BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

University of Zurich^UZH

Linux on Tensilica Xtensa

Gregory Frommelt
Zürich, Switzerland
Student ID: 15-922-909

Supervisor: Dr. Eryk Schiller, Chao Feng
Date of Submission: September 6, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Das Internet der Dinge (Internet-of-Things, IoT) wird zu einem immer wichtigeren Bestandteil des modernen Lebens. Es bietet eine breite Palette von Anwendungen auf verschiedenen Geräten, die eine Vielzahl von unterschiedlichen Betriebssystemen (OS) verwenden. Während Linux das am weitesten verbreitete Betriebssystem in der IoT Landschaft ist, verhindert sein Ressourcenbedarf oft den Einsatz auf weniger leistungsfähigen, kostengünstigen Geräten wie denen der ESP32 Familie von Mikrocontrollern (MCU). Das Ziel dieser Arbeit ist es daher zu untersuchen, inwiefern Linux auf Geräte der ESP32 Familie portiert werden kann, was sowohl einen wirtschaftlichen Anreiz, als auch einen in puncto IoT Standardisierung hat. Eine Kostenanalyse zeigt, dass die Verwendung des ESP32-WROVER-IE Moduls im Vergleich zum Raspberry Pi Zero W eine Kostenreduktion von ca. 80% mit sich bringt. Es wurde eine Toolchain konstruiert, mit derer Hilfe ein Linux Kernel image kompiliert werden konnte, welches anschliessend erfolgreich auf ein ESP32-S3-DevKitC-1 Board portiert wurde. Erste Auswertungen deuten darauf hin, dass das portierte System grundlegende Funktionalität bietet, die für IoT Aufgaben geeignet sind, jedoch bestimmte Limitierungen derzeit seinen praktischen Nutzen einschränken.

The Internet-of-Things (IoT) is becoming increasingly integral to modern living, offering a wide range of applications across diverse devices employing a multitude of different operating systems (OS). While Linux is the most prevalent OS in the IoT landscape, its resource requirements often prevent its use on less powerful, cost-efficient devices like those in the ESP32 family of microcontrollers (MCUs). The goal of this thesis is therefore to explore the feasibility of porting Linux to ESP32 devices, motivated by both economic and IoT standardization incentives. A cost analysis reveals an approximately 80% reduction in expenses when using the ESP32-WROVER-IE module compared to the Raspberry Pi Zero W. A toolchain was constructed to compile a Linux kernel image, which was successfully ported to an ESP32-S3-DevKitC-1 board. Initial evaluations indicate that the ported system offers basic functionality suitable for IoT tasks, although certain limitations currently restrict its practical utility.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Eryk Schiller and Chao Feng, for their invaluable guidance, unwavering support and constructive feedback throughout the course if this research. Their expertise and insights have been instrumental in shaping this work and I am profoundly grateful for the opportunity to learn under their tutelage.

Furthermore, I would like to extend my sincere thanks to Prof. Dr. Burkhard Stiller, head of the Communication System Research Group (CSG) at the University of Zurich (UZH), for providing me with the opportunity to delve into such an exciting subject and thereby enriching my personal academic expertise.

Special recognition is also owed to M. Filippov, whose pioneering work in this field greatly influenced the direction of my research. The advice and ideas provided through our correspondence were critical to the development and implementation of the methodologies used in this study.

Lastly, I want to thank my family and friends, in particular S. Scheuss, N. Peyer, J. Weber and R. Abraham-Schmitz, for their support and encouragement throughout this time.

Contents

Chapter 1

Introduction

In an era defined by hyperconnectivity, the Internet-of-Things (IoT) has risen as a transformative force, bridging the physical and digital domains. With billions of devices [1] globally communicating vast amounts of data, IoT is no longer a future prediction but an existing reality. It seamlessly integrates into various sectors, from healthcare to transportation [2], bringing forth both opportunities and challenges *e.g.*, the establishment of security and enabling means of standardization [3] of these resource-constrained devices that form the backbone of the IoT ecosystem. This chapter delves into the significance of IoT as an essential component of modern living, setting the stage for a deeper exploration of IoT technology and the underlying motivation driving this thesis.

1.1 Overview

Traditionally, devices such as the Raspberry Pi (RPI) family have exemplified the possibility of running Linux on microcontroller units (MCUs) [4]. The Raspberry Pi 3 Model B for example is a small computer equipped with the Advanced RISC Machines (ARM) Cortex-A Central Processing Unit (CPU) and Random Access Memory (RAM). Instead of a Hard Disk Drive (HDD) it employs flash memory on which an operating system (OS) can be installed. It also features Universal Serial Bus (USB) connectors, a video output and a Wireless Fidelity (WiFi) adapter [5]. Integration of Linux with these devices has not only expanded their capabilities but has also simplified the developmental process for microcontroller applications as developers and users tread familiar territory with Linux based operating systems.

As the technological landscape continues to evolve, newer microcontroller families like Espressif's ESP32 family are emerging, endowed with features such as the dual-core Tensilica LX6 and LX7 processors with the Reduced Instruction Set Computer (RISC) based Xtensa Instruction Set Architecture (ISA) [6]. The ESP32's hardware specifications are reminiscent of early computers that successfully ran Linux [7]. Linux was first developed for the Complex Instruction Set Computer (CISC) based Intel 386 (i386) [8] in 1991. At the time the CPU clock frequency did not exceed 40 MHz, while the typical computer was running with a mere single-digit number of Megabytes (MB) of RAM, *e.g.*, 4 MB. In comparison, the Espressif's ESP32-WROVER-IE MCU module, featuring the ESP32-D0WD-V3 System-on-a-Chip (SoC) that employs two Tensilica Xtensa 32-bit LX6 microprocessors, has an adjustable clock frequency from 80 MHz to 240 MHz. The module is available with configurations of up to 8 MB of Pseudo-Static Random Access Memory (PSRAM) and 16 MB of flash memory [9]. This indicates the possibility of running a Linux kernel with an appropriately small embedded C library such as uClibc-ng [10] and the necessary binary utilities.

