A MASTER THESIS ON

# Rust in the Linux Kernel: Analyzing Rust Implementations of Device Drivers

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

Diplom-Ingenieur
(Equivalent to Master of Science)

in

Embedded Systems (066 504)

by

**Fabian Garber, BSc**

01425023

**Supervisor:**
Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner

Vienna, Austria
1st of June 2025

# Abstract

This thesis investigates the comparative performance and energy consumption of Rust and C-based virtual General Purpose Input Output (GPIO) drivers in the Linux kernel. For the experimental platform a Raspberry Pi 4 Model B is used. The research aims to evaluate the implications of integrating Rust, a language known for its memory safety features, into the Linux kernel development.

The findings reveal that Rust-based drivers, while offering enhanced safety and reliability, have longer execution times and higher energy consumption compared to their C counterparts. Specifically, Rust modules demonstrated an approximate 8% increase in execution time for the basic variant, 11% with the inclusion of a wait queue, and 14.3% with both a wait queue and a spinlock. Energy consumption measurements showed that Rust modules consumed 46.15% more energy in the basic variant, 27.68% more with a wait queue, and 33.86% more with both a wait queue and a spinlock. Although the Rust based module performed slower than the C equivalent, Rust might still be a viable alternative to C, because of easier and quicker debugging compared to C, which in the long run leads to safer and more stable software.

# Kurzfassung

In dieser Arbeit wird die Leistung und der Energieverbrauch von Rust- und C-basierten virtuellen GPIO-Treibern im Linux-Kernel untersucht. Als Versuchsplattform wird ein Raspberry Pi 4 Model B verwendet. Die Forschung zielt darauf ab, die Auswirkungen der Integration von Rust, einer Sprache, die für ihre Speichersicherheitsfunktionen bekannt ist, in der Linux-Kernelentwicklung zu bewerten.

Die Ergebnisse zeigen, dass Rust-basierte Treiber zwar mehr Sicherheit und Zuverlässigkeit bieten, aber im Vergleich zu ihren C-Pendants längere Laufzeiten und einen höheren Energieverbrauch haben. Konkret zeigte sich, dass das Rust-Modul eine um 8% längere Ausführungszeit für die Basisvariante, um 11% längere Ausführungszeit mit der Verwendung einer Warteschlange und eine um 14,3% längere Ausführungszeit bei der Verwendung einer Warteschlange und einem Spinlock, aufweist. Messungen des Energieverbrauchs ergaben, dass Rust-Module in der Basisvariante 46,15% mehr Energie verbrauchten, 27,68% mehr mit einer Warteschlange und 33,86% mehr mit einer Warteschlange und einem Spinlock. Obwohl das Rust-basierte Modul langsamer war als das C-Äquivalent, könnte Rust immer noch eine brauchbare Alternative zu C sein, da das Debugging im Vergleich zu C einfacher und schneller ist, was auf lange Sicht zu sicherer und stabiler Software führt.

## *Erklärung*

*Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.*

*Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.*

## Copyright Statement

I, Fabian Garber, BSc, hereby declare that this thesis is my own original work and, to the best of my knowledge and belief, it does not:

- Breach copyright or other intellectual property rights of a third party.
- Contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
- Contain material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.
- Contain substantial portions of third party copyright material, including but not limited to charts, diagrams, graphs, photographs or maps, or in instances where it does, I have obtained permission to use such material and allow it to be made accessible worldwide via the Internet.

Signature: ______________________

Vienna, Austria, 1st of June 2025 Fabian Garber, BSc

# Danksagung

Ich möchte meinem Betreuer, Wilfried Steiner, herzlich für seine Unterstützung und fachkundige Begleitung während meiner Diplomarbeit danken. Seine Rückmeldungen und wertvollen Anregungen haben maßgeblich zum Gelingen dieser Arbeit beigetragen.

Ein besonderer Dank gilt auch meinen Freunden Viktor und Matthias für ihre kontinuierliche Unterstützung, Motivation und die vielen inspirierenden Diskussionen, die mich auf meinem Weg begleitet haben.

Außerdem möchte ich meiner Freundin Siri danken, die mich stets unterstützt und immer wieder aufs Neue motiviert hat.

Den größten Dank schulde ich jedoch meinen Eltern Evelin und Walter. Ohne ihr Vertrauen und ihre stetige Unterstützung wäre das alles nicht möglich gewesen. Danke!

# Contents

# List of Tables

# List of Figures

# Listings

# Acronyms

**AI** Artificial Intelligence. 17

**API** Application Programming Interface. 11, 15, 19, 33, 42

**CPU** Central Processing Unit. 1, 10

**CTSS** Compatible Time-Sharing System. 6

**DMA** direct memory access. 42

**DUT** Device Under Test. 30, 36

**EL** Exception Level. 7

**EOL** end-of-life. 19

**FFI** foreign function interface. 13

**FIFO** First In – First Out. 9

**FUSE** Filesystem in Userspace. 10

**GPIO** General Purpose Input Output. iii, iv, xi, 3, 18, 29, 30, 33, 35, 37, 41

**GUI** Graphical User Interface. 5, 6

**I/O** Input/Output. 1, 5, 7, 8, 15, 17, 20

**ioctl** input/output control. 30, 31

**IoT** Internet of Things. 1, 33

**IPC** Inter-Process Communication. 7, 9

**IRQ** Interrupt Request. 12

**KVM** Kernel-based Virtual Machine. 7

**LKM** Linux Kernel Module. 10

**LSB** Least Significant Bits. 11

**LTS** Long Term Support. 17

| | |
|---|---|
| **MIT** | Massachusetts Institute of Technology. 6 |
| **MSB** | Most Significant Bits. 11 |
| **MSLOC** | millions of source lines of code. 6 |
| **mutex** | mutual exclusion. 33 |
| **NVMe** | Non-Volatile Memory Express. xiii, 13, 14 |
| **OS** | Operating System. 5, 6, 9, 10, 12 |
| **OSes** | Operating Systems. 5, 6, 7, 12 |
| **PAC** | pointer authentication code. 25, 26 |
| **PCI** | Peripheral Component Interconnect. 12, 13 |
| **RFL** | Rust for Linux. 11, 12, 13 |
| **sp** | stack pointer. 25, 26 |
| **SSD** | Solid State Disk. 13 |
| **SSH** | Secure Shell. 19 |
| **TTL** | Transistor-Transistor Logic. xiii, 18, 19, 21 |
| **UDP** | User Datagram Protocol. 13, 14 |
| **USB** | Universal Serial Bus. xiii, 18, 20, 21 |
| **VAS** | Virtual Address Space. 7 |
| **VFS** | Virtual Filesystem Switch. 7 |
| **VM** | Virtual Machine. 17 |

# Chapter 1

# Introduction

At the heart of every operating system lies the kernel. The kernel is a fundamental part of the operating system. It manages the systems' resources such as the Central Processing Unit (CPU), memory allocation, and Input/Output (I/O) devices to ensure that they are used efficiently and fairly among all running processes. In addition, the kernel handles the management of the creation, scheduling, and termination of those processes. It is important that each process has the necessary CPU time and resources to guarantee a proper functioning. Also part of these responsibilities are tasks that manage devices (called "device drivers"), because they act as an interface between the hardware and software parts of a computer. Furthermore, the kernel handles security and access control to ensure that there is no unauthorized access to system resources [4].

One of the most used kernels is the Linux kernel. The Linux kernel, developed in 1991 by Linus Torvalds, has become increasingly important in the last decade. Most supercomputers, servers, and mobile phone operating systems rely on the Linux kernel [5, 6]. Due to the sharp increase of embedded and Internet of Things (IoT) devices, more devices are running Linux than ever before. Since the development of the Linux kernel, there was (mainly) only one programming language present: C. This changed with the release of the 6.1 Kernel in October 2022. Although the kernel fully supported Rust in October 2022, the first Linux drivers written in Rust were released in Linux 6.8 in December 2023.

The reason why the support for Rust was implemented was because of concerns about the Linux kernel's maintainability and scalability, due to concurrency and memory bugs could never be completely eliminated [7,8]. Rust promises memory safety and consistency with C [9]. This memory safety is achieved via the principle of 'Ownership'. Some programming languages such as Java or Python have garbage collection that periodically check for no-longer-used memory while the program is running. In languages such as C, memory gets manually allocated and deallocated. Rust on the other hand manages its memory via a certain system of ownership with a special set of rules that the compiler checks [10]. In case the rules are violated, the program won't compile. In Rust, each value, which is a piece of data stored in memory with a specific type, is owned by a single scope where it is defined. The transfer of ownership of a value to a function, which is a subroutine or a block of reusable code, involves passing that value from one part of the code (the caller) to another (the called function), resulting in the caller losing access to the value after the call. This concept of ownership enables static analysis of the lifetime of each heap-allocated object and explicit deallocation. When its lifetime ends, the value is automatically dropped, and its storage is reclaimed [11].

Rust has shown to be safer than C, C++, Java, Go, and Python, while still being among the top from a performance point of view [12]. In Android, the share of memory vulnerabilities out of all vulnerabilities dropped from 76% in 2019 to 35% in 2022 after the switch to Rust [13]. This should improve the quality of device drivers. Drivers are specialized kernel modules that interface directly with hardware components, providing low-level control functions. They are able to communicate through the computer bus or communication subsystem with the devices. Every device type has its own device driver. Furthermore device drivers on Windows don't work on Linux, which means they are also operating system specific [14].

Device drivers serve as a translator between the hardware and the operating system. This allows the operating system to access and control hardware without detailed knowledge about the hardware. There are three different types of device drivers. Character device drivers, which transmit data character by character. This is used in keyboards or serial ports. Block device drivers manage, as the name suggests, data in blocks. Examples are hard drives or USB storage devices and the final type are network device drivers. Network device drivers deal with network interfaces which handle transmission and receiving of data packets over a network [15]. As mentioned above, Rust was allowed as a second language for the Linux kernel, to enhance memory safety and security. This is because Rust prevents common programming errors such as buffer overflows and null pointer dereferences. Those are the main sources of security vulnerabilities in the kernel. Another advantage of Rust is that the writing of safe concurrent code is facilitated. The kernel often has to deal with multiple processes and threads simultaneously. Consequently, the addition of Rust might help to improve the overall quality of the kernel [16].

The (initial) focus of Rust in the kernel is on writing new drivers [16]. Drivers are a good place to start gradually introducing Rust to the kernel, because device drivers are isolated pieces of code. So they can be tested and and integrated without affecting the core functionality of the kernel [17].

## 1.1 Research objective

The aim of this thesis is to address the following research question:

What are the differences in (i) performance, (ii) energy consumption, and (iii) safety between a Rust-based virtual General Purpose Input Output (GPIO) driver and an equivalent C-based driver in the Linux kernel?

We address this research question by the design of an evaluation platform using a Raspberry Pi 4 Model B.

The thesis is divided in different chapters.

Chapter 2 examines the state of the art and how it developed. Chapter 3 addresses the system setup, which contains the setup of the experimental environment and the test setup, such as target platform, kernel customization as well as the used tools. We discuss the practical aspects of this thesis including writing kernel modules in C, in Rust, and test cases in chapter 4. Chapter 5 presents the results of the experiments. Finally chapter 6 gives a conclusion that summarizes the findings and discusses their implications.

# Chapter 2

# State of the art

The evolution of Operating Systems (OSes) has been an important part in the development of modern computing [18]. From the early days of mechanical switches and punch cards to the creation of accessible Graphical User Interfaces (GUIs). This chapter discusses the historical progression of OSes as well as the different kernel architectures and their advantages and disadvantages. Furthermore, the integration of the Rust programming language in the Linux kernel is discussed. Finally some Rust-based driver projects are presented.

## 2.1 Operating System

The first computers were not using an Operating System (OS), the user had to enter a program part by part in machine code. At the beginning mechanical switches were used and later on stacks of punch cards helped to program the computer.

The beginning of the OS dates back to the 1950s. The goal was to make more efficient use of the computer resources, which were expensive at that time. Those systems were called 'batch processing systems', because they allowed to run one job. Programs and data were submitted in batches, therefore the name.

The next evolution step was the 'multi-batch' system. This allowed computers to run various different jobs at once. This allowed for the usage of the processor and I/O devices at the same time. Those jobs were usually created with punch cards or utilizing computer tape and a small error would cause the program to fail and a restart would be necessary. Due to those reason the software development process was slow and tedious.

To deal with that, operating system designers developed the concept of 'multiprogramming'. This allowed for several jobs to be in the main memory at the same time. In addition the concept of interrupts was added. Interrupts enable a unit of the system to get the attention of another unit. When the system gets an interrupt it saves the state of the interrupted part of the program, then it services the interrupt and afterwards it restores the saved state.

In the 1960s typewriter-like terminals were developed that allowed for the use of a highly-interactive environment, which enabled quick responses to user requests. This was possible due to the construction of time-sharing systems. One of those was the Massachusetts Institute of Technology (MIT) Compatible Time-Sharing System (CTSS). CTSS was able to ensure high usage of expensive computer resources by running a conventional batch stream and provided the programmer who edited or debugged programs with a quick response [19].

With the development of the first GUI based operating systems in the 1980s, computers were more intuitive and user friendly, which is the standard even until today [20].

## 2.2 Different kernel designs

Modern OSes are complex and big software systems that serve as a fundamental interface between computer hardware and user applications. Due to their wide scope it is difficult to design them, both in security and functionality. The kernel is the most important part of an OS. It is responsible for all the basic system service and it is constructed in a layered fashion. Starting from the process management to the interfaces up to the rest of the OS, such as libraries. Stacked on top of all this are the applications.

One of the main differences between the OS and the user-mode software is that user-mode software can be removed and changed from the user. Parts that reside in the OS, such as a clock interrupt handler can't be changed by the user. But the distinction is not always as clear as in the mentioned example, so it is not always possible to draw a clear boundary. [21, 22].

There have been different approaches for the design of OSes. Microkernel-based OSes, monolithic-based OSes and hybrid-based OSes [23, 24].

In the next part the differences, advantages and disadvantages of the different designs will be discussed.

### 2.2.1 Monolithic kernel

Monolithic kernels, contrary to microkernels, contain most services in the part of the system that executes in privileged mode. With added complexity comes an explosive growth in millions of source lines of code (MSLOC). Code that can be executed in privileged mode has the potential to bypass security and can therefore lead to insecurities [25].

In computer science there are hierarchical protection domains, also called ring levels. They provide different levels of access to resources within the architecture of the computer system. Different architectures provide a different amount of ring levels. The AArch32 microprocessor family has up to seven modes and the used AArch64 microprocessor family supports four so called Exception Levels (ELs).

They are arranged from EL0 (least privileged) to EL3 (most privileged). In systems without a hypervisor, the kernel space has the highest privileges because it directly manages the hardware. Applications are the least privileged and have therefore also the most access restrictions to resources. Those access rings exist to improve security and to prevent applications from misusing resources. There are special mechanisms in place so an outer ring can also access resources from an inner ring and allow for context switches [1]. Over time OSes have become more complex. Features such as dynamic module loading, multithreading and kernel locks were added [23].

