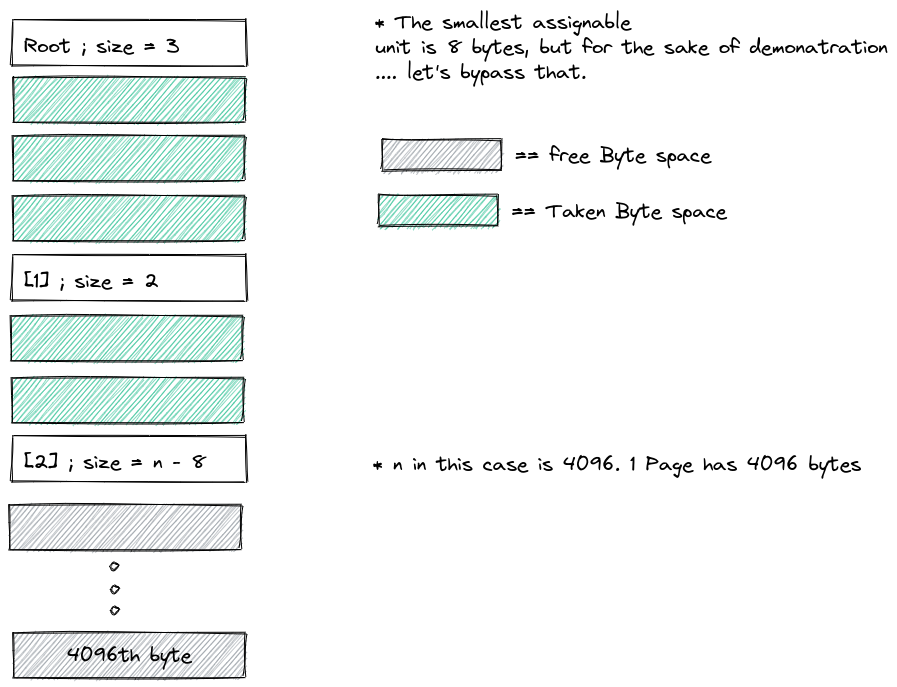
The memory initialization module is a module that runs in machine mode, its main purpose being to dermarcate the heap section of the kernel into usable descriptors and corresponding pages. It abstracts the physical memory for other programs such as the Page allocator, byte allocator and Virtual Memory Management unit.

## Page Management

The page manager takes advantage of the abstractions generated by the memory initializer. It is responsible for allocating and deallocating contiguous pages. Each page is 4096 bytes large.

## Byte Management

The byte allocation system allocates space per-byte. To keep track of which bytes have been allocated within a single page, the byte allocator uses a linked list as demonstrated below :



The Byte allocator was then connected to the rust compiler’s global allocator.

## The Memory Management Unit

The instructions in the Elf files of programs typically reference virtual addresses. The CPU cannot execute an instruction that has a virtual address. For this reason, everytime the CPU encounters a virtual address in an instruction, the CPU uses the MMU circuitry to translate the virtual address into a physical address.

The MMU has been programmed to enforce access rights to certain physical memory addresses, such that a translation will fail if an access right is being violated. If the translation fails, a page fault is thrown by the MMU and the interrupt handler handles it. In this case, the MMU acts as a memory protector, ensuring translations only happen when all access rights are adhered to.

In Machine Mode, RISCV provides a mechanism of protecting memory called Physical Memory Protection(PMP). But PMP does not scale well.

Each Process gets assigned its own translation tables, therefore, each process gets its own address space