1.2 Motivation

The RPI 3 Model B is available from around 30 CHF [11] whereas development boards for the ESP32-WROVER-IE are available for less than 10 CHF [12]. However, the sole ESP32-WROVER-IE module - decoupled from a development board - can be obtained for less than 3 CHF [13]. The RPI Zero W & WH development board series' prices start from around 14 CHF [14], hence, the cost reduction from the RPI 3 B to the RPI Zero W/WH, capable of running a Linux kernel, is already around 53%. The cost reduction from the RPI Zero W/WH to a development board for the ESP32-WROVER-IE would be another approximately 35%, when comparing with the sole ESP32-WROVER-IE module, the reduction drops even further to around 80%. For the sake of completeness regarding the approach taken in this thesis may here also be mentioned that prices of Espressif's newer ESP32-S3 series are comparable with the ones of the ESP32-WROVER-IE, *e.g.*, the ESP32-S3-WROOM-1-N8R8 module, equipped with 8 MB of flash memory and 8 MB of PSRAM costs around 3 CHF [15].

The economic incentives are evident; transitioning from devices like the RPI 3B to the ESP32 family could lead to substantial cost reductions. Moreover, enabling ESP32 devices to run Linux not only offers a cost-efficient alternative but also benefits from the robustness and extensive documentation of a well-established OS. This dual advantage has the potential to both significantly reduce overall expenses and advance the means of standardization within the IoT landscape.

1.3 Applicability Scope: From IoT to NFTs

Enabling a Linux kernel to run on an embedded device offers a technical advancement in many aspects, one of them being the kernel's open-source nature, essentially providing the means of a large community. Nonetheless, without dismissing the utility of Linux in a standalone system, its capabilities can be further exploited by integrating it into a connected environment. Given the crucial matter of the IoT this thesis surrounds, the primary value of operating Linux on an ESP32 device is derived from its ability to function as a linked entity within a networked domain.

The global connectivity facilitated by the internet introduces a myriad of potential application scenarios for ESP32 IoT devices, expanding the horizon of possibilities even further when endowed with the means of Linux. One niche yet significant application lies within the realm of art — specifically, in the secure transportation and real-time tracking of artworks. It is desirable to monitor a variety of factors – such as vibration, impact forces, and environmental conditions like temperature and humidity – to ensure the art's preservation during transit [16].

[17] addresses precisely these difficulties involved in securely and transparently tracking and monitoring artworks during transportation. His work involves the integration of Non-Fungible Tokens (NFTs) and IoT devices, utilizing blockchain technology to register and monitor environmental parameters. The system he proposes aims to ensure the integrity and authenticity of the artwork, relying on IoT devices for real-time tracking and environmental monitoring. This is where the present thesis intersects with his research. By enabling Linux to operate on WiFi-capable ESP32 devices, this thesis provides the potential for a cost-effective yet functional hardware solution for the kind of secure and transparent tracking system described in his work.

1.4 Thesis Goals & Description of Work

Given the potential economical benefits and the associated applicability range of running a Linux kernel on cheap devices such as Espressif's ESP32 device family, as a preliminary goal, this thesis aims to analyse and devise an approach to this integration. Building upon a working example, in a subsequent step, the aim is further oriented around evaluating the functional prototype in terms of various performance metrics and practicality in regards to IoT-related use cases as outlined in 1.3.

In accordance with the thesis goals, this thesis adopts a methodological approach based on experimental prototyping and evaluation. The first pursued examination revolves around both the hardware and software facets provided by Espressif, encompassing the ESP32 and ESP32-S3 series of their ESP32 family of MCUs accompanied by their Espressif IoT Development Framework (ESP-IDF). Alongside, a parallel study focuses on the roles of relevant components within the development ecosystem, including toolchains, the Linux kernel source, and auxiliary tools such as Buildroot and BusyBox. With a foundational understanding in place, focus is shifted towards existing attempts of implementing Linux on MCUs of the aforementioned device family, aiming to comprehend and elaborate on the design choices made. This leads into a showcase of a functional Linux implementation, ultimately providing an opportunity to assess the system's effectiveness and suitability for its intended applications.

1.5 Thesis Outline

In alignment to the methodological approach described in the preceding section, this thesis spans over seven chapters. Chapter 2 is devoted to findings of external research related to the matter at hand, thereby addressing the significance of IoT in the first section, further transitioning to the second section that revolves around the role of Linux within this context. Chapter 3 is dedicated to establish background knowledge vital for the remainder of this thesis, including underlying technical details and concepts, specifics about the Linux kernel and relevant tools and frameworks. Chapter 4 outlines the design choices made for the subsequent implementation; the subject of the MCU is addressed, a concept for program execution introduced, on memory management elaborated and the executable file format explained. Chapter 5 presents the implementation, encompassing the necessary steps to reproduce a prototype running Linux. Chapter 6 deals with the evaluation of the preceding implementation, therein displaying general observations, time measurements and the prototype's WiFi capabilities. The thesis concludes in chapter 7 by assessing the status quo and providing an outlook on further research based on the findings presented herein.

Chapter 2

Related Work

In the following chapter, relevant work in the domains of the IoT and embedded systems, particularly focusing on Linux-based solutions, will be presented. The objective is to identify the current state of the art, understand the challenges and opportunities in the field, and establish the context within which this thesis contributes. This chapter outlines the influence of IoT and the role of OSes like Linux in this context. It concludes by focusing on the application of Linux on the ESP32 series of microcontrollers, which serves as the cornerstone of this thesis.

2.1 The IoT: Importance and Diversity

The IoT has cemented its position in such a substantial role within the modern era that it has become an indispensable cornerstone of the 21st century. The extensive scope of its applications spans a broad spectrum of conceivable areas, ranging from healthcare and mobility to home automation and environmental monitoring, thereby introducing the necessity for smart solutions along the way. Advances in IoT have significantly simplified or even automated numerous tasks, serving as the backbone for data-driven decision-making and resource optimization [18]. Yet, by narrowing the focus of the metaphorical magnifying glass, already the individual's gain in user experiences can be observed in actions that seamlessly integrate into daily life, such as biometrical mechanisms [19].

The diversity of areas in which IoT is assuming a key role appears limitless. However, this term *diversity* is not solely tied to the wide array of IoT applications; it also extends to the underlying technical aspects, including the multitude of OSes available for embedded design. Various Real-Time Operating Systems (RTOS) like FreeRTOS, ChibiOS/RT, and RIOT are commonly employed, to name a few examples. In addition, other specialized operating systems such as Contiki and Tiny OS are also utilized in the embedded realm [20]. Notably, Linux, with its monolithic kernel architecture, has found applicability in this diverse landscape as well.