In Figure 2.1 the Linux kernel space, a monolithic kernel design, is shown. A monolithic kernel architecture is an architecture in which all kernel components reside and share the kernel Virtual Address Space (VAS). The Linux kernel space consists of the following components [1]:

- Core kernel: Responsible for essential tasks such as managing processes.
- Memory Management: Manages the system memory.
- Virtual Filesystem Switch (VFS): Provides ways for the kernel to interact with different filesystems e.g. ext4, FAT32.
- Block I/O: Handles input and output operations for block devices like hard drives.
- Network protocol stack: Implements necessary protocols to allow communication over a network.
- Inter-Process Communication (IPC): Allows for different programs to communicate with each other.
- Sound support: Handles everything audio related.
- Virtualization support: Includes the virtualization technology called Kernel-based Virtual Machine (KVM). Allows to run multiple virtual machines on a single physical machine.

Furthermore, there is architecture specific code (arch-specific), Kernel initialization (init), security frameworks (security) as well as various device drivers [1].

As seen in Figure 2.2 monolithic kernels contain VFS, IPC, Scheduler, Virtual Memory as well as Device Drivers and Dispatcher.

This means the basic system services such as memory management, I/O communication, interrupt handling and the file system are all managed in privileged mode. Having everything in the kernel space also leads to some disadvantages such as large kernel size, lack of extensibility as well as bad maintainability. A larger code size leads to longer compilation times during bug fixing and also to a more unreliable system [26].

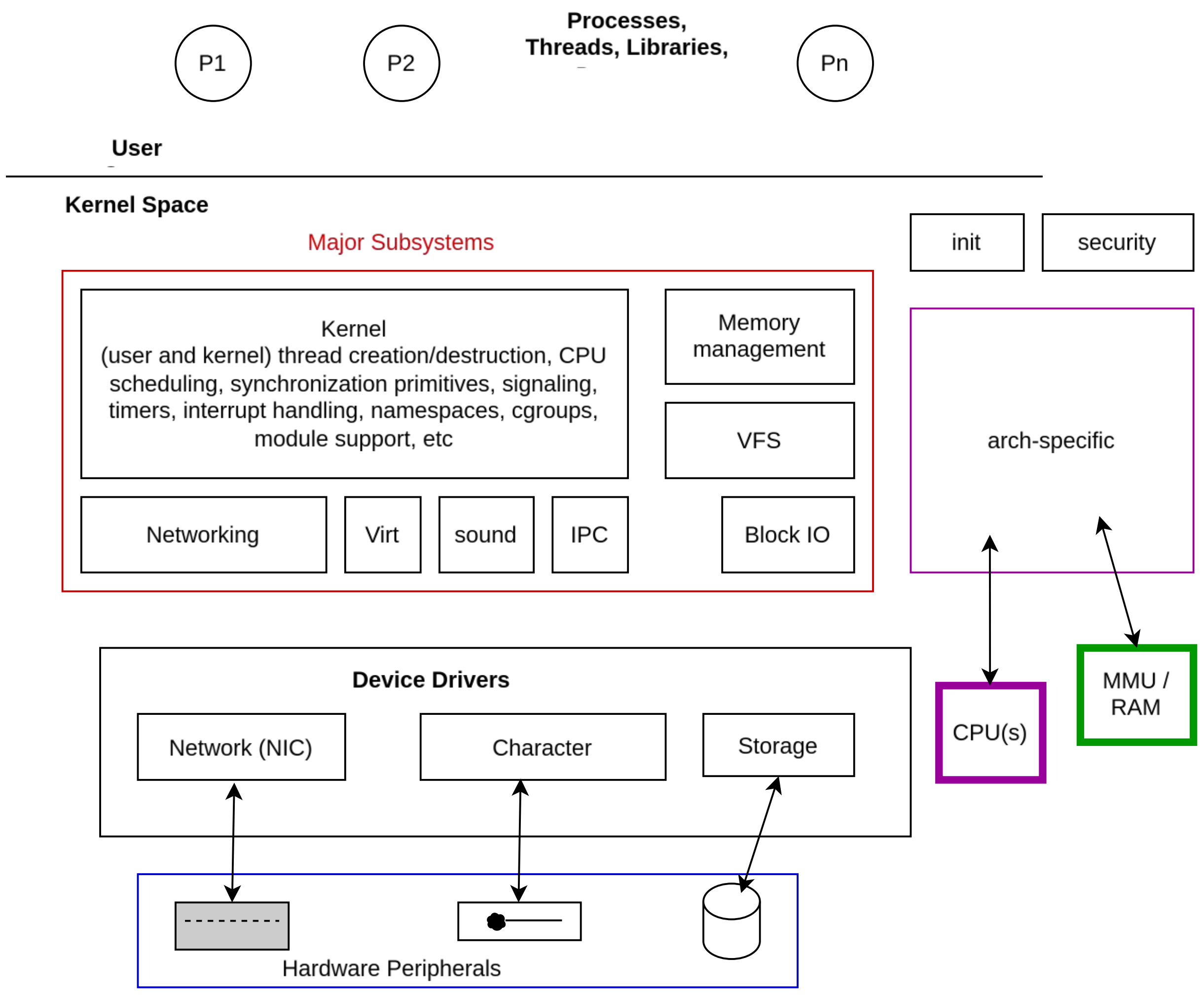
P1
P2
Processes, Threads, Libraries,
Pn
User
Kernel Space
Major Subsystems
init
security
Kernel
(user and kernel) thread creation/destruction, CPU scheduling, synchronization primitives, signaling, timers, interrupt handling, namespaces, cgroups, module support, etc
Memory management
VFS
arch-specific
Networking
Virt
sound
IPC
Block IO
Device Drivers
Network (NIC)
Character
Storage
CPU(s)
MMU / RAM
Hardware Peripherals

Figure 2.1: Linux kernel space - major subsystems and block [1]

### 2.2.2 Microkernel

To handle the disadvantages mentioned in the previous section the idea of microkernels appeared at the end of the 1980's. The idea was to reduce the functionality of the kernel to basic process communication and I/O control. This allows the other system services to reside in user space and be used as normal processes [21]. A microkernel consists of core services, such as threads, signals, message passing, synchronization, scheduling as well as timer services. To add more functionality cooperative processes are implemented. Cooperative processes are used to extend functionality. These processes act as servers, handling requests from clients in a true client/server model. This approach moves services like device drivers and file systems out of the kernel and into user space, keeping the kernel minimal and focused on core tasks like process communication and basic hardware management. This design enhances modularity and system stability [24].

In comparison to monolithic kernels that use signals and sockets for inter process communication, microkernels use message queues. Message queues are built as a First In – First Out (FIFO) queue. This means all the incoming messages are stored in a message queue and each message queue takes the responsibility for various kinds of messages [21].

Microkernels remove the functions, such as device drivers, protocol stacks and file systems from the kernel [27]. The OS kernel is separated into distinct parts that are isolated by memory protection barriers [23, 28].

In Figure 2.2 the structure of a monolithic and a microkernel are shown.

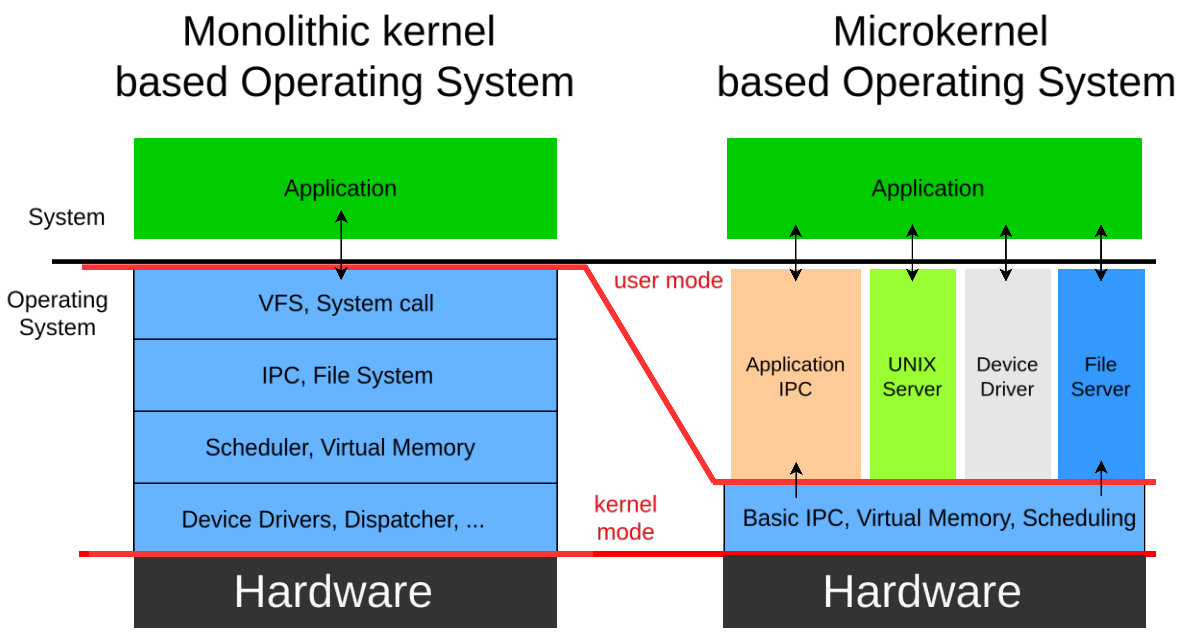
Monolithic kernel based Operating System
Microkernel based Operating System
System
Operating System
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
user mode
kernel mode
Application
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware

Figure 2.2: Structure of monolithic and microkernel-based operating systems

Utilizing a smaller size kernel has the goal to enhance reliability by providing only basic process communication. A disadvantage of this separation is the created overhead that leads to limited performance, compared to a monolithic approach [23].

### 2.2.3 Hybrid kernel

To mitigate the aforementioned degradation of performance, a hybridization was investigated. The hybrid kernel combines characteristics of the microkernel as well as the monolithic kernel. Basic services are provided through a compact core that handles essential tasks such as memory and process management. In addition device drivers and file systems are provided via user space as separate processes [29].

In Figure 2.3 the hybrid structure is visible. Compared to a monolithic kernel, which has everything in the kernel space, or the microkernel based operating system, which puts all but the most basic parts into the user space, the hybrid kernel based operating system moved the Application IPC and the device

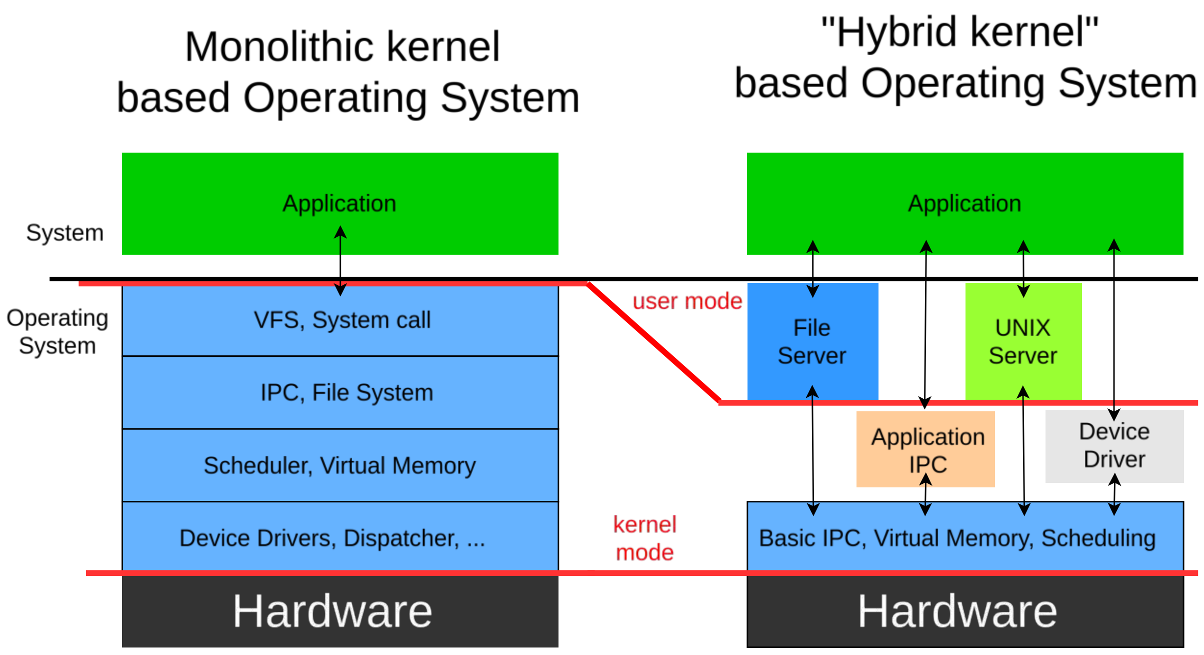
Monolithic kernel based Operating System
"Hybrid kernel" based Operating System
System
Application
Application
Operating System
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
user mode
kernel mode
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
Hardware
Hardware

Figure 2.3: Structure of monolithic and hybrid kernel based operating systems

drivers back to the kernel space.

It allows for strongly logically connected components to reside in the same memory space. The advantage of this approach is that it can utilize the modularity and flexibility of a microkernel without losing the performance benefits of a monolithic kernel [23, 30].

The hybrid kernel approach is used in various modern systems such as Apple's iOS or Windows 10 [30].

## 2.3 Device drivers

As mentioned in chapter 1, device drivers serve as a translator between the hardware and the operating system. This allows the OS to communicate with the hardware and the hardware with the OS. This means most of the device drivers will run in kernel space, although user space drivers also exist in Linux, such as Filesystem in Userspace (FUSE) [31]. Using user space drivers may lead to worse performance and higher CPU loads, with a relative increase of 31% [32].

Linux allows to develop kernel modules independently without recompiling the whole kernel source. Those are so called 'out-of-tree' modules. This is achieved via the Linux Kernel Module (LKM) framework, which allows for dynamically linking of external modules [1]. In Figure 2.4, the building and inserting of an out-of-tree kernel module using the 'insmod' command is shown.

It is not possible to load all kinds of modules via the LKM framework. CPU scheduler or memory

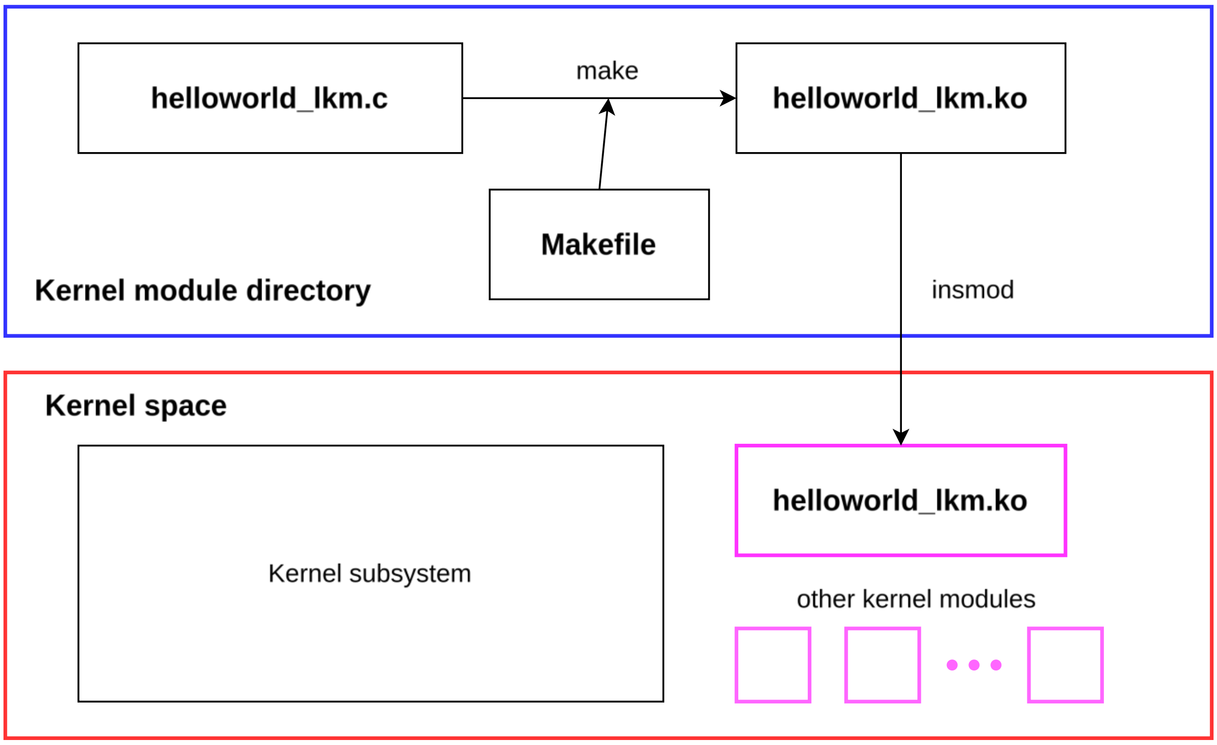

Figure 2.4: Building and inserting a kernel module [1]

management can't be loaded dynamically into the kernel. So those kernel modules are practical for the use with device drivers [1].

The Linux kernel assigns a unique number to each driver, the so called 'major' and 'minor' numbers. They are stored in the same 32-bit integer. The 20 Least Significant Bits (LSB) are for the minor number and the remaining 12 Most Significant Bits (MSB) are used for the major number [33].

There are three types of device drivers: block device drivers, character device drivers as well network device drivers. Network drivers are needed when working with a network interface, block drivers are used for mass storage and character device drivers are the most flexible ones. This thesis also focuses on the development on character device drivers. A character device driver is based on a stream of bytes, this is similar to a serial port [15].

## 2.4 Rust for Linux

Rust for Linux (RFL) is a device driver Application Programming Interface (API) that enables the development of Linux kernel extensions using the programming language Rust. It is a systems programming language developed by Mozilla and has focus on memory safety and performance [34].

Rust is designed to provide safety while still keeping the performance characteristics of unsafe languages. It is a programming language that has the ability to eliminate a wide class of low-level vulnerabilities and therefore improving the security of the Linux kernel. Rust implements memory safety without the need for a managed runtime or a garbage collector. It also uses the the concept of ‘ownership’, which means that every value is assigned to a single owner. This defines the scope in which it is characterized. Through function calls it’s also possible to move the ownership. The caller will loose all access to the value after this call. Using the ownership concept allows for explicit deallocation. After the lifetime of the value, the value is dropped and the storage is regained. This confined ownership model is sometimes too restrictive. So Rust allows to use ‘unsafe’ Rust. Unsafe Rust allows operations such as dereferencing raw pointers in C [11]. Rust also uses so called ‘traits’. Traits are a collection of properties a certain type can use. They can guarantee to the compiler that the used type behaves a certain way [2].

In addition, certain aspects of low-level driver code have the need to use different unsafe operations such as unsafe type casts, combinations of manual memory management and reference counting or arithmetic pointer operations, which could lead to a decrease of the overall security of Rust [11].

Device drivers are one of the biggest sources of OSes errors in the kernel [35]. Mitigating those vulnerabilities would create a more stable and secure OS. There have been various different approaches to improve the security of the kernel. For example executing device drivers in an isolated subsystem which resides on top of minimal microkernels [36]. Another approach is using virtual machines [37]. Backward-compatible driver execution frameworks is another approach to increase security [38]. As well as different methodologies using software and hardware in the kernel itself [39, 40]. But as of now, none of these options are in the mainline kernel due to different issues, such as high performance overhead and practical use regarding security features.

This leads to a search for a more practical solution, with low-overhead, safe programming and the ability of writing device drivers in Rust. RFL implements certain Rust bindings for individual kernel subsystems, such as network, block , as well as, non-volatile memory on Peripheral Component Interconnect (PCI)e. With those bindings, Rust has the ability to communicate with kernel interfaces that are implemented in C. In addition, those Rust bindings enable a safe Rust interface using a set of high-level abstractions and wrapper types. A schematic overview of the the interactions between Rust Drivers, Rust Abstractions and C Code is shown in Figure 2.5.

Rust drivers internally include run-time checks and guarantee correct reference counting. This allows that the driver can be utilized in a safe subset of Rust. Furthermore, this enables the security benefits of the Rust programming language [11].

RFL is still new, it was officially added to the kernel in 2021 and the first Rust drivers appeared in December 2023 in the kernel. In lines of code, it is 0.125% of the whole kernel code. Most of the code is found in scheduling, memory management, and Interrupt Request (IRQ) infrastructure. Drivers, on

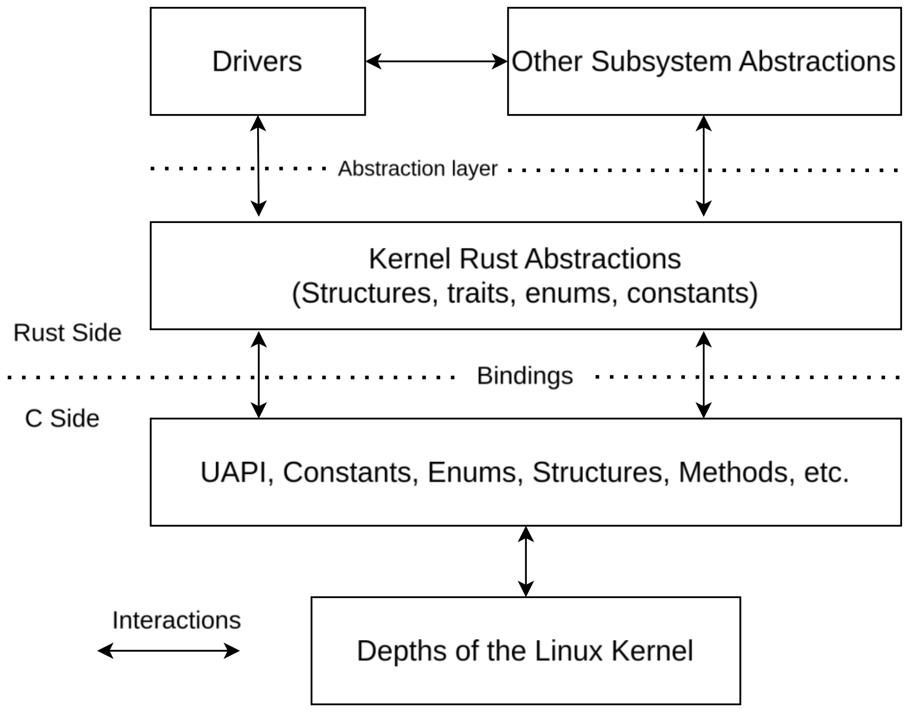
Drivers
Other Subsystem Abstractions
Abstraction layer
Kernel Rust Abstractions
(Structures, traits, enums, constants)
Rust Side
Bindings
C Side
UAPI, Constants, Enums, Structures, Methods, etc.
Interactions
Depths of the Linux Kernel

Figure 2.5: Overview of Rust Drivers, Rust Abstraction and C Code interactions [2]

the other hand, have only been around 2% [11, 41].

To write device driver kernel modules, RFL relies on bindgen, which automatically creates foreign function interface (FFI) bindings from the header files in the kernel [10]. Bindgen uses the C header files to translate the C structures and function declaration into compatible structures for Rust. Those bindings are then used to create a kernel 'crate' that is utilized as a trusted layer between the device driver implemented in safe Rust and the unsafe kernel. The Rust compiler cannot verify Linux C code, so the Rust drivers that use the provided safe interface by abstractions need to trust that the underlying C code is correct [2, 11].

In the following sections the three existing driver projects, Non-Volatile Memory Express (NVMe), Native Rust User Datagram Protocol (UDP) Tunneling Network Driver and the Null Block Driver will be covered. The Null Block Driver is the only driver that is already actively used in the kernel.

### 2.4.1 NVMe Driver

NVMe based solid state devices allow for high performance in regard to latency and peak bandwith. It is a software based standard, with the purpose of optimizing Solid State Disks (SSDs) attached through the PCIe interface [42].

The Rust NVMe driver project aims to implement a PCI NVMe driver in safe Rust for use within the Linux Kernel. Its primary goal is to enable the development of safe Rust abstractions and demonstrate the viability of Rust as a language for building high-performance device drivers.

As shown in Figure 2.6 the performance of the Rust NVMe driver is similar to the C driver for 4 KiB block sizes. The C driver performs better than the Rust driver by up to 6% with a 512 B block size [3].

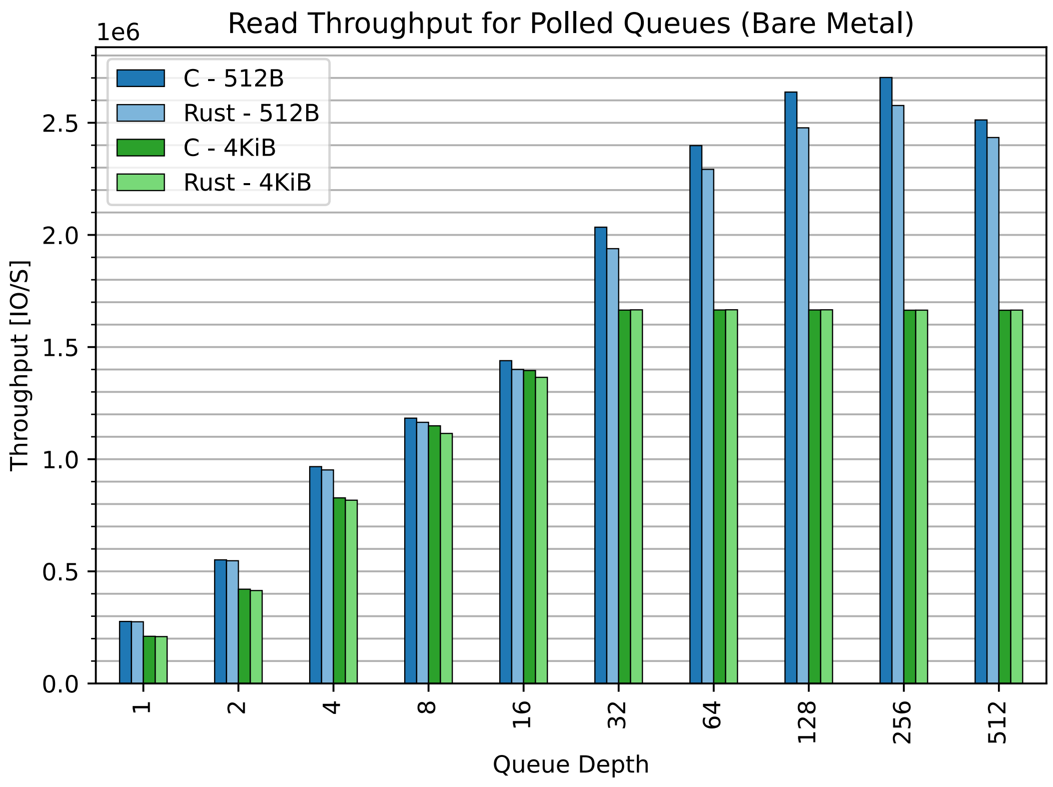

Figure 2.6: Analysis of the Rust and C NVMe driver (January 2023) [3]

At the moment, this driver is not intended for general use and is still in development.

### 2.4.2 Native Rust UDP Tunneling Network Driver

Gonzales et al. concentrate on the implementation of a Rust-based UDP tunneling network driver in the Linux kernel. The authors compare the performance to a similar driver written in C. In contrary to the C driver, the Rust driver utilizes abstractions, which increases the speed of the development process.

To allow developers to perform operations that the compiler can not guarantee are safe, Rust provides the unsafe keyword. This lets programmers mark specific functions or code blocks where the compiler should skip certain memory safety checks. In these cases, it is up to the developer to ensure the code upholds Rust's safety guarantees even without compiler enforcement. However, this introduces a challenge: while only unsafe code can directly cause memory issues, the resulting bugs might appear elsewhere in the program, even within safe code.

Due to the more complex process, such as handling pointer structures with socket buffer, more

| Interface | Mean | Min | Max | $\sigma$ | Points outside 95% interval |
|---|---|---|---|---|---|
| Baseline | 122.2 | 117 | 127 | 1.10 | 176 |
| C | 126.8 | 120 | 132 | 1.37 | 273 |
| Rust | 127.1 | 121 | 134 | 1.34 | 176 |

Table 2.1: Latency Measurements in Microseconds

| Interface | Mean | Min | Max | $\sigma$ |
|---|---|---|---|---|
| Baseline | 934.30 | 930.95 | 934.39 | $7.97 \times 10^{-2}$ |
| C | 915.79 | 913.92 | 915.93 | $8.15 \times 10^{-2}$ |
| Rust | 915.78 | 911.89 | 915.92 | $1.03 \times 10^{-1}$ |

Table 2.2: Throughput Statistics in Mbps

effort is required to safely wrap the C API. The results show that the Rust driver has a slightly higher latency of 0.19% and a marginally lower throughput of -0.0009% as seen in Table 2.1 and Table 2.2 [2].

### 2.4.3 Null Block Driver

A null block driver refers to a driver that simulates a block device but does not actually perform any real I/O operations. Its main use case is for testing and debugging purposes, because it discards all data written to it. Furthermore, it can also be used in scenarios where interaction with real storage is unnecessary, but integration with the kernel block subsystem is still required [43].

# Chapter 3

# Evaluation Platform Design

In this chapter the evaluation platform design will be discussed, which is used for the development of device drivers, including the choices for platform, tools and kernel configurations. The main goal of this chapter is to discuss the setup, the cross-compilation of a customized kernel for the Raspberry Pi 4 Model B, the target platform and the used tools. These customizations include support for Rust, on which the future device drivers will be developed and tested.

## 3.1 Virtual Machine Setup

The development environment is hosted on a Virtual Machine (VM) running Ubuntu Server 22.04 Long Term Support (LTS). The host for the VM is a Windows 11 machine. The VM is utilized as a build server for cross-compiling the kernel and modules. Using a VM instead of a native system has the purpose of using an isolated and controlled building environment. The decision to use Ubuntu 22.04 LTS was mainly chosen because of its stability, long-term support and compatibility for the required tools [44].

## 3.2 Target platform: Raspberry Pi 4 Model B

As the goal of the thesis is to compare I/O device drivers in both C and Rust, a Raspberry Pi 4 Model B implementation was selected. The single-board computer was chosen due to its performance, versatility and support for Linux.

It is an efficient and powerful minicomputer, that is constantly improving due to the development involvement of embedded systems scholars and researchers. The Raspberry Pi 4 Model B is used in multiple applications from home automation system, motion capture security cameras, Artificial Intelligence (AI) assistants as well applications across the biological domain and the implementation of neural networks [45–47].

### 3.2.1 Technical specifications

Below the Specifications of the Raspberry Pi 4 Model B are presented:

- Processor: Broadcom BCM2711, Quad core
- Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- RAM: 8GB LPDDR4-3200 SDRAM
- Bluetooth: Bluetooth 5.0, BLE
- Wi-Fi: 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless
- USB: 2 USB 3.0 ports; 2 USB 2.0 ports
- Ethernet: Gigabit Ethernet
- HDMI: 2 × micro-HDMI ports (up to 4kp60 supported)
- Storage: MicroSD Card Slot
- Power Supply: 5.1V 3A USB Type C Power
- Dimensions: 85.6mm × 56.5mm

Furthermore the Raspberry Pi 4 Model B allows for simple interaction with the hardware via GPIO pins and peripherals. To communicate with the Raspberry Pi 4 Model B a Universal Serial Bus (USB) to Transistor-Transistor Logic (TTL) converter is used as seen in Figure 3.1.