2.2 Linux in the IoT

Using Linux for embedded development is beneficial for several reasons. It supports a broad range of hardware architectures, including x86, Alpha, Sparc, ARM, Xtensa and many more. This allows for flexibility in design and system architecture. Furthermore, the open-source nature of Linux allows for tailored customization; the kernel can be modified in order to meet specific needs, thereby providing economical advantages [21]. Linux also shines when it comes to performance-intensive tasks such as deep learning and big data [4]. According to [22] is Linux employed as OS by more than 70% of IoT devices. Consequently, a lot of research utilizing embedded Linux can be found, *e.g.*, [23] leverage Linux running on a RPI model 3B+ to illustrate the potential for enhancing manufacturing logistics through advanced tracking and monitoring systems; [24] developed a system for real-time cloud monitoring of a decentralized solar plant with the use of Linux on a RPI model 2B; [25] discussed the development of a multi-protocol IoT gateway for smart buildings, using a wireless network. The latter employs various different MCUs, one of them being the CubieTruck SBC running on embedded Linux and interestingly enough also a FreeRTOS enabled ESP32 device.

Although Linux is widely used in the domain of embedded design, the only research that could be found suggesting its integration on devices of the ESP32 family is the one this thesis builds upon [26]. Instead, besides the aforementioned example, there exists further research utilizing ESP32 devices by the means of FreeRTOS [27], [28]. Therefore, the opportunity to equip these cost-efficient devices with a well-established OS such as Linux and at the same time advance standardization in the IoT field is yet to be explored.

Chapter 3

Background

This chapter serves the purpose of providing the reader with comprehensive knowledge about specific aspects in the field where this thesis is situated. Topics range from underlying hardware-related technical details to software-related matters, including information about essential tools for building the Linux kernel, which also makes appearance herein. Furthermore, the framework upon which the final step in the development process will be based is introduced.

3.1 Harvard architecture

The term *Harvard architecture* has undergone various interpretations over time. Initially, the term was applied to machines designed by and also for the Harvard Computing Laboratory (HCL). These machines had completely separate memories for instructions and data. In the 1970s, the term was formally coined. In the context of designing the first microcontroller, it was used to describe a complete computing device on a single chip still with separate memories for instructions and data. Later on, the term was applied to RISC processors that had separate caches for instructions and data but not necessarily concomitant separation of physical memory [29]. As stated in 1.1, the LX6 and LX7 cores employed in the ESP32 and ESP32-S3 devices are based on RISC processors and inherently follow a Harvard architecture. However, the external memory, *i.e.*, the PSRAM, is not physically separated. Figure 3.1 shows the dual-core-shared ICache and DCache nature of the ESP32-S3 SoC.

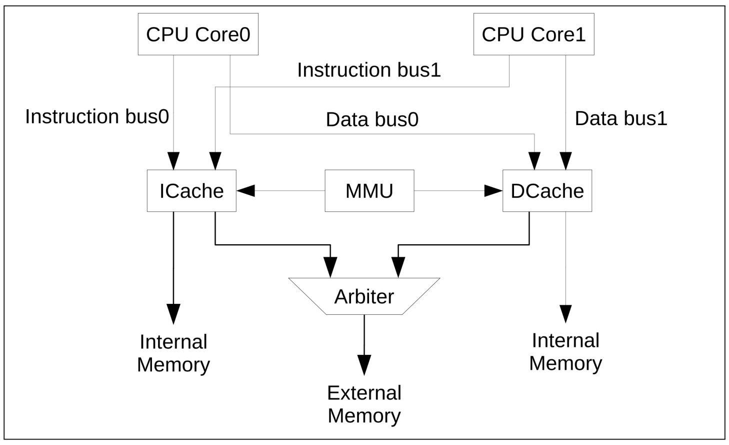


Figure 3.1: ESP32-S3 Harvard architecture structure [30, p. 394]

3.2 Toolchain

In order to fully grasp the concept of a *toolchain* within the domain of embedded systems development, one must initially understand the notion of cross-compilation as it pertains to this specialized field. A compiled executable program, including complex binaries such as the Linux kernel, can be fundamentally understood as a product facilitated through the utilization of a compiler tailored to function in alignment with a designated ISA, such as the Xtensa ISA. A particular challenge emerges when considering the constrained environment of embedded devices, which usually lack the resources needed to utilize the compiler directly. Addressing this issue necessitates the employment of a technique known as cross-compilation. This process involves compiling a binary on a powerful machine (host machine) utilizing a compiler that has been configured for the embedded device (target device) and its specific ISA, which usually differs from that of the host machine's. Following this compilation process, the resulting binary can be transferred to the target device and executed. Though this approach circumvents the challenges tied to resource limitations, it also introduces the need for tools tailored to function with the target ISA, adding a layer of complexity.

Such tools are part of what is referred to as a toolchain in the context of embedded systems development. A toolchain is a comprehensive suite of development tools that encompasses a wide array of software development processes, including the compilation, assembly, linking, and debugging of code. These tools collectively facilitate the transformation of source code into executable machine code that can run on the hardware platform, forming the essential bridge between the high-level programming language and the specific architecture of the target hardware. In this way, the toolchain serves as both a practical solution to the problem of resource constraints and an integral component of the development process for embedded applications.

Within the toolchain, various components play distinct roles. In the case of Linux kernel compilation, these are [31]:

- **Compiler**: This tool translates high-level programming lanugage (*e.g.*, C) into archi tecture-specific assembly language. For example, a C compiler configured for the Xtensa architecture produces Xtensa assembly code. The compiler used in this thesis is the C compiler from the GNU Compiler Collection (GCC). Even though the GCC consists of several compilers for different high-level programming languages besides C (*e.g.*, C++, Java, Fortran, Go, etc.), in the remainder of this thesis by GCC will be referred to as the C compiler.
- **Assembler**: The assembler takes the assembly language code and converts it into relocatable machine code, represented in object files. The assembler used in this thesis is part of GNU's Binutils, a set of binary utilities.
- **Linker**: The linker connects the object files produced by the assembler together, resolving references between them and generating the final executable file. As with the assembler, the linker used in this thesis is also part of GNU's Binutils.
- **C library**: This is a collection of precompiled routines that a program can utilize. They are crucial for embedded development as they provide commonly used functions without the need for re-writing them. In this thesis, the used library is uClibc-ng [10], a small C library designed specifically for embedded system development.
- **Linux Kernel Headers**: Kernel headers are a collection of C header files used in the Linux kernel, providing the necessary definitions and structures to allow user space applications to interface with the kernel. They define the Application Binary Interface (ABI) and are essential for building programs that interact with the kernel.
- **Debugger**: A debugger is often also seen as part of the toolchain, even though it is not essential. The debugger fitting the aforementioned tools would be the GNU Debugger (GDB), however, the GDB will be omitted from the toolchain used in this thesis.