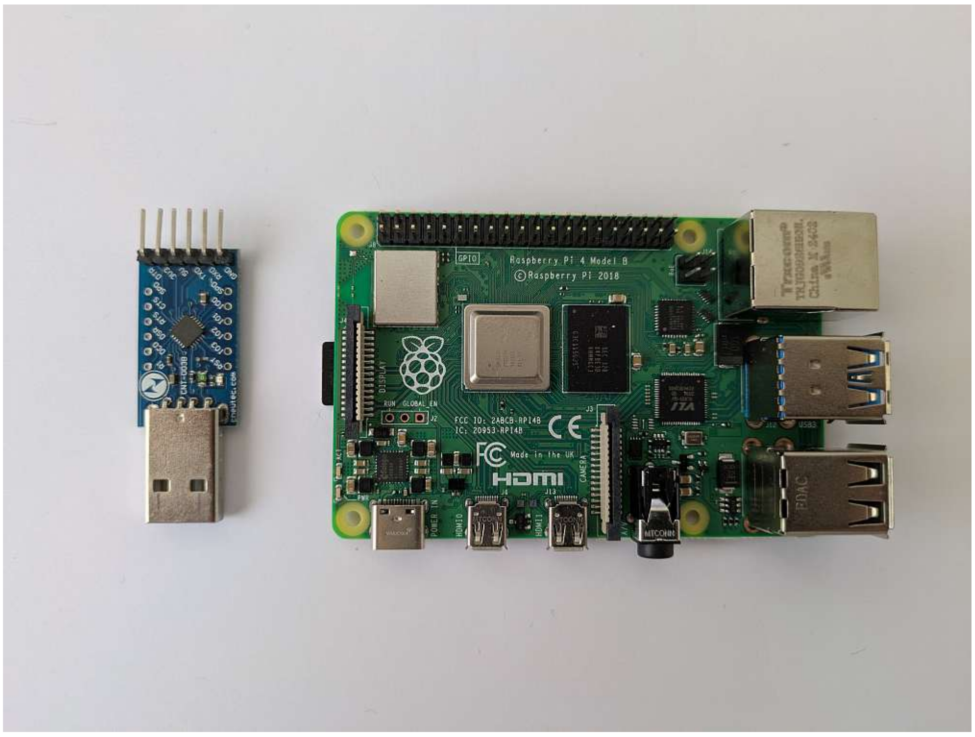

Figure 3.1: Image of Raspberry Pi 4 Model B with USB to TTL converter

The USB to TTL converter allows to communicate between USB-enabled devices, such as a laptop and TTL logic devices, such as the Raspberry Pi 4 Model B. This is needed because kernel messages appear before Secure Shell (SSH) is available and a serial connection helps in the debugging process.

The serial connection is set as follows:

- Baudrate: 115200
- Databits: 8
- Parity: none
- Stopbits: 1
- Timeout: 10

## 3.3 Kernel Selection and Customization

The kernel, which is used in this project is based on the official Raspberry Pi github respository, to be specific the 6.12.y branch is used. This kernel was chosen for multiple reasons:

- With its latest changes in January 2025, the software reflects the current state-of-the-art.
- Its projected end-of-life (EOL) is December 2026 [48].
- It contains all the newest changes to support Rust.

The exact kernel used is the 6.12.9-v8. V8 means that it is running on a 64 bit architecture instead of a 32 bit one. To differentiate the custom kernel, the name was changed to '6.12.9.-v8-FTG_V2'. By default, Rust is not enabled on the default kernel, so some changes needed to be made to the kernel to enable Rust support and add the implemented Rust samples. Here are the most important modifications, created by the kernel helper script 'diffconfig'.

```
$ LOCALVERSION "" -> "-v8-FTG_V2"
$ MODVERSIONS y -> n
$+RUST y
$+RUSTC_VERSION_TEXT "rustc 1.84.0 (9fc6b4312 2025-01-07)"
$+SAMPLES_RUST y
```

Before the introduction to the actual programming part and the first simple program in Rust and C, it is necessary to delve a bit more into the mechanics of the Linux kernel and especially the logging.

To emit messages in user space with the C programming language the printf() glibc API can be utilized [49]. In the kernel, on the other hand, there is no access to the printf() API.

Therefore a reimplemented version of the printf() command - the pr_*() or dev_*() macro - is used. For the first simple 'Hello, World!' program (see Listing 4.1) the pr_info() macro will be used to write to the kernel memory ring buffer. In the main module, the I/O device driver, the dev_*() macro will get utilized, because it is specifically designed for logging messages in the context of device drivers [1].

## 3.4 Tools

This subsection will discuss the deployed tools to measure performance and energy consumption of the device drivers.

### 3.4.1 FNB58USB tester

To measure power consumption, a FNB58USB tester will be used. It is a USB tester device which is designed to measure and display various electrical parameters of USB ports as well as connected devices. The FNB58USB tester can be used to measure voltage, current, power and energy consumption. The specification is as shown in Table 3.1:

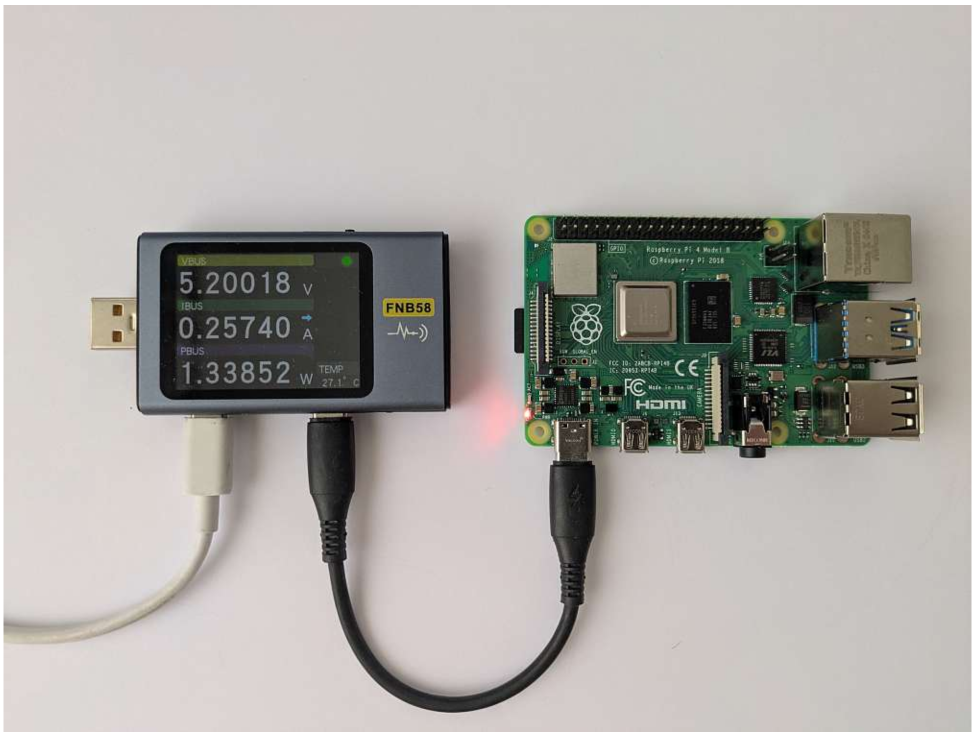

Figure 3.2: Image of the test setup with FNB58 and Raspberry Pi 4 Model B

According to the specifications in Table 3.1 the accuracy of the current at 0.5 A could be off by ±0.000 27 A. Therefore, the meter will be used for long-term stress tests, which should reveal trends related to power usage. In addition, it has been used in other embedded systems based projects to measure energy consumption [50].

As shown in Figure 3.2, the FNB58 is connected to a power source and the Raspberry Pi 4 Model B via

USB C. For the test setup the USB to TTL converter is removed to not interfere with the measurements.

Table 3.1: Device Specifications

| **Index** | **Range** | **Resolution** | **Accuracy** |
|---|---|---|---|
| Monitor voltage | 4 V to 28 V | 0.000 01 V | ± (0.2 ‰ + 2) |
| Monitor current | 0 A to 7 A | 0.000 01 A | ± (0.5 ‰ + 2) |
| Monitor power | 0 W to 120 W | 0.000 01 W | ± (0.5 ‰ + 2) |
| Load equivalent internal resistance | 0 Ω to 9999.9 Ω | 0.0001 Ω | ± (0.5 ‰ + 2) |
| D+/D- voltage | 0 V to 3.3 V | 0.001 V | ± (1.0 ‰ + 2) |
| Equipment temperature | −10 °C to 100 °C | 1 °C | ± (1.2 ‰ + 3) |
| Capacity | 0 A h to 9999.99 A h | 0.000 01 A h | |
| Energy used | 0 W h to 9999.99 W h | 0.000 01 W h | |
| Cable resistance | 0 Ω to 9999.9 Ω | 0.0001 Ω | |
| Equipment running time | 99d 23h 59min 59s | 1 s | |
| Record time | 99d 23h 59min 59s | 1 s | |

# Chapter 4

# Test Case Development

As already mentioned in Chapter 1, the kernel modules will be implemented on a Raspberry Pi 4 Model B. In Chapter 3 the complete setup of the kernel and their changes got explained. This chapter addresses the writing of the modules in Rust and C.

## 4.1 Writing a basic kernel module

It has been customary in the programming world to start with a simple program called ‘Hello, World!’. The first module will be a simple ‘Hello, World!’ program, which prints to the kernel log. In the following a comparison between the C ‘Hello, World!’ module and the Rust ‘Hello, World!’ module will be presented. In this chapter, the source code of the module as well as the generated assembly code will be discussed. For readability purposes the addresses of the Rust assembly code are shortened.

### 4.1.1 Analysis of C ‘Hello, World!’ code

The beginning in Listing 4.1 shows the license and the included header files, which provide the necessary files for kernel module development. From line 6 until line 9 the metadata for the module is defined.

In line 11 the actual code starts, the module’s initialization function. This means it will be executed as soon as the module is loaded. If it is loaded at startup, it will write ‘Hello, World! From a C module’ into the kernel memory buffer at the start. Defined at line 17 is the exit function. This defines the actions that are taken as soon as the module is removed. It can also be referred to as a clean-up function.

Lines 22 and 23 are essential because they are responsible for registering the functions as initialization and cleanup functions.

Listing 4.1: helloworld_c.c: A simple kernel module in C that prints ‘Hello, World!’

```c
 1 // SPDX-License-Identifier: GPL-2.0
 2 #include <linux/init.h>
 3 #include <linux/module.h>
 4 #include <linux/kernel.h>
 5 
 6 MODULE_LICENSE("GPL");
 7 MODULE_AUTHOR("Fabian T. Garber");
 8 MODULE_DESCRIPTION("A simple Hello World module in C");
 9 MODULE_VERSION("1.0");
10 
11 static int __init hello_init(void)
12 {
13     pr_info("Hello, World! From a C module.\n");
14     return 0;
15 }
16 
17 static void __exit hello_exit(void)
18 {
19     pr_info("Goodbye, World! From a C module.\n");
20 }
21 
22 module_init(hello_init);
23 module_exit(hello_exit);
```

### 4.1.2 Analysis of Rust ‘Hello, World!’ code

In the Rust module in Listing 4.2 there is a similar structure. Line 2 is a documentation comment, which gives a brief description of the module. Instead of ‘includes’, like in C, Rust uses the use command to import all the necessary items, which are needed to interact with the kernel. The module block at line 6 sets all the metadata. Line 14 creates a ‘struct’ name ‘HelloWorld’. This ‘struct’ represents the module itself, usually it will hold any module-specific data or state. Due to the simplicity of the program, it does not have any member in this case.

Line 16 implements the kernel::Module trait for the ‘HelloWorld struct’ [1]. This trait defines the methods the kernel module must implement. This trait defines the methods the kernel module must

[1]A trait in Rust defines a set of methods that types can implement to share common behavior.

implement. Similar to the C example, we have an initialization function in line 17. The Rust example, different to the C one, returns the initialized 'HelloWorld struct', indicating successful initialization. For the cleanup function the 'Drop' trait is implemented for the 'HelloWorld struct'. As seen in line 24 the 'Drop' trait defines the 'drop' method, which is called when an instance of the 'struct' goes out of scope. In this case when the module gets removed/unloaded.

Listing 4.2: helloworld_rust.rs: A simple kernel module in Rust that prints 'Hello, World!'

```rust
 1 // SPDX-License-Identifier: GPL-2.0
 2 //! Simple Hello World Rust kernel module.
 3
 4 use kernel::prelude::*;
 5
 6 module! {
 7     type: HelloWorld,
 8     name: "hello_world_rust",
 9     author: "Fabian T. Garber",
10     description: "A simple Hello World Rust kernel module",
11     license: "GPL",
12 }
13
14 struct HelloWorld;
15
16 impl kernel::Module for HelloWorld {
17     fn init(_module: &'static ThisModule) -> Result<Self> {
18         pr_info!("Hello, World! From a Rust module.\n");
19         Ok(HelloWorld)
20     }
21 }
22
23 impl Drop for HelloWorld {
24     fn drop(&mut self) {
25         pr_info!("Goodbye, World! From Rust a module.\n");
26     }
27 }
```

To summarize, apart from the differences in syntax the structure of the programs is fairly similar: Definition of module metadata, initialization and cleanup function and both modules invoke a print method that prints to the kernel ring buffer.

### 4.1.3 Analysis of the C assembly code

Similar structure on the surface level does not tell us what it looks like on the machine level. To analyze that, the generated object files were disassembled into assembler code. In the next subsection the differences in actual assembly code will be discussed.

The assembler code for the C module is shown in Listing 4.3. Only relevant lines of code will be analyzed - load and move instructions will not be discussed. There are two main blocks of instructions: the 'init_module' from line 6 to 16 and the 'cleanup_module' from line 20 to 29.

First is the 'init_module'.

Line 7 starts with calculation of a pointer authentication code (PAC) for the stack pointer (sp) and saves it. This is done to protect against stack smashing attacks. This is an attack where the a program writes more data to a buffer located on a stack than what is actually allocated for that buffer, therefore creating a stack buffer overflow. This can lead to corruption of adjacent data, such as return addresses. Attackers could exploit this by executing arbitrary code [51].

Line 8 stores the frame pointer (x29) and the link register (x30) onto the stack.

Line 9 moves the sp to x29 and line 10 loads the 'init_module' function.

Line 12 branches to the _printk function.

Line 15 authenticates the sp using the previously stored PAC. This is done to check that the sp has not been tempered with.

Line 16 returns from the function.

The cleanup_module won't be discussed because, apart from the move instruction that returns 0, it's the same.

A 'Hello, World!' program is a very basic program, therefore the assembly code is quite short and understandable.

Listing 4.3: C 'Hello, World!' assembly code

```
 1 helloworld_c.o:     file format elf64-littleaarch64
 2
 3
 4 Disassembly of section .init.text:
 5
 6 0000000000000000 <init_module>:
 7    0:   d503233f        paciasp
 8    4:   a9bf7bfd        stp     x29, x30, [sp, #-16]!
 9    8:   910003fd        mov     x29, sp
10    c:   90000000        adrp    x0, 0 <init_module>
11   10:   91000000        add     x0, x0, #0x0
12   14:   94000000        bl      0 <_printk>
13   18:   2a1f03e0        mov     w0, wzr
14   1c:   a8c17bfd        ldp     x29, x30, [sp], #16
15   20:   d50323bf        autiasp
16   24:   d65f03c0        ret
17
18 Disassembly of section .exit.text:
19
20 0000000000000000 <cleanup_module>:
21    0:   d503233f        paciasp
22    4:   a9bf7bfd        stp     x29, x30, [sp, #-16]!
23    8:   910003fd        mov     x29, sp
24    c:   90000000        adrp    x0, 0 <cleanup_module>
25   10:   91000000        add     x0, x0, #0x0
26   14:   94000000        bl      0 <_printk>
27   18:   a8c17bfd        ldp     x29, x30, [sp], #16
28   1c:   d50323bf        autiasp
29   20:   d65f03c0        ret
```

## 4.1.4 Analysis of the Rust assembly code

In this subsection the generated assembly code from Rust will be looked at. It is shown in Listing 4.4. At first glance it is clear that the amount of instructions is higher than on the C assembly code. It has around 4 times more instructions.

As before we will take a closer look at the 'init_module'.

Compared to the C assembly code, the Rust assembly code has 43 instructions and the C version 10.

Not all the individual instructions will be discussed, instead only the differences and the reason for those differences will get examined.

- Compared to the C assembly code the Rust version allocates significantly more stack space. Rust allocates 80 bytes (Listing 4.4 line 90: sp,sp, #0x50) compared to the C version that allocates 16 bytes (Listing 4.3 line 8: x29, x30, [sp, #016]!).
- The Rust version uses more registers (x8, x19, x20, w8, w9) for temporary storage and calculation. The C version uses only the registers x0, x29 and x30, but none specific to temporary storage.
- The C version does not use a conditional branch instruction (b.ne) like the Rust version.
- In the Rust version at line 48 and line 90 the (..)print11call_printk is called, while the C version calls _printk. This indicates that the Rust code is using a wrapper function or at least a different mechanism for printing message to the kernel log.

Listing 4.4: Rust 'Hello, World!' assembly code

```
 1 helloworld_rust.o:     file format elf64-littleaarch64
 2
 3
 4 Disassembly of section .text:
 5
 6 0000000000000000 <<add1>helloworld_rustNtB2_10<HW><add2>_<modinit>>:
 7    0:   d503233f        paciasp
 8    4:   d10103ff        sub     sp, sp, #0x40
 9    8:   a9037bfd        stp     x29, x30, [sp, #48]
10    c:   9100c3fd        add     x29, sp, #0x30
11   10:   90000008        adrp    x8, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
12   14:   91000108        add     x8, x8, #0x0
13   18:   52800029        mov     w9, #0x1                        // #1
14   1c:   a90027e8        stp     x8, x9, [sp]
15   20:   52800108        mov     w8, #0x8                        // #8
16   24:   90000000        adrp    x0, 0 <_RNvNt<add2>_<k>print14format_strings4INFO>
17   28:   91000000        add     x0, x0, #0x0
18   2c:   90000001        adrp    x1, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
19   30:   91000021        add     x1, x1, #0x0
20   34:   910003e3        mov     x3, sp
21   38:   52800222        mov     w2, #0x11                       // #17
22   3c:   a901ffff        stp     xzr, xzr, [sp, #24]
23   40:   f9000be8        str     x8, [sp, #16]
24   44:   94000000        bl      0 <_RNv<add2>_<k>print11call_printk>
25   48:   a9437bfd        ldp     x29, x30, [sp, #48]
26   4c:   2a1f03e0        mov     w0, wzr
27   50:   910103ff        add     sp, sp, #0x40
28   54:   d50323bf        autiasp
29   58:   d65f03c0        ret
30
31 000000000000005c <<add1><hw>_rustNtB4_10<HW><add3>_4core3ops4drop4Drop4drop>:
32   5c:   d503233f        paciasp
33   60:   d10103ff        sub     sp, sp, #0x40
34   64:   a9037bfd        stp     x29, x30, [sp, #48]
35   68:   9100c3fd        add     x29, sp, #0x30
36   6c:   90000008        adrp    x8, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
37   70:   91000108        add     x8, x8, #0x0
```

```
  74:   52800029        mov     w9, #0x1                        // #1
  78:   a90027e8        stp     x8, x9, [sp]
  7c:   52800108        mov     w8, #0x8                        // #8
  80:   90000000        adrp    x0, 0 <_RNvNt<add2>_<k>print14format_strings4INFO>
  84:   91000000        add     x0, x0, #0x0
  88:   90000001        adrp    x1, 0 <<add1><hw>_rustNtB2_10H<HW><add2>_<modinit>>
  8c:   91000021        add     x1, x1, #0x0
  90:   910003e3        mov     x3, sp
  94:   52800222        mov     w2, #0x11                       // #17
  98:   a901ffff        stp     xzr, xzr, [sp, #24]
  9c:   f9000be8        str     x8, [sp, #16]
  a0:   94000000        bl      0 <_RNv<add2>_<k>print11call_printk>
  a4:   a9437bfd        ldp     x29, x30, [sp, #48]
  a8:   910103ff        add     sp, sp, #0x40
  ac:   d50323bf        autiasp
  b0:   d65f03c0        ret

00000000000000b4 <cleanup_module>:
  b4:   d503233f        paciasp
  b8:   d10143ff        sub     sp, sp, #0x50
  bc:   a9037bfd        stp     x29, x30, [sp, #48]
  c0:   f90023f3        str     x19, [sp, #64]
  c4:   9100c3fd        add     x29, sp, #0x30
  c8:   90000013        adrp    x19, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
  cc:   39400268        ldrb    w8, [x19]
  d0:   7100051f        cmp     w8, #0x1
  d4:   540001e1        b.ne    110 <cleanup_module+0x5c>  // b.any
  d8:   90000008        adrp    x8, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
  dc:   91000108        add     x8, x8, #0x0
  e0:   52800029        mov     w9, #0x1                        // #1
  e4:   a90027e8        stp     x8, x9, [sp]
  e8:   52800108        mov     w8, #0x8                        // #8
  ec:   90000000        adrp    x0, 0 <_RNvNt<add2>_<k>print14format_strings4INFO>
  f0:   91000000        add     x0, x0, #0x0
  f4:   90000001        adrp    x1, 0 <<add1><hw>_rustNtB2_10<HW><add2>_<modinit>>
  f8:   91000021        add     x1, x1, #0x0
  fc:   910003e3        mov     x3, sp
 100:   52800222        mov     w2, #0x11                       // #17
 104:   a901ffff        stp     xzr, xzr, [sp, #24]
 108:   f9000be8        str     x8, [sp, #16]
 10c:   94000000        bl      0 <_RNv<add2>_<k>print11call_printk>
 110:   a9437bfd        ldp     x29, x30, [sp, #48]
 114:   3900027f        strb    wzr, [x19]
 118:   f94023f3        ldr     x19, [sp, #64]
 11c:   910143ff        add     sp, sp, #0x50
 120:   d50323bf        autiasp
 124:   d65f03c0        ret

Disassembly of section .init.text:

0000000000000000 <init_module>:
   0:   d503233f        paciasp
   4:   d10143ff        sub     sp, sp, #0x50
   8:   a9037bfd        stp     x29, x30, [sp, #48]
   c:   a9044ff4        stp     x20, x19, [sp, #64]
  10:   9100c3fd        add     x29, sp, #0x30
  14:   90000008        adrp    x8, 0 <init_module>
  18:   91000108        add     x8, x8, #0x0
  1c:   52800033        mov     w19, #0x1                       // #1
  20:   a9004fe8        stp     x8, x19, [sp]
  24:   52800108        mov     w8, #0x8                        // #8
  28:   90000000        adrp    x0, 0 <_RNvNt<add2>_<k>print14format_strings4INFO>
  2c:   91000000        add     x0, x0, #0x0
  30:   90000001        adrp    x1, 0 <init_module>
  34:   91000021        add     x1, x1, #0x0
  38:   910003e3        mov     x3, sp
  3c:   52800222        mov     w2, #0x11                       // #17
  40:   a901ffff        stp     xzr, xzr, [sp, #24]
  44:   f9000be8        str     x8, [sp, #16]
  48:   94000000        bl      0 <_RNv<add2>_<k>print11call_printk>
  4c:   90000014        adrp    x20, 0 <init_module>
  50:   39400288        ldrb    w8, [x20]
```

```
54:   7100051f        cmp     w8, #0x1
58:   540001e1        b.ne    94 <init_module+0x94>  // b.any
5c:   90000008        adrp    x8, 0 <init_module>
60:   91000108        add     x8, x8, #0x0
64:   52800029        mov     w9, #0x1                      // #1
68:   a90027e8        stp     x8, x9, [sp]
6c:   52800108        mov     w8, #0x8                      // #8
70:   90000000        adrp    x0, 0 <_RNvNt<add2>_<k>print14format_strings4INFO>
74:   91000000        add     x0, x0, #0x0
78:   90000001        adrp    x1, 0 <init_module>
7c:   91000021        add     x1, x1, #0x0
80:   910003e3        mov     x3, sp
84:   52800222        mov     w2, #0x11                     // #17
88:   a901ffff        stp     xzr, xzr, [sp, #24]
8c:   f9000be8        str     x8, [sp, #16]
90:   94000000        bl      0 <_RNv<add2>_<k>print11call_printk>
94:   39000293        strb    w19, [x20]
98:   a9444ff4        ldp     x20, x19, [sp, #64]
9c:   a9437bfd        ldp     x29, x30, [sp, #48]
a0:   2a1f03e0        mov     w0, wzr
a4:   910143ff        add     sp, sp, #0x50
a8:   d50323bf        autiasp
ac:   d65f03c0        ret
```

Also taking a look at the rest of Listing 4.4, it is obvious that there is more overhead than on the C equivalent. The ‘cleanup_module’ consists of more instructions and in addition to that there are multiple instructions to set up the module and the ‘Drop’ trait.

As already mentioned in the introduction, Rust provides memory safety. To achieve this additional safety and error handling, further instructions are needed. Rust also has mangled names, which embeds more information about types and functions. This results in larger code size. An additional reason could be compiler optimization. The Rust compiler might not yet be as optimized as the C compiler.

## 4.2 GPIO Modules

### 4.2.1 Purpose and Functionality

The main purpose of GPIO modules is to interact with the environment using sensors and actuators. They provide control and monitoring capabilities and are therefore an essential part of embedded systems. GPIO pins can either be used as input, with data from an external source that is provided to the system, which is used for sensors or switches but they can also be configured as an output to communicate to outside devices, such as motors or relays [52, 53].

They can be implemented as virtual GPIO modules or physical GPIO modules in hardware.

The next subsections will discuss the differences of virtual and physical GPIO modules.

### 4.2.2 Hardware GPIO Device Drivers

Hardware GPIO device drivers are software components that provide an interface between the operating system and the physical GPIO pins on a hardware device.

These drivers allow to access and manipulate the GPIO pins. Key functionalities of hardware GPIO device drivers are pin configuration, data transfer, interrupt handling as well as pin multiplexing.

### 4.2.3 Virtual GPIO Device Drivers

Virtual GPIO device drivers simulate the behavior of GPIO pins without the need for actual hardware. They create a software abstraction of GPIO functionality to allow for the development and testing of applications. Virtual GPIO device drivers have some advantages over physical ones in the scope of this thesis, because there are no hardware dependencies.

There is no physical hardware required. Testing, debugging and comparing the two device drives to each other is easier because of a controlled environment. Virtual GPIOs provide a more consistent and reproducible test and in addition it allows to focus on the functionality and logic of the module without limitations from external factors, such as hardware.

## 4.3 C Module

We develop kernel modules as out-of-tree modules. This means it is not necessary to recompile the whole kernel, just the desired module. The modules are then getting uploaded to the Device Under Test (DUT) and inserted using the command 'insmod'. This allows for better flexibility and makes it easier to debug errors in the module. To interact with the module from userspace input/output control (ioctl), system calls are used.

To make it possible to actually use the device, it is necessary to define file operations. In Listing 4.5 a sample file operations 'struct' in C is shown. In the kernel it is located under 'include/linux/fs.h'. It allows to interact with the device via different methods. In Listing 4.5 the functionality for an owner, to open the device, release the device, read it and use ioctl is shown. This allows to communicate with the driver via ther userspace.

Listing 4.5: vgpio_c.c: File operations in C

```c
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,
    .read = device_read,
};
```

In Listing 4.6 the 'device_ioctl' function of the C module is shown. This function allows to communicate to the kernel module via the userspace using the 'GPIO_SET_VALUE' and 'GPIO_GET_VALUE' commands.

Listing 4.6: vgpio_c.c: Device ioctl in C module

```
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct gpio_data data;

    switch (cmd) {
        case GPIO_SET_VALUE:
            if (copy_from_user(&data, (struct gpio_data __user *)arg, sizeof(data)))
                return -EFAULT;
            if (data.pin < 0 || data.pin >= NUM_GPIO_PINS)
                return -EINVAL;

            gpio_values[data.pin] = data.value;

            if (debug)
                dev_info(dev, "GPIO[%d] set to %d\n", data.pin, data.value);
            break;

        case GPIO_GET_VALUE:
            if (copy_from_user(&data, (struct gpio_data __user *)arg, sizeof(data)))
                return -EFAULT;
            if (data.pin < 0 || data.pin >= NUM_GPIO_PINS)
                return -EINVAL;

            data.value = gpio_values[data.pin];

            if (copy_to_user((struct gpio_data __user *)arg, &data, sizeof(data)))
                return -EFAULT;
            break;

        default:
            return -EINVAL;
    }
    return 0;
}
```

## 4.4 Rust Module

The Rust file operations structure, shown in Listing 4.7, looks similar to the C module, but with some distinct differences. Most of the properties are the same (owner, open, release, read, and ioctl) except the last line. The 'Some' keyword is an 'enum' variant of the 'Option' type. The 'Option' type is used to represent an Optional value. This means every 'Option' is either 'Some' and contains a value, or 'None', and does not contain a value.

Here in in the 'file_operations', 'Some' is used to wrap function pointers. Setting it to 'Some' means that it is here and that it should be used. In case a function pointer is not used, it is set to 'None'. Another difference is the 'read' line. Here it specifies the C calling convention, which is needed to interact with the C code in the Linux kernel, the underscores are just used as placeholders for the parameters and the return values. This last line '..unsafe core::mem::zeroed() ' is necessary because it ensures that all remaining fields are properly initialized with zero.