Basically, such a toolchain can be re-used for a variety of devices, as long as it is tailored for the particular environment. The term *environment* in this context is understood to mean compatibility with the host machine or rather its ISA, the fact that it was compiled for the correct target ISA, and the library (*i.e.*, C library) with which it was compiled. Thus, for example, pre-built toolchains can also be found that have been compiled specifically for target machines with the Xtensa ISA [32]. In the course of this thesis, however, it has become apparent that the factors mentioned above do not exclusively determine whether a toolchain can be used. The specific difficulties encountered in this context and the approaches to solving them are presented in more detail in Chapter 4 and 5. In such cases, it is necessary to compile the toolchain oneself.

3.3 Crosstool-NG

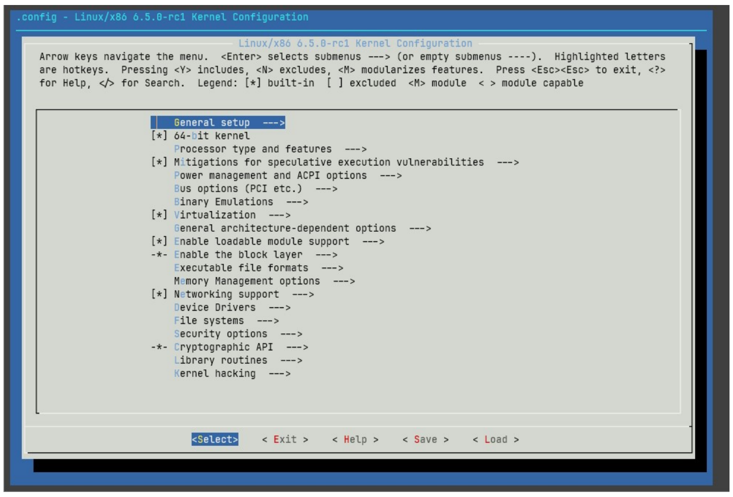
Crosstool-NG offers a framework for toolchain construction. Unlike other available cross-toolchain solutions, which are often restricted by limitations such as sub-optimal configurations for specialized hardware or the use of outdated components, crosstool-NG aims to offer a more flexible and up-to-date alternative. In contrast to other tools designed for broader purposes, like Buildroot, which primarily focus on the construction of a kernel image and the root filesystem (rootfs), crosstool-NG narrows its focus strictly to the generation of toolchains. With the need for customization and optimization in the toolchain construction, crosstool-ng acts as a modular piece of software, enabling the user to decouple those necessary adjustments from the bigger build tool. This consequently makes the development process easier.

3.4 Linux kernel

Linux is a key member of the Unix-like family of OSes, initially developed by Linus Torvalds in 1991 for IBM-compatible personal computers. Unlike many commercial counterparts, Linux is open-source and licensed under the GNU General Public License (GPL), offering free access to its source code. Over the years, Linux has adapted to multiple hardware architectures, and it continues to be driven by a global development community coordinated by Torvalds [33, pp. 1–2].

Building on Linux's adaptability to different hardware architectures, this research specifically employs a variant of the Linux kernel – the linux-xtensa port [34]. This is a kernel source fork, designed specifically for the Xtensa architecture and as such serves as the basis for this thesis. Configuring and building the kernel relies on the GNU Make language [35] which is used from within the kernel source's root directory.