Listing 4.7: vgpio_rust.rs: File operations in Rust module

```
static VGPIO_FOPS: VgpioFops = VgpioFops(kernel::bindings::file_operations {
    owner: THIS_MODULE,
    open: Some(vgpio_open),
    release: Some(vgpio_release),
    read: Some(vgpio_read as unsafe extern "C" fn(_, _, _, _) -> _),
    unlocked_ioctl: Some(vgpio_ioctl),
    ..unsafe { core::mem::zeroed() }
});
```

Listing 4.8: vgpio_rust.rs: Device ioctl in Rust module

```
pub extern "C" fn vgpio_ioctl(
    _file: *mut bindings::file,
    cmd: u32,
    arg: u64,
) -> c_long {
    let mut data: GpioData = unsafe { core::mem::zeroed() };

    match cmd {
        GPIO_SET_VALUE => {
            let ret = unsafe {
                bindings::copy_from_user(
                    &mut data as *mut GpioData as *mut core::ffi::c_void,
                    arg as *const core::ffi::c_void,
                    core::mem::size_of::<GpioData>() as u64,
                )
            };
            if ret != 0 {
                return -(bindings::EFAULT as c_long);
            }
            if data.pin < 0 || data.pin >= 8 {
                return -(bindings::EINVAL as c_long);
            }
            unsafe {
                VGPIO_PINS[data.pin as usize] = data.value;
            }
        },
        GPIO_GET_VALUE => {
            let ret = unsafe {
                bindings::copy_from_user(
                    &mut data as *mut GpioData as *mut core::ffi::c_void,
                    arg as *const core::ffi::c_void,
                    core::mem::size_of::<GpioData>() as u64,
                )
            };
            if ret != 0 {
                return -(bindings::EFAULT as c_long);
            }
            if data.pin < 0 || data.pin >= 8 {
                return -(bindings::EINVAL as c_long);
            }
            unsafe {
                data.value = VGPIO_PINS[data.pin as usize];
            }
            let ret = unsafe {
                bindings::copy_to_user(
                    arg as *mut core::ffi::c_void,
                    &data as *const GpioData as *const core::ffi::c_void,
                    core::mem::size_of::<GpioData>() as u64,
                )
            };
            if ret != 0 {
                return -(bindings::EFAULT as c_long);
            }
        },
        _ => return -(bindings::EINVAL as c_long),
    }
    0
}
```

In Listing 4.8, the ‘device_ioctl’ function of the Rust module is shown. Both C and Rust handle GPIO set/get ioctl commands in a similar way: they copy data from user space, validate the pin number, update or retrieve the pin value, and copy data back if needed. Rust uses fixed-size arrays and explicit unsafe blocks for raw pointer manipulation, and kernel bindings to interface with the underlying C kernel API. In addition Rust’s implementation adds type safety and clearer boundaries around unsafe operations.

## 4.5 Test Cases

To test the differences in the C and Rust modules, different variants of the C and Rust modules are used. All of them are tested with 1 billion GPIO write accesses.

To determine the efficiency of the the used modules, the modules are separated into three different variants. In the first case, they are using spinlocks and a wait queue. The second is without a wait queue but with a spinlock and the third module variant is not using a wait queue nor a spinlock.

A spinlock is the lowest-level of mutual exclusion (mutex) mechanism, which is used for synchronizing access to data shared in the kernel. The spinlock has a serious impact on the performance of the applications it is used in. Therefore, a spinlock should only be used for operations that are finished in a very short period of time, to not waste resources [54, 55].

A wait queue is basically used to process blocking input and output. This is done by either waiting for a given event to occur or waiting until a certain condition becomes ‘true’. It is a list that includes sleeping processes and a spinlock to protect the access [56].

## 4.6 Power consumption

The fast development of IoT devices leads to more and more systems deployed. Large-scale IoT systems are composed of thousands of devices. Many of those systems are deployed out-doors and need batteries to be powered. There has been extensive research to extend the battery lifespan of those systems. There are mainly two ideas for energy saving: first is reducing the power consumption and second is decreasing the active phase of the devices [57].

GPIO pins are used for waking up the device and/or driving external actuators. Furthermore, they are used for communication with other devices or systems, using various transmission protocols. This is why it is important that device drivers, for example for GPIO devices, are efficient and quick.

Therefore, it is crucial to minimize the time spent in the active phase to reduce the overall energy consumption of the device.

# Chapter 5

# Evaluation

This chapter presents data for the measurements that were taken during the tests. In Figure 5.1, charts of the Raspberry Pi 4 Model B running without any of the virtual GPIO modules are shown. The x-axis is formatted as hh:mm:ss. The measured values are current, energy, power and voltage. The mean power is 1.386 W.

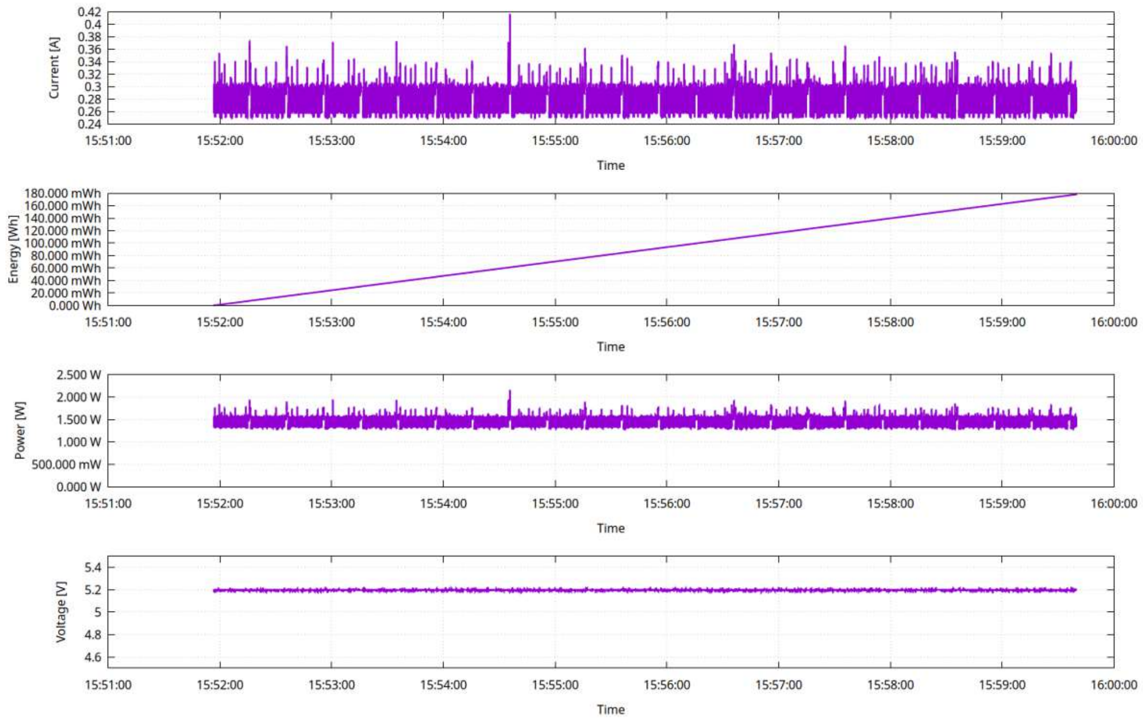

Figure 5.1: Baseline chart for Raspberry Pi 4 Model B

In Figure 5.2 an example test run of the C kernel module with wait queue and spinlock for 1 billion write accesses is shown.

The first chart in Figure 5.2 shows the used current, the second chart shows an energy consumption

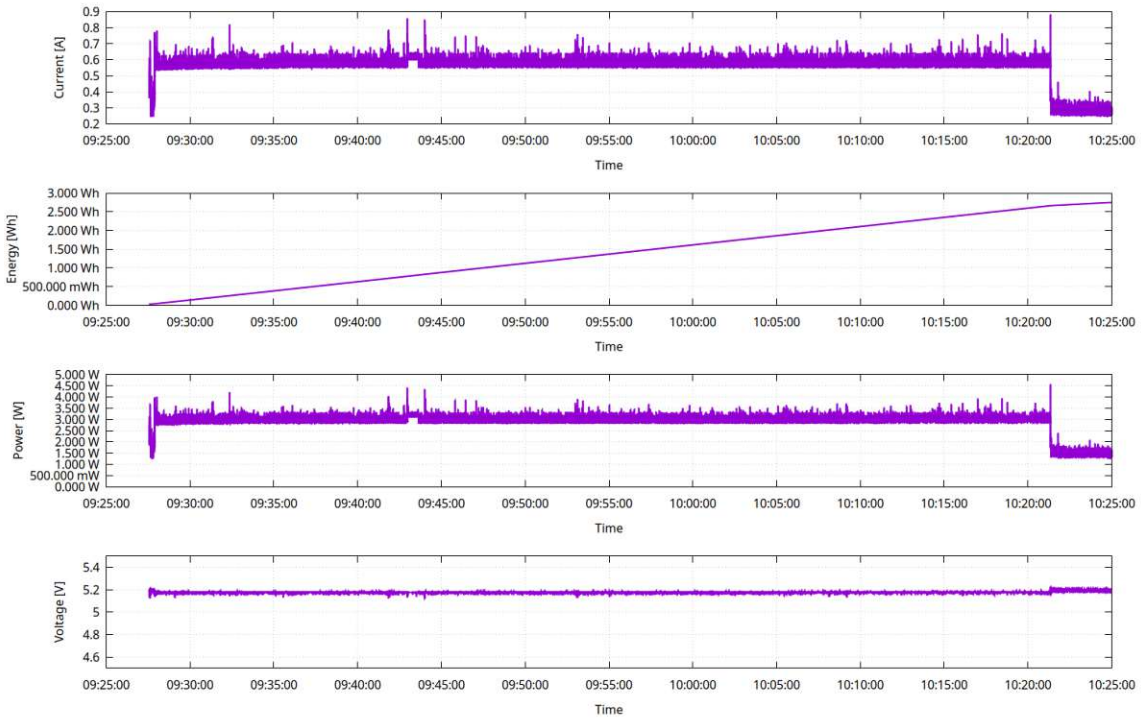

Figure 5.2: Graph for 1 billion write accesses with C kernel module with wait queue and spinlock

of 2.66 W h, the third chart shows the power consumption. In the current or power part of the figure it is visible when the module started and when it ended due to an increase and later a decrease of the current consumption. In the fourth chart the voltage is visible, which is stable at around 5.2 V. The Rust charts follow a similar pattern, but with different values.

As mentioned previously without the module active the DUT has a mean power consumption of 1.386 W. The mean power consumption during this specific run was 2.930 W.

## 5.1 Statistics

To determine the number of tests that need to be conducted for the C module with wait queue and spinlock, a confidence level of 99% was set. A margin of error (E) of 60 seconds was specified. Five runs were measured, and the results are shown in Table 5.1. The required number of tests was calculated using the formulas derived from sample size determination principles [58].

| Run | Time (s) |
|---|---|
| 1 | 3207 |
| 2 | 3649 |
| 3 | 3348 |
| 4 | 3412 |
| 5 | 3040 |

Table 5.1: Run times for 1B GPIO writes on the C module with wait queue and spinlock.

**Sample Mean ($\bar{X}$):**

$$\bar{X} = \frac{\sum X_i}{n_0} = 3333.2\text{s}$$

**Sample Standard Deviation ($s$):**

$$s = \sqrt{\frac{\sum(X_i - \bar{X})^2}{n_0 - 1}} \approx 228.05\text{s}$$

**Required Number of Tests ($n$):**

$$n = \left(\frac{z \cdot s}{E}\right)^2 \approx 56.12$$

So 57 tests are necessary to have a confidence of 90% with a 50 seconds error margin.

The variables are described in the following list:

- $n_0$: The initial sample size number.
- $n$: The number of tests required to achieve the desired confidence level and margin of error.
- $\bar{X}$: The sample mean of the run times.
- $s$: The sample standard deviation of the run times.
- $z$: The z-score corresponding to the desired confidence level. For a 90% confidence level, $z \approx 1.645$.
- $E$: Is the margin of error.
- $\sigma$: The population standard deviation. In practice, this is often estimated using the sample standard deviation $s$.

The same was done for the other variants. In Table 5.2 the confidence rate, margin of error and the needed number of tests is presented.

| Module variant | Margin of Error (s) | Required Samples | Confidence Level (%) |
|---|---|---|---|
| C module | 1 | 17 | 99 |
| Rust module | 1 | 14 | 99 |
| C module (wait queue) | 20 | 84 | 99 |
| Rust module (wait queue) | 20 | 35 | 99 |
| C module (wait queue / spinlock) | 60 | 96 | 99 |
| Rust module (wait queue / spinlock) | 60 | 72 | 99 |

Table 5.2: Required number of samples for different confidence levels and margins of error.

## 5.2 Results

The results of the average running times and the mean power consumption for each module and variant are shown in Table 5.3. As expected, the running time of each variant increases with the complexity in the C module and the Rust module.

As the complexity increases, the running times get longer. This is especially significant when using a spinlock, as it blocks the CPU and keeps it busy-waiting until the lock is released.

The chart shows that in each variant the Rust implementation is slower than the C implementation by about 8% in the basic variant, to 11% in the variant with wait queue and by 14.3% in the variant with wait queue and spinlock.

| Module variant | Average running times (s) | Percentage difference (%) |
|---|---|---|
| C module | 581 | |
| Rust module | 630 | +8.43 |
| C module (wait queue) | 1347 | |
| Rust module (wait queue) | 1539 | +14.25 |
| C module (wait queue / spinlock) | 3221 | |
| Rust module (wait queue / spinlock) | 3572 | +10.89 |

Table 5.3: Average running times and percentage differences of the module variants.

| Module variant | Energy consumption (Wh) | Percentage difference (%) |
|---|---|---|
| C module | 0.39 | |
| Rust module | 0.57 | +46.15 |
| C module (wait queue) | 1.12 | |
| Rust module (wait queue) | 1.43 | +27.68 |
| C module (wait queue / spinlock) | 2.54 | |
| Rust module (wait queue / spinlock) | 3.40 | +33.86 |

Table 5.4: Energy consumption and percentage difference of the module variants.

In Figure 5.3 and Figure 5.4 a better visual representation of the compared data is shown.

Even with the lowest complex variant of the module in Rust, the average power consumption is

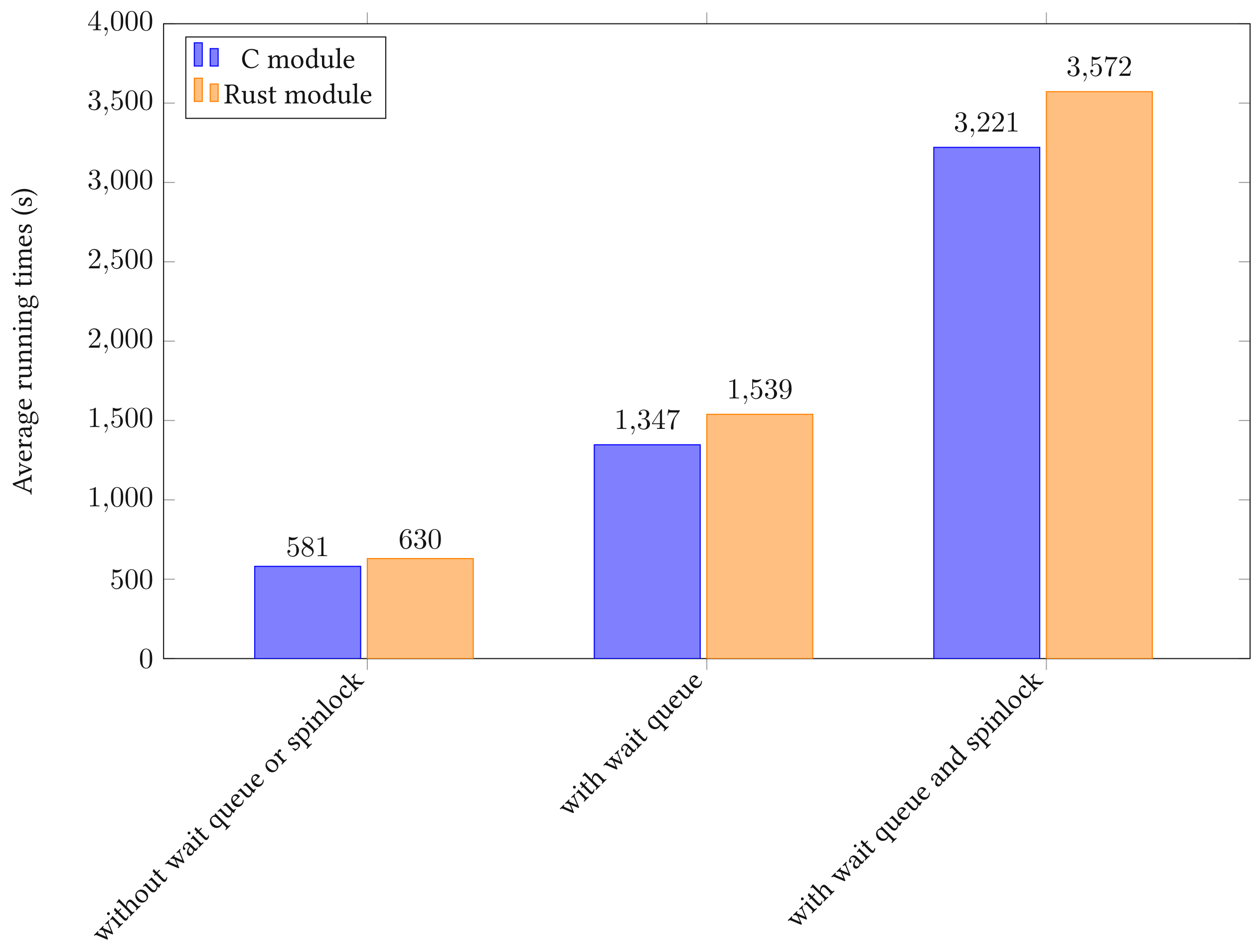

Figure 5.3: Bar chart for average running times of tested modules.

higher than the most complex variant of the C module. The C module variants average power consumption increases with adding a wait queue by 9.5% and with the added spinlock another 7.5%. This leads to an average power consumption of 2.84 W.

On the other hand, the Rust module in its least complex version already has an average power consumption of 3.26 W, but increases only by 2.1% with and added wait queue and by 2.7% with an additional spinlock.

The average power consumption difference between the two modules is for the least complex variant at +35%, for the variant with wait queue +26% and for the variant with the added spinlock and the wait queue +20%.

This shows that the Rust version uses more energy. In addition to the slightly longer running time, this leads to the following differences in energy consumption as shown in Table 5.4. The biggest difference with +46.15% more energy consumption has the basic variant of the module, because the overhead of the rust module has a bigger impact at this level of complexity. The variant with just the

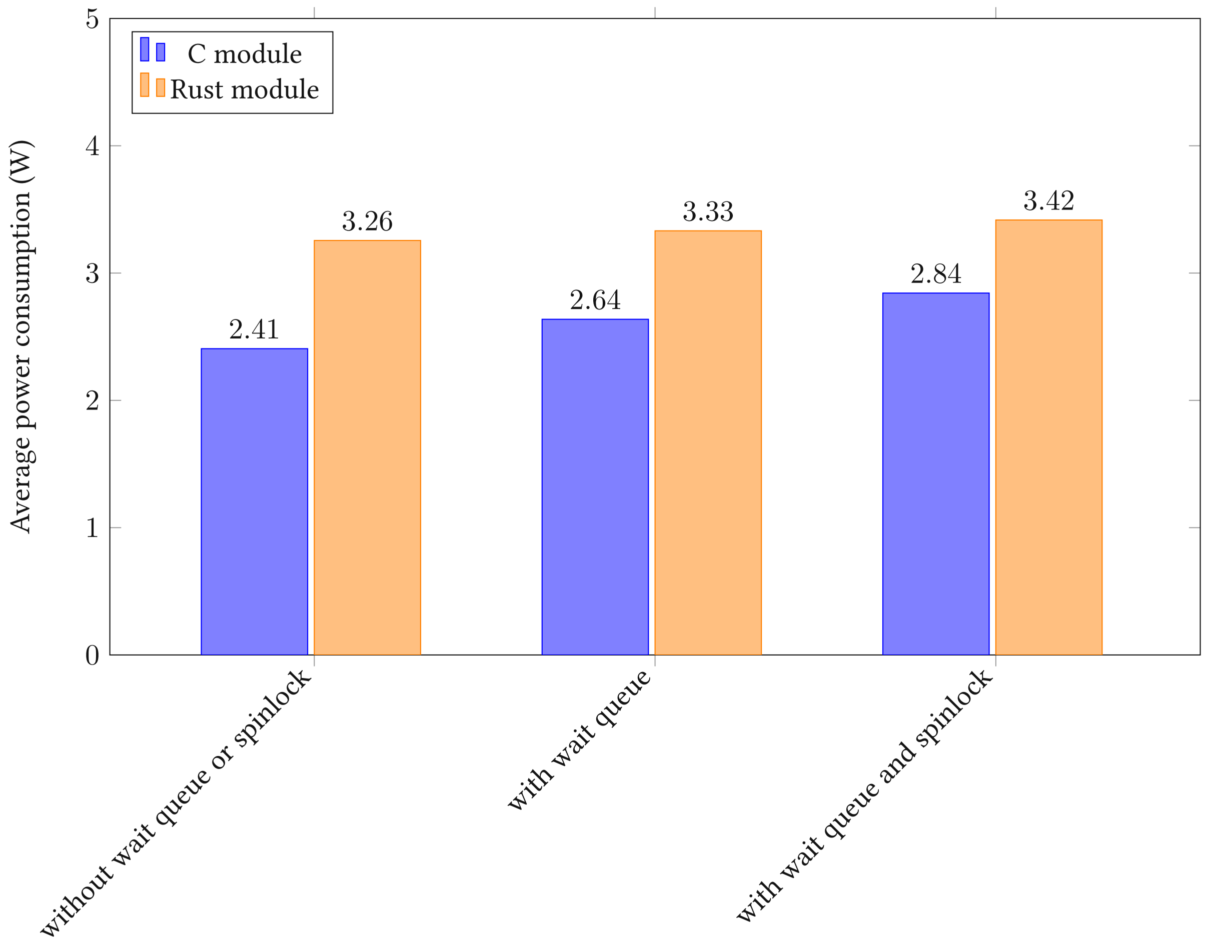

Figure 5.4: Bar chart for average power consumption of tested modules.

wait queue already has a lower difference in energy consumption with +27.68% and the most complex variant has a difference of +33.86% compared to the C module.

In Figure 5.3 the average running times of all the modules and variants are shown and compared. The chart shows that the average running times of the Rust modules are longer than the ones of the C module.

In Figure 5.4 the average power consumption of the modules and variants is compared. The Rust module uses more power than the C module in all the available variants.

# Chapter 6

# Conclusion

The integration of Rust into the Linux kernel marks a significant evolution in the development of operating systems, particularly in enhancing the safety and reliability of device drivers. This thesis explores the comparative performance and energy consumption of Rust-based and C-based virtual GPIO drivers on a Raspberry Pi 4 Model B.

The fundamental role of the Linux kernel in managing system resources and ensuring efficient operation is discussed as well the historical progression of operating systems, kernel architectures, and the recent integration of Rust into the Linux kernel. Rust, known for its memory safety features, is introduced as a promising solution to reduce memory safety vulnerabilities, which are prevalent in programming languages, such as C [59, 60].

The implementation outlines the development of basic kernel modules in both C and Rust, providing a comparative analysis of their structure and generated assembly code. The analysis reveals that Rust introduces additional safety checks and abstractions, resulting in more complex assembly code compared to C. Due to Rusts additional security features, Rust does not compile if there are memory vulnerabilities present. This is different to how the C compiler works. Therefore, developing in Rust provides a more convenient platform, because the errors are already getting caught during the compilation and not during the runtime as in C. This allows for easier and quicker debugging sessions.

The evaluation chapter discusses the results of performance tests, power and energy consumption measurements. The findings show that Rust-based drivers have longer running times and higher energy consumption compared to their C counterparts. Specifically, Rust modules were found to be approximately 8% slower in the basic variant, 11% slower with a wait queue, and 14.3% slower with both a wait queue and a spinlock. The energy consumption of Rust modules is also higher, with the basic variant consuming 46.15% more energy, the variant with a wait queue consuming 27.68% more energy, and the variant with both a wait queue and a spinlock consuming 33.86% more energy. This is partly because of longer running times, but also due to the reason that the power consumption of the Rust module was higher.

At the moment Rust-for-Linux still has some limitations, such as not having access to all important APIs. One example of this is the direct memory access (DMA) layer [61]. Without this, Rust device drivers won't be able to set up memory areas for DMA transfers. As Rust-for-Linux is an actively developed project, it is likely that more and more APIs will be available in the future. This will allow for future works to focus on increasingly more complex device drivers.

# Bibliography

[1] K. N. Billimoria, *Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization.* Packt Publishing, 2021.

[2] A. Gonzalez, D. Mvondo, and Y.-D. Bromberg, “Takeaways of implementing a native rust UDP tunneling network driver in the Linux kernel,” in *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, ser. PLOS ’23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 18–25.

[3] “Rust for linux: Nvme driver,” https://web.archive.org/web/20240719205942/https://rust-for-linux.com/nvme-driver, 2024.

[4] B. Ward, *How Linux works: what every superuser should know*, 2nd ed. No Starch Press, 2014.

[5] D. A. Bader, “Linux and supercomputing: How my passion for building cots systems led to an hpc revolution.” *IEEE Ann. Hist. Comput.*, vol. 43, no. 3, pp. 73–80, 2021.

[6] M. Davis, B. McInnes, and I. Ahmed, “Forensic investigation of instant messaging services on linux os: Discord and slack as case studies,” *Forensic Science International: Digital Investigation*, vol. 42, p. 301401, 2022, proceedings of the Twenty-Second Annual DFRWS USA. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281722000828

[7] D. Zhang, O. R. Gatla, W. Xu, and M. Zheng, “A study of persistent memory bugs in the linux kernel,” in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR ’21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456727.3463783

[8] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, “Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 66–83. [Online]. Available: https://doi.org/10.1145/3477132.3483549

[9] "The rust programming language," https://www.rust-lang.org/, 2023.

[10] S. Klabnik and C. Nichols, *The Rust programming language.* No Starch Press, 2023.

[11] Z. Li, V. Narayanan, X. Chen, J. Zhang, and A. Burtsev, "Rust for Linux: Understanding the Security Impact of Rust in the Linux Kernel," in *2024 Annual Computer Security Applications Conference (ACSAC)*, Dec. 2024, pp. 548–562, iSSN: 2576-9103. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10917595?casa_token=RUfZh8Y5LjsAAAAA:5SMS5CkghOAyRQg0MqaBhP9okYwPzekSXflgg-kVh4FGeApikC9Xa-NnLPXNewIU4TFgpuuhsg

[12] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," *arXiv preprint arXiv:2206.05503*, 2022.

[13] J. V. Stoep, "Google Online Security Blog: Memory Safe Languages in Android 13." [Online]. Available: https://web.archive.org/web/20250313161630/https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[14] P. Eshwarla, *Practical System Programming for Rust Developers: Build Fast and Secure Software for Linux/Unix Systems with the Help of Practical Examples.* Packt Publishing, 2021.

[15] C. S. Frank Vasquez, *Mastering Embedded Linux Programming: Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1 (Dunfell)*, 3rd ed. Packt Publishing, 2021.

[16] "The rfc in the maillist for rust support," https://lore.kernel.org/rust-for-linux/20210414184604.23473-1-ojeda@kernel.org/, 2021.

[17] G. K.-H. Jonathan Corbet, Alessandro Rubini, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005.

[18] Y. Peng, F. Li, and A. Mili, "Modeling the evolution of operating systems: An empirical study," *Journal of Systems and Software*, vol. 80, no. 1, pp. 1–15, Jan. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121206001245

[19] P. J. Denning, W. F. Tichy, E. D. Lazowska, and J. Liedtke, "Operating systems," in *Encyclopedia of Computer Science.* GBR: John Wiley and Sons Ltd., Jan. 2003, pp. 1290–1324.

[20] G. O'Regan, "Overview of Operating Systems," in *World of Computing: A Primer Companion for the Digital Age*, G. O'Regan, Ed. Cham: Springer International Publishing, 2018, pp. 203–215. [Online]. Available: https://doi.org/10.1007/978-3-319-75844-2_10

[21] B. Roch, "Monolithic kernel vs. Microkernel," 2004.

[22] A. S. Tanenbaum and H. Bos, *Modern operating systems*, 4th ed. Boston: Prentice Hall, 2015.

[23] D. Griffiths and D. Makaroff, "Hybrid vs. monolithic OS kernels: a benchmark comparison," in *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, ser. CASCON '06. USA: IBM Corp., Oct. 2006, pp. 30–es. [Online]. Available: https://dl.acm.org/doi/10.1145/1188966.1189007

[24] I. Stankov and G. Spasov, "Discussion of Microkernel and Monolithic Kernel Approaches," *International Scientific Conference Computer Science*, 2006.

[25] S. Biggs, D. Lee, and G. Heiser, "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ser. APSys '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/3265723.3265733

[26] R. I. Mutia, "Inter-Process Communication Mechanism in Monolithic Kernel and Microkernel," 2014.

[27] J. N. Herder, "TOWARDS A TRUE MICROKERNEL OPERATING SYSTEM," Master's thesis, Vrije Universiteit Amsterdam, 2005.

[28] O.-A. Isaac, K. Okokpujie, H. Akinwumi, J. Juwe, H. Otunuya, and O. Alagbe, "An Overview of Microkernel Based Operating Systems," *IOP Conference Series: Materials Science and Engineering*, vol. 1107, no. 1, p. 012052, Apr. 2021, publisher: IOP Publishing. [Online]. Available: https://dx.doi.org/10.1088/1757-899X/1107/1/012052

[29] B. A. A. Abdalkarim and D. Akgun, "Analysis of hybrid kernel-based operating systems," *1st Internation Conference on Innovative Academic Studies*, 2022.

[30] H. Harshvardhan and S. Irabatti, "Study of Kernels in Different Operating Systems in Mobile Devices," *Journal of Online Engineering Education*, vol. 14, no. 1s, pp. 21–26, May 2023, number: 1s. [Online]. Available: https://onlineengineeringeducation.com/index.php/joee/article/view/82

[31] "Fuse," https://web.archive.org/web/20250320192641/https://www.kernel.org/doc/html/latest/filesystems/fuse.html, 2025.

[32] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or Not to {FUSE}: Performance of {User-Space} File Systems," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, 2017, pp. 59–72. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[33] K. N. Billimoria, *Linux Kernel Programming Part 2 - Char Device Drivers and Kernel Synchronization*. Packt Publishing, 2021.

[34] S. Panter and N. Eisty, "Rusty Linux: Advances in Rust for Linux Kernel Development," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 496–502. [Online]. Available: https://dl.acm.org/doi/10.1145/3674805.3690756

[35] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: Association for Computing Machinery, Oct. 2001, pp. 73–88. [Online]. Available: https://dl.acm.org/doi/10.1145/502034.502042

[36] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "MINIX 3: a highly reliable, self-repairing operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 80–89, Jul. 2006. [Online]. Available: https://dl.acm.org/doi/10.1145/1151374.1151391

[37] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[38] A. Kantee, "Flexible operating system internals: the design and implementation of the anykernel and rump kernels," 2012.

[39] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002, pp. 102–107.

[40] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 115–128.