[36, pp. 17–21, 28] elaborates on the multiple methods available for configuring the kernel. The simplest is the `make config` command, which prompts the user to individually confirm or deny each configuration option. Given that the number of options exceeds a thousand, this method is seldom practical for most development scenarios. An alternative approach is to use a pre-existing default configuration. These configurations are located in the Linux kernel source – depending on the architecture – at /arch/ARCH/configs, where ARCH denotes the specified architecture. For instance, the `make ARCH=xtensa defconfig` command applies the configuration located in /arch/xtensa/defconfig, if the defconfig file exists. The chosen settings are then stored in a .config file in the source directory, allowing for additional adjustments through the `make menuconfig` command. This command provides a user-friendly, menu-based interface for further customization as can be seen in Figure 3.2. Once the kernel is configured with the desired options, the build process can be initiated by simply using the `make` command without any parameters.



```
.config - Linux/x86 6.5.0-rc1 Kernel Configuration
                         Linux/x86 6.5.0-rc1 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters
  are hotkeys.  Pressing <Y> includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                   General setup  --->
               [*] 64-bit kernel
                   Processor type and features  --->
               [*] Mitigations for speculative execution vulnerabilities  --->
                   Power management and ACPI options  --->
                   Bus options (PCI etc.)  --->
                   Binary Emulations  --->
               [*] Virtualization  --->
                   General architecture-dependent options  --->
               [*] Enable loadable module support  --->
               -*- Enable the block layer  --->
                   Executable file formats  --->
                   Memory Management options  --->
               [*] Networking support  --->
                   Device Drivers  --->
                   File systems  --->
                   Security options  --->
               -*- Cryptographic API  --->
                   Library routines  --->
                   Kernel hacking  --->

                  <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 3.2: Menuconfig in linux-xtensa kernel port

However, it's important to note that the kernel build process doesn't automatically include the construction of a rootfs. Even if a custom toolchain is utilized — facilitated by the `CROSS_COMPILATION` flag — the rootfs must be built as a separate entity [36, pp. 28, 59–62]. This challenge is addressed by using a build tool such as Buildroot, the details of which will be covered in 3.5.

Another important fork of the Linux kernel source tree, which eventually became a standalone project and has played an important role in the embedded Linux world, was the μClinux project. Originally launched in 1998, the project aimed to enable embedded devices without the support of a memory management unit (MMU) to run Linux [37]. Its integration into the mainline Linux kernel with version 2.6 marks a significant milestone, further solidifying its role in the broader Linux ecosystem [38]. Development on the separate fork of μClinux concluded in 2016 [39]. Among its key contributors, Erik Andersen stands out for creating uClibc, a compact embedded C library, originally intended to be used with μClinux. While the official support for uClibc ended in 2012, its added benefits have been re-introduced with the uClibc-ng spin-off, the C library that is also utilized in the context of this thesis [10].

For the remainder of this thesis, it is vital to understand the Linux kernel's fundamental mode of operation. In the essence, the Linux kernel handles two main tasks: Managing the underlying hardware components by the means of the Hardware Abstraction Layer (HAL); providing an execution environment for the programs on the system to run. When a program needs to use resources on the hardware side, *e.g.*, allocate memory, it issues a request to the kernel upon which the request is being evaluated and if granted, hardware interaction by the kernel takes place on behalf of the program. This mechanism is enforced with a concept called *execution modes*, provided by the CPU. These execution modes consist of two different modes: *user mode* and *kernel mode*. The user mode is a non-privileged mode, whereas the kernel space is a privileged one, *i.e.*, resource allocation is restricted in user mode. Behind the curtains, when a program sends a request for hardware resources to the kernel, the system switches from user mode to kernel mode. Moreover, these modes are associated with separate memory spaces called *user space* and *kernel space* respectively [33, pp. 8–10].

3.5 Buildroot

Buildroot is an open-source tool specifically engineered to simplify and automate the creation of a complete Linux OS for embedded systems through the use of cross-compilation. Operating on Linux-based host systems, Buildroot can be configured and initiated in the same fashion as the Linux kernel by leveraging the GNU make language. Buildroot comes equipped with the ability to generate various components besides the kernel image, including a cross-compilation toolchain, a rootfs and even a bootloader for the target system. While the tool itself builds most of the host packages it needs for compilation, *i.e.*, packages used by the host system on which the build process takes place, it does anticipate certain standard Linux utilities, such as make and GCC, to be pre-installed on the host system [31, pp. 165–178], [40].

The versatility of Buildroot lies in its flexibility to produce any combination of these essential components; for instance, an externally specified toolchain can be employed while still utilizing Buildroot to construct the kernel image and rootfs. Specifying an externally built toolchain (by the means of crosstool-NG) will also be the approach taken in this thesis. Furthermore does Buildroot provide a high level of configuration options in several regards, *e.g.*, a selection of the most common packages to integrate in a kernel build is readily available. The configuration of auxiliary tools, such as BusyBox — which will be elaborated upon in 3.6 — is also facilitated. Buildroot supports a wide range of architectures and their processor variants; this includes support for the Xtensa architecture, making it particularly relevant for this thesis. This modular approach makes Buildroot a favored choice for developing embedded Linux systems.

3.6 BusyBox

In the context of embedded systems and lightweight Linux implementations suited for such systems, BusyBox combines multiple common Unix utilities into a single executable. BusyBox is designed to reduce the resource usage in constrained environments. The main purpose is to offer a functional interface similar to the traditional GNU Core Utilities package, but with much less memory and storage requirements. BusyBox achieves this by focusing on a minimalist design, removing functionalities down to only the most essential parts. This results in an executable that can be as small as a few hundred Kilobytes (KB), depending on the options chosen during compilation [41].

Under the hood, BusyBox operates through symbolic links and hard links that point to the BusyBox binary. When invoked under different names, the BusyBox binary reads its own name and executes the corresponding applet. This allows for a modular approach, where system administrators can choose to include or exclude specific utilities according to the unique requirements of a given project. Thus, BusyBox offers a flexible toolkit for embedded system developers, enabling them to equip their Linux-based systems with a versatile set of utilities without burdening the constrained resources [42].

3.7 Espressif IoT Development Framework

The Espressif IoT Development Framework (ESP-IDF) is the official development framework for the ESP32 and ESP32-S series of SoCs from Espressif Systems. Designed to provide a robust foundation for building IoT applications, ESP-IDF offers a range of software libraries and tools necessary for configuring, compiling, and flashing firmware onto ESP32 devices. The framework is modular and allows for high-level abstractions for tasks such as networking, sensor interfacing, and peripheral management, thus simplifying the development process for complex IoT applications [43].

In this thesis, the ESP-IDF is relevant for its role in creating the environment required to load the compiled Linux kernel image, rootfs, and a custom bootloader onto a development board. It offers this functionality through its application programming interface (API) and integration of the FreeRTOS kernel, thereby providing RTOS capabilities [44].

Chapter 4

Design

The process of building a complex system demands the means of a roadmap upon which the actual implementation is based. This chapter is focused on the design choices that will be vital for the actual realisation of running a Linux kernel on an ESP32 board in the subsequent implementation chapter. These design choices encompass technicalities of the selected development board and further software-related topics crucial to the underlying concept. By analysing these design choices, the reader will gain insight into the logic and structure that form the basis of the implementation. Thus, this chapter serves as a pivotal link in the thesis, connecting the theoretical exploration of related work with the practical steps of the implementation.

4.1 From ESP32 to ESP32-S3

Initially, the implementation was planned to be carried out utilizing the ESP-EYE development board of the ESP32 series. The chosen board is furnished with 4 MB of PSRAM, 8 MB of flash memory, and the ESP32-D0WD SoC. The latter is built around two 32-bit Xtensa LX6 microprocessors, each operating at a frequency of 240 MHz [45]. However, as the implementation phase advanced, an unexptected inconsistency was encountered with the aforementioned board. Specifically, the API of the ESP-IDF framework utilizes the `esp_partition_mmap` function to map a given partition into the data memory [46]. When a partition larger than 3 MB is mapped, the function call is executed successfully. Nevertheless, in the context of the ESP32 series, any partition exceeding this size appears to be mapped erroneously. This issue has been reported and is still open on Espressif's official ESP-IDF GitHub repository at the time of this writing. This bug is believed to be a hardware-related one, thus presenting an obstacle that required a deviation from the ESP32 series [47]. Espressif's newer ESP32-S3 series of the same ESP32 device family, but equipped with a different SoC, emerged as a promising implementation target.

The ESP32-S3-DevKitC-1 development board of aforementioned ESP32-S3 series employs the ESP32-S3-WROOM-1 module with the ESP32-S3 SoC. This module is available with different memory configuration, however, the utilized one employs 8 MB of flash memory and 8 MB of PSRAM [48]. The ESP32-S3 SoC integrates two 32-bit Xtensa LX7 microprocessors, each operating at up to 240 MHz [30, p. 389].

4.2 eXecute-In-Place

Despite the presence of 8 MB each of PSRAM and flash memory, the challenge remains to efficiently allocate space for both the Linux kernel and the rootfs, or the specific parts of them needed at a given time. For instance, as will be shown in the implementation in the next chapter, a stripped-down Linux kernel image alone requires a minimum of 3 MB, and the rootfs demands at least 3.5 MB. This inherently impacts the remaining available memory for dynamic memory allocation. As will be outlined in section 4.3, methods like demand paging, which rely on a capable MMU [49], are not applicable due to the architectural limitations of Espressif's boards. This effectively eliminates the possibility to simply load the kernel along with necessary executables into RAM and run them from there.

For systems with tightly constrained memory resources a technique called eXecute-In-Place (XIP) has proved successful [50]. As the name implies, with XIP the non-volatile storage (*e.g.*, flash memory) is utilized for program execution instead of loading the relevant parts of a program into memory first; in other words, the flash memory directly serves as instruction memory. The XIP kernel support for the Xtensa architecture has been mainlined into the Linux kernel source tree already in 2019 [51]. A kernel compiled with this XIP support differs mostly in the way the static kernel data gets allocated in a distinct segment, which is being loaded from flash memory into a designated memory area [26], [52].

4.3 Memory Management

Both high-performance personal computers and resource-constrained MCUs share a fundamental requirement: the imperative of memory management. In contemporary Linux systems running on robust hardware, the MMU serves as a standard architectural component. This unit is integral to the hardware and interfaces directly with the OS, which is responsible for orchestrating its operations to optimize memory utilization and enable memory protection mechanisms at the same time [49, pp. 164–174]. It is not uncommon for systems to incorporate multiple MMUs functioning at various hierarchical levels. For instance, the Xtensa ISA specifies an MMU at the core level, while another MMU operates at the SoC level in the ESP32-S3 [30, p. 393], [6, p. 217]. One of the most crucial roles of the MMU is to facilitate the concept of virtual memory — a conceptual abstraction that enables the mapping of physical memory locations to virtual or logical memory addresses. A fully-equipped MMU allows the OS to allocate discrete virtual address spaces for individual processes; for instance, in a 32-bit machine, this allocation spans a range of 4 GB, extending from virtual address 0x00000000 to 0xFFFFFFFF. Another ability featured by such a MMU is the concept of paging or swapping. Paging refers to the illusion of having all of a program's essentials readily stored in memory even though this might not be the case, however, by leveraging the concept of virtual addresses, the MMU can create the impression as if it were the case. It achieves this by loading a page stored physically on secondary storage (*e.g.*, on an HDD) on demand into main memory and swapping out the least recently used page from main memory to the secondary storage [53, pp. 496–504].

Unfortunately, this degree of virtualization remains unfeasible in the context of the ESP32-S3 and its underlying Xtensa LX7 microprocessors. Although the Xtensa ISA specifies the architectural capability for cores to employ this form of a MMU, this feature is absent in Espressif's SoC. Instead, the microprocessor cores in the ESP32-S3 deploy a region protection option supplemented by address translation [6, pp. 196–200]. The lack of an MMU with such capabilities has the implication of a single shared address space among all processes, which further complicates matters. In the remainder of this thesis, it will be implied for the ESP32 family to be MMUless (*i.e.*, not having an MMU) for the sake of simplicity.

4.4 Executables and Libraries

4.4.1 Statically & Dynamically linked Libraries

Libraries serve as reusable components that contain pre-compiled pieces of code which can be utilized by multiple programs. Libraries generally exist in two distinct forms: statically and dynamically linked libraries, each with its own set of advantages and limitations. Statically linked libraries are collections of code that are built into a program. Because the code is part of the program, you get a bigger executable that doesn't need separate library files at runtime. However, this means each program has its own copy of the library, which uses more memory. Also, if the library gets updated, you have to recompile your program. Dynamically linked libraries, on the other hand, are separate files loaded when you run the program. Multiple programs can use the same dynamically linked library, saving memory. However, since code from dynamically linked libraries has to be loaded dynamically when needed by the calling process, it imposes additional time required for linking and loading the corresponding location in memory. Updates to the library apply to all programs that use it, without recompiling them. But, the program won't work if the library file is missing or incompatible. Regarding the embedded case at hands, the use of shared libraries is a necessity given by the memory-constraints [54].

4.4.2 The Executable and Linkable Format

The so-called Executable and Linkable Format (ELF) was developed by the Unix System Laboratories and has been the executable file format used in Unix and Unix-like systems such as Linux for many years [33, pp. 824–826].

An ELF file is fundamentally organized into a hierarchical structure that starts with an ELF header at the beginning of the file. The ELF header outlines the properties of the file and references either a program header table or a section header table, or both. These tables, in turn, specify the details of the binary - including code, data segments, dynamic sections, symbol tables, and relocations among other elements. The ELF header not only serves as the roadmap for the ELF file, but also includes key attributes such as the type of machine for which the file is intended, the version of the file, and the program point, to name a few. An ELF file can be position-dependent, *i.e.*, it describes segments that must be loaded into a process's address space at fixed, predetermined addresses, specified in the ELF program header [55], [54, pp. 82–94]. In Linux systems with an MMU, this works seamlessly because each process is being provided with an isolated address space. When the kernel loads a new process, it can allocate these addresses without conflict. But even in such systems a problem arises when dealing with dynamically linked objects (*e.g.*, dynamically linked libraries). Regarding the embedded case, the aforementioned fact that the kernel and all other processes share one common address space and the need for dynamically linked libraries leads directly to the inability of utilizing position-dependent executables [26].

An approach to tackle the aforementioned constraints has found applications in other approaches to MMUless Linux, such as the μClinux project introduced in chapter 3. Regarding the issue of executables in an MMUless environment, μClinux utilizes a file format called binary flat (bFLT) format. Essentially, during the linking process of the toolchain a tool called elf2flt generates the linked binaries in the bFLT format along the ELF ones [56]. The bFLT format enables executables to be loaded where processes share a common address space, however, it poses the problem of statically linked binaries and is there no viable option [26].

To finally solve the limitations at hands, another approach – and also the one taken in this thesis – is the so-called function descriptor position-independent code (FDPIC) ELF. The position-independent code (PIC) part of FDPIC refers to a fairly old concept, with its first applications in 1996. In its core, PIC uses the fact that the .test section of the ELF (*i.e.*, usually the read-only code to be executed) and the .data section of the ELF (*i.e.*, the readable and writable data) have an offset that is known and constant among all processes. When the .text and .data sections have a constant offset, a function pointer can directly point to the function to be executed, *e.g.*, when a program needs to call a subroutine from a library, the function pointer gets resolved to the .text section of that subroutine from where it can be executed. The subroutine's .data section can be easily located since the offset between .text and .data section is constant [54, pp. 203, 208–212]. This, however, is only possible if the running process is being provided with a separate address space for the ELF. When sharing a common address space, the offset between .text and .data section cannot be constant, that's where the function descriptor comes into play. Essentially, instead of a function pointer directly pointing to a subroutine, it points to a function descriptor. The function descriptor in return contains both the address of the subroutine to be executed and the starting address of the subroutine's .data section [57].

Chapter 5

Implementation

As outlined in 4.1, this chapter presents the steps taken to get a Linux kernel image running on an ESP32-S3-DevKitC-1 that employs a ESP32-S3-WROOM-1-N8R8 module, *i.e.*, equipped with 8 MB of both flash memory and PSRAM. The chapter starts with a general setup of the implementation environment, followed by configuring and building the toolchain, paving the way for the build process with the use of Buildroot. After these steps, the kernel image and rootfs are obtained. The chapter wraps up by describing the last steps, which involve using a custom bootloader and the ESP-IDF framework to flash everything onto the board. The condensed sources to reproduce the implementation can be found online [58], however, the original forks on which this implementation is based on can be found by following the git submodules. It is also important to note that the implementation this one is based on is under active development at the time of this writing and hence, this implementation features not the most recent updates. A streamlined version attempting to stay updated can be found online [59].

5.1 Setup

The repository provided in [58] has to be cloned with the `--recursive` flag enabled to ensure the submodules get cloned as well. Change into the cloned linux-esp32s3 directory and source the setup.sh bash script. It is essential to source the script in order to export required environment variables for the current shell process. The wpa_supplicant that gets installed in the kernel image is responsible for connecting to the WiFi. The setup.sh script can be sourced as `source setup.sh -w ssid psk`, where ssid has to be replaced with the actual WiFi's ssid and psk with the actual WiFi's passkey, which will essentially modify the wpa_supplicant.conf file that gets built into the /etc directory of the rootfs by Buildroot. The purposes of the exported environment variables are as follow:

- **ESP32S3_ROOT_DIR**: Specifies the absolute path to the root directory of the implementation repository. Specifying this environment variable is crucial for the subsequent build steps, since the streamlined implementation is configured with its presence in mind.

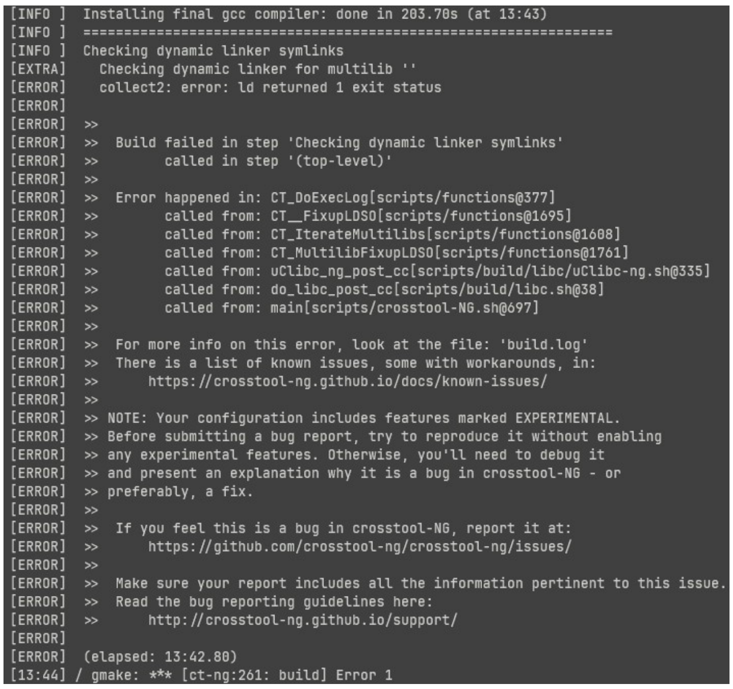
- **CT_PREFIX**: Used by crosstool-NG internally. It points to the output directory where the built toolchain will reside. Since Buildroot's configuration in this implementation depends on this specific location, it is essential to leave this unchanged as well.
- **CONF_DIR**: The xtensa-dynconfig tool needs this environment variable to find the correct location of the dynamic configuration files. Leave this unchanged when building with dynamic configuration.
- **ORIG**: The same applies to this environment variable, it is used by the xtensa-dynconfig tool to include all essential files for the dynamic configuration. Leave this unchanged when building with dynamic configuration.