[41] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An empirical study of {Rust-for-Linux}: The success, dissatisfaction, and compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 425–443.

[42] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance analysis of nvme ssds and their implication on real world databases," in *Proceedings of the 8th ACM International Systems and Storage Conference*, 2015, pp. 1–11.

[43] "Rust for linux: Null block driver," https://web.archive.org/web/20240927064645/https://rust-for-linux.com/null-block-driver, 2024.

[44] J. LaCroix, *Mastering Ubuntu Server: Explore the versatile, powerful Linux Server distribution Ubuntu 22.04 with this comprehensive guide.* Packt Publishing Ltd, Sep. 2022, google-Books-ID: kXe-JEAAAQBAJ.

[45] H. D. Ghael, D. L. Solanki, and G. Sahu, “A Review Paper on Raspberry Pi and its Applications,” *International Journal of Advances in Engineering and Management (IJAEM)*, vol. 2, no. 12, 2020.

[46] J. W. Jolles, “Broad-scale applications of the Raspberry Pi: A review and guide for biologists,” *Methods in Ecology and Evolution*, vol. 12, no. 9, pp. 1562–1579, 2021, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/2041-210X.13652. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/2041-210X.13652

[47] V. Gonzalez-Huitron, J. A. León-Borges, A. E. Rodriguez-Mata, L. E. Amabilis-Sosa, B. Ramírez-Pereda, and H. Rodriguez, “Disease detection in tomato leaves via CNN with lightweight architectures implemented in Raspberry Pi 4,” *Computers and Electronics in Agriculture*, vol. 181, p. 105951, Feb. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169920331562

[48] “The linux kernel archives,” https://www.kernel.org/category/releases.html, 2025.

[49] D. M. R. Brian W. Kernighan, *The C programming language.* Prentice Hall, 1988.

[50] O. Hryshchuk and S. Zagorodnyuk, “Managing energy consumption in FPGA-based edgecomputing systems with soft-core CPUs,” *Journal of Edge Computing*, Mar. 2025. [Online]. Available: https://acnsci.org/journal/index.php/jec/article/view/717

[51] J. Pincus and B. Baker, “Beyond stack smashing: recent advances in exploiting buffer overruns,” *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, Jul. 2004, conference Name: IEEE Security & Privacy. [Online]. Available: https://ieeexplore.ieee.org/document/1324594/?arnumber=1324594

[52] S. Balachandran, “General purpose input output (gpio),” *Michigan State University College of Engineering. Published*, pp. 08–11, 2009.

[53] M. Maksimović, V. Vujović, N. Davidović, V. Milošević, and B. Perišić, “Raspberry pi as internet of things hardware: performances and constraints,” *design issues*, vol. 3, no. 8, pp. 1–6, 2014.

[54] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont, “The lock holder and the lock waiter pre-emption problems: Nip them in the bud using informed spinlocks (i-spinlock),” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 286–297.

[55] A. Inc, *AUUG Conference Proceedings.* AUUG, Inc., 2001, google-Books-ID: xPe8dPxcU98C.

[56] J. Madieu, *Linux Device Driver Development: Everything you need to start with device driver development for Linux kernel and embedded Linux*, 2nd ed. Packt Publishing, 2022.

[57] X. Ji, X. Zhou, M. Xu, W. Xu, and Y. Dong, "Opcio: Optimizing power consumption for embedded devices via gpio configuration," *ACM Transactions on Sensor Networks (TOSN)*, vol. 16, no. 2, pp. 1–28, 2020.

[58] "Resistance to rust abstractions for dma mapping," https://web.archive.org/web/20250602200637/https://en.wikipedia.org/wiki/Sample_size_determination, 2025.

[59] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 763–779.

[60] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state and open problems," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2296–2310. [Online]. Available: https://doi.org/10.1145/3658644.3690275

[61] "Resistance to rust abstractions for dma mapping," https://web.archive.org/web/20250415172057/https://lwn.net/Articles/1006805/, 2025.

# Appendix A

Listing A.1: vgpio_rust_wq_sl.rs: Rust module with wait queue and spinlock

```
// SPDX-License-Identifier: GPL-2.0

//! A simple Rust character device using the C API.

#![allow(missing_docs)]

use kernel::prelude::*;
use kernel::bindings;
use kernel::c_str;
use kernel::error::code;
use core::ffi::{c_int, c_long};
use core::marker::PhantomData;
use kernel::sync::*;
use kernel::pin_init;
use core::sync::atomic::{AtomicBool, Ordering};

module! {
    type: VgpioRust,
    name: "vgpio_rust",
    author: "Fabian T Garber",
    description: "A virtual Rust GPIO module",
    license: "GPL",
}

const DEVICE_NAME: &CStr = c_str!("vgpio_rust");
const CLASS_NAME: &CStr = c_str!("vgpio");

const GPIO_SET_VALUE: u32 = 0x40086701;
const GPIO_GET_VALUE: u32 = 0x80086702;

#[repr(C)]
struct GpioData {
    pin: c_int,
    value: c_int,
}

struct VgpioInner {
    pins: [c_int; 8],
}

#[pin_data]
struct VgpioData {
    c: u32,
    #[pin]
    vgpio: SpinLock<VgpioInner>,
}

impl VgpioData {
    fn new() -> impl PinInit<Self> {
        pin_init!(Self {
            c: 0,
            vgpio: new_spinlock!(VgpioInner { pins: [0; 8] }),
        })
    }
}

static mut VGPIO_DATA: Option<Pin<Box<VgpioData>>> = None;

static GPIO_CHANGED: AtomicBool = AtomicBool::new(false);

static mut GPIO_WAITQUEUE: bindings::wait_queue_head_t =
bindings::wait_queue_head_t {
    lock: core::mem::MaybeUninit::uninit(),
    task_list: kernel::bindings::list_head {
        prev: core::ptr::null_mut(),
        next: core::ptr::null_mut(),
    },
};

#[repr(transparent)]
struct VgpioFops(kernel::bindings::file_operations);
unsafe impl Sync for VgpioFops {}

static VGPIO_FOPS: VgpioFops = VgpioFops(kernel::bindings::file_operations {
    owner: core::ptr::null_mut(),
    open: Some(vgpio_open),
    release: Some(vgpio_release),
    read: Some(vgpio_read as unsafe extern "C" fn(_, _, _, _) -> _),
    unlocked_ioctl: Some(vgpio_ioctl),
    write: None,
    llseek: None,
    poll: None,
    mmap: None,
    flush: None,
    ..unsafe { core::mem::zeroed() }
});

struct VgpioRust {
    major: i32,
    class: *mut bindings::class,
    device: *mut bindings::device,
    _marker: PhantomData<*mut ()>,
}

unsafe impl Send for VgpioRust {}
unsafe impl Sync for VgpioRust {}

impl kernel::Module for VgpioRust {
    fn init(_module: &'static ThisModule) -> Result<Self> {
        pr_info!("vgpio_rust: module loaded.\n");

        unsafe {
            VGPIO_DATA = Some(Box::pin(VgpioData::new()));
        }

        unsafe {
            bindings::init_waitqueue_head(&mut GPIO_WAITQUEUE);
        }

        let major = unsafe {
            bindings::__register_chrdev(
                0,
                0,
                1,
                DEVICE_NAME.as_char_ptr(),
                &VGPIO_FOPS.0,
            )
        };
        if major < 0 {
            pr_err!("vgpio_rust: failed to register device: {}\n", major);
            return Err(code::EINVAL.into());
        }

        let class = unsafe { bindings::class_create(CLASS_NAME.as_char_ptr()) };
        if class.is_null() {
            pr_err!("vgpio_rust: failed to create class\n");
            unsafe {
                bindings::__unregister_chrdev(major as u32, 0, 1,
                DEVICE_NAME.as_char_ptr());
            }
            return Err(code::EINVAL.into());
        }

        let dev = ((major as u32) << 20) | 0;
        let device = unsafe {
            bindings::device_create(
                class,
                core::ptr::null_mut(),
                dev,
                core::ptr::null_mut(),
                DEVICE_NAME.as_char_ptr(),
            )
        };
        if device.is_null() {
            pr_err!("vgpio_rust: failed to create device\n");
            unsafe {
                bindings::class_destroy(class);
                bindings::__unregister_chrdev(major as u32, 0, 1,
                DEVICE_NAME.as_char_ptr());
            }
            return Err(code::EINVAL.into());
        }

        pr_info!("vgpio_rust: registered character device under
        /dev/vgpio_rust.\n");

        Ok(VgpioRust {
            major,
            class,
            device,
            _marker: PhantomData,
        })
    }
}

impl Drop for VgpioRust {
    fn drop(&mut self) {
        let dev = ((self.major as u32) << 20) | 0;
        unsafe {
            if !self.device.is_null() {
                bindings::device_destroy(self.class, dev);
            }
            if !self.class.is_null() {
                bindings::class_destroy(self.class);
            }
            bindings::__unregister_chrdev(self.major as u32, 0, 1,
            DEVICE_NAME.as_char_ptr());
        }
        pr_info!("vgpio_rust: module unloaded.\n");
    }
}

#[no_mangle]
pub extern "C" fn vgpio_open(
    _inode: *mut bindings::inode,
    _file: *mut bindings::file,
) -> c_int {
    pr_info!("vgpio_rust: device opened.\n");
    0
}

#[no_mangle]
pub extern "C" fn vgpio_release(
    _inode: *mut bindings::inode,
    _file: *mut bindings::file,
) -> c_int {
    pr_info!("vgpio_rust: device released.\n");
    0
}

#[no_mangle]
pub extern "C" fn vgpio_read(
    _file: *mut bindings::file,
    buf: *mut u8,
    count: usize,
    pos: *mut bindings::loff_t,
) -> isize {
    pr_info!("vgpio_rust: read called with count={}\n", count);

    let ret = unsafe {
        bindings::wait_event_interruptible(
            &mut GPIO_WAITQUEUE,
            GPIO_CHANGED.load(Ordering::SeqCst),
        )
    };
    if ret != 0 {
        pr_info!("vgpio_rust: read interrupted by signal\n");
        return -bindings::EINTR as isize;
    }

    GPIO_CHANGED.store(false, Ordering::SeqCst);

    let msg = b"GPIO state changed\n";
    let len = msg.len();

    let offset = unsafe { *pos as usize };

    if offset >= len {
        return 0;
    }

    let bytes_left = len - offset;
    let to_copy = count.min(bytes_left);

    if buf.is_null() {
        pr_err!("vgpio_rust: error: buf is null!\n");
        return -(bindings::EFAULT as isize);
    }

    let res = unsafe {
        bindings::copy_to_user(
            buf as *mut core::ffi::c_void,
            msg[offset..offset + to_copy].as_ptr() as *const core::ffi::c_void,
            to_copy as u64,
        )
    };

    if res != 0 {
        pr_err!("vgpio_rust: error: copy_to_user() failed with {}\n", res);
        return -(bindings::EFAULT as isize);
    }

    unsafe {
        *pos += to_copy as i64;
    }

    to_copy as isize
}
```

Listing A.2: vgpio_c.h: C module header file

```c
#ifndef VGPIOC_H
#define VGPIOC_H

#include <linux/ioctl.h>

#define MAJOR_NUM 158

struct gpio_data {
        int pin;
        int value;
};

#define GPIO_SET_VALUE _IOW(MAJOR_NUM, 0, struct gpio_data)
#define DEVICE_NAME "vgpio_c"
#define DEVICE_PATH "/dev/vgpio_c"

#endif
```

Listing A.3: vgpio_c_wq_sl.c: C module with wait queue and spinlock

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/spinlock.h>
#include <linux/wait.h>
#include <linux/version.h>
#include <linux/types.h>
#include <linux/printk.h>

#include "vgpio_c.h"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Fabian T Garber");
MODULE_DESCRIPTION("Virtual GPIO Driver");
MODULE_VERSION("0.7");

#define NUM_GPIO_PINS 8  // Number of virtual GPIOs

#define GPIO_SET_VALUE _IOW(MAJOR_NUM, 0, struct gpio_data)
#define GPIO_GET_VALUE _IOR(MAJOR_NUM, 1, struct gpio_data)

// Debug flag (default: 0)
static int debug = 0;
module_param(debug, int, 0644);
MODULE_PARM_DESC(debug, "Enable debug output (default: 0)");

static struct class *vgpio_class = NULL;
//static struct device *vgpio_device = NULL;
static spinlock_t gpio_lock;
static wait_queue_head_t gpio_wait_queue;
static bool gpio_values[NUM_GPIO_PINS] = {0};
static bool gpio_changed = false;

// Device Open
static int device_open(struct inode *inode, struct file *file)
{
    dev_info(dev, "Virtual GPIO device opened\n");
    try_module_get(THIS_MODULE);
    return 0;
}

// Device Close
static int device_release(struct inode *inode, struct file *file)
{
    dev_info(dev, "Virtual GPIO device closed\n");
    module_put(THIS_MODULE);
    return 0;
}

// Device IOCTL
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct gpio_data data;

    switch (cmd) {
        case GPIO_SET_VALUE:
            if (copy_from_user(&data, (struct gpio_data __user *)arg, sizeof(data)))
                return -EFAULT;
            if (data.pin < 0 || data.pin >= NUM_GPIO_PINS)
                return -EINVAL;

            spin_lock(&gpio_lock);
            gpio_values[data.pin] = (bool)data.value;
            gpio_changed = true;
            spin_unlock(&gpio_lock);
            wake_up_interruptible(&gpio_wait_queue);

            if (debug)
                dev_info(dev, "GPIO[%d] set to %d\n", data.pin, data.value);
            break;

        case GPIO_GET_VALUE:
            if (copy_from_user(&data, (struct gpio_data __user *)arg, sizeof(data)))
                return -EFAULT;
            if (data.pin < 0 || data.pin >= NUM_GPIO_PINS)
                return -EINVAL;

            spin_lock(&gpio_lock);
            data.value = gpio_values[data.pin];
            spin_unlock(&gpio_lock);

            if (copy_to_user((struct gpio_data __user *)arg, &data, sizeof(data)))
                return -EFAULT;
            break;

        default:
            return -EINVAL;
    }
    return 0;
}

static ssize_t device_read(struct file *file, char __user *buf,
size_t len, loff_t *offset)
{
    if (wait_event_interruptible(gpio_wait_queue, gpio_changed))
        return -ERESTARTSYS; // If interrupted

    gpio_changed = false;
    char data = '1';
    if (copy_to_user(buf, &data, 1))
        return -EFAULT;

    return 1;
}

// File Operations
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,
    .read = device_read,
};

// Module Init
static int __init virtual_gpio_init(void)
{
    int ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &fops);

    spin_lock_init(&gpio_lock);
    init_waitqueue_head(&gpio_wait_queue);

    if (ret < 0) {
        dev_alert(dev, "%s failed with %d\n",
                 "Sorry, registering the character device ", ret);
        return ret;
    }

    vgpio_class = class_create(DEVICE_NAME);

    device_create(vgpio_class, NULL, MKDEV(MAJOR_NUM, 0), NULL, DEVICE_NAME);

    dev_info(dev, "Virtual GPIO driver loaded\n");
    return 0;
}

// Module Exit
static void __exit virtual_gpio_exit(void)
{
    device_destroy(vgpio_class, MKDEV(MAJOR_NUM, 0));
    class_destroy(vgpio_class);
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
    pr_info("Virtual GPIO driver unloaded\n");
}

module_init(virtual_gpio_init);
module_exit(virtual_gpio_exit);
```