```
~$ git clone https://github.com/fromGreg/linux-esp32s3.git --recursive
~$ cd linux-esp32s3
~/linux-esp32s3$ source setup.sh -w ssid psk
wpa_supplicant.conf configured with...
        SSID: ssid
        PSK: psk
Exported environment variables...
        ESP32S3_ROOT_DIR: /home/user/linux-esp32s3
        CT_PREFIX: /home/user/linux-esp32s3/crosstool-NG/output
        CONF_DIR: /home/user/linux-esp32s3/configs
        ORIG: 1
```

Listing 5.1: Sourcing setup.sh with wpa_supplicant

5.2 Configuring and building the Toolchain

The initial step in the implementation process involves compiling the toolchain, configured to align with the Xtensa architecture and the specific ESP32-S3 SoC. It is important to highlight that the release of GCC-13 and Binutils-2.40 allows for dynamic configuration, *i.e.*, the toolchain can be compiled using default settings and make adjustments later during the building of the Linux kernel image and rootfs. This approach provides the advantage of creating a more versatile toolchain, which can be adapted for various processors as needed [60]. Therefore, both procedures are described in separate subsections. At this point be mentioned that the build process successfully compiles all required binaries in both cases. However, following the compilation of the final C compiler, an exception related to the linking process is thrown. The specific consequences of this exception are currently unknown but do not appear to impede the construction of the kernel image and rootfs, nor the operation of the kernel and user space on the device.



```
[INFO ]  Installing final gcc compiler: done in 203.70s (at 13:43)
[INFO ]  =================================================================
[INFO ]  Checking dynamic linker symlinks
[EXTRA]    Checking dynamic linker for multilib ''
[ERROR]    collect2: error: ld returned 1 exit status
[ERROR]
[ERROR]  >>
[ERROR]  >>  Build failed in step 'Checking dynamic linker symlinks'
[ERROR]  >>        called in step '(top-level)'
[ERROR]  >>
[ERROR]  >>  Error happened in: CT_DoExecLog[scripts/functions@377]
[ERROR]  >>        called from: CT__FixupLDSO[scripts/functions@1695]
[ERROR]  >>        called from: CT_IterateMultilibs[scripts/functions@1608]
[ERROR]  >>        called from: CT_MultilibFixupLDSO[scripts/functions@1761]
[ERROR]  >>        called from: uClibc_ng_post_cc[scripts/build/libc/uClibc-ng.sh@335]
[ERROR]  >>        called from: do_libc_post_cc[scripts/build/libc.sh@38]
[ERROR]  >>        called from: main[scripts/crosstool-NG.sh@697]
[ERROR]  >>
[ERROR]  >>  For more info on this error, look at the file: 'build.log'
[ERROR]  >>  There is a list of known issues, some with workarounds, in:
[ERROR]  >>      https://crosstool-ng.github.io/docs/known-issues/
[ERROR]  >>
[ERROR]  >> NOTE: Your configuration includes features marked EXPERIMENTAL.
[ERROR]  >> Before submitting a bug report, try to reproduce it without enabling
[ERROR]  >> any experimental features. Otherwise, you'll need to debug it
[ERROR]  >> and present an explanation why it is a bug in crosstool-NG - or
[ERROR]  >> preferably, a fix.
[ERROR]  >>
[ERROR]  >>  If you feel this is a bug in crosstool-NG, report it at:
[ERROR]  >>      https://github.com/crosstool-ng/crosstool-ng/issues/
[ERROR]  >>
[ERROR]  >>  Make sure your report includes all the information pertinent to this issue.
[ERROR]  >>  Read the bug reporting guidelines here:
[ERROR]  >>      http://crosstool-ng.github.io/support/
[ERROR]
[ERROR]  (elapsed: 13:42.80)
[13:44] / gmake: *** [ct-ng:261: build] Error 1
```

Figure 5.1: Toolchain build exception related to the linker

5.2.1 Installing crosstool-NG

The toolchain is compiled with the help of crosstool-NG. The following packages must be pre-installed on the host OS.:

- gcc
- g++
- gperf
- bison
- flex
- texinfo
- help2man
- make
- libncurses5
- python3
- autoconf
- automake
- libtool
- libtool-bin
- gawk
- wget
- bzip2
- xz-utils
- unzip
- patch
- libstdc++6
- rsync
- git
- meson
- ninja-build

[61]

Changing into the crosstool-NG directory, the tool needs to be initialized first by issuing the following commands:

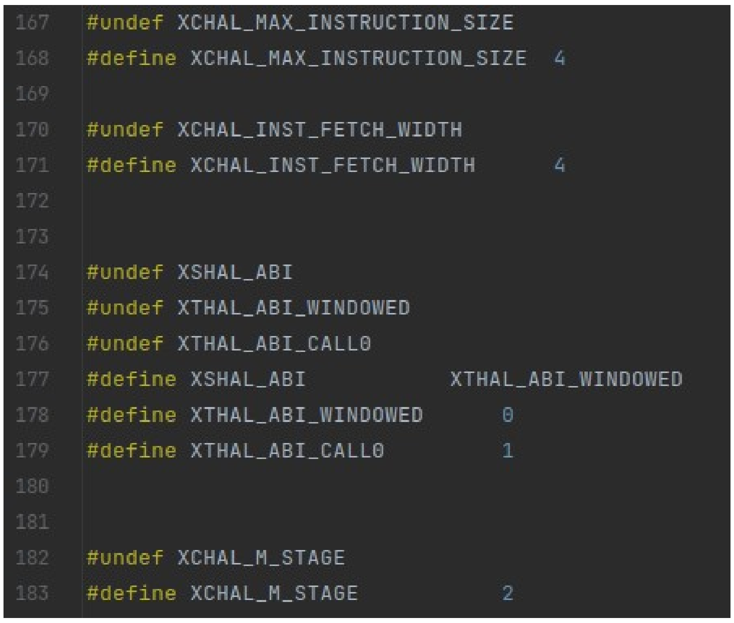
```
~/linux-esp32s3/crosstool-NG$ ./bootstrap
~/linux-esp32s3/crosstool-NG$ ./configure --enable-local
~/linux-esp32s3/crosstool-NG$ make
```

Listing 5.2: crosstool-NG installation

These commands are essentially the crosstool-NG installation procedure. Executing `./bootstrap` enables GNU's Autoconf and Automake utilities to create the configure executable from the configure.ac file along with several *.in files [62], [63]. Subsequently, executing the generated `./configure --enable-local` generates a Makefile in return, which GNU's make utility uses to compile another set of files, however, the generated utility of interest is the ct-ng executable which is used for building the toolchain [35]. Note that by using the `--enable-local` flag, crosstool-NG gets installed only locally, *i.e.*, in the crosstool-NG working directory [64].

5.2.2 Overlay configuration

In the non-dynamic case, the toolchain must be configured with so-called overlay files. These overlay files consist primarily of C header files and a few C source files that use the former. Those files are usually provided by the processor vendor, *i.e.*, by Cadence Design Systems or in this case by Espressif [65]. The overlay files define things relevant for compiling the toolchain correctly for the Xtensa ISA, such as the maximum instruction size as can be seen in Figure 5.2. In the actuality, the Xtensa ISA defines instruction sizes to be 16- or 24-bit, *i.e.*, 2- or 3-byte instructions [6, pp. 32–36]. However, since the LX7 cores of Cadence Design Systems are designed after the Harvard model, they restrict the instruction bus access to only 4-byte aligned manner, which is why the `XCHAL_MAX_INSTRUCTION_SIZE` must be defined as 4 bytes [30, p. 390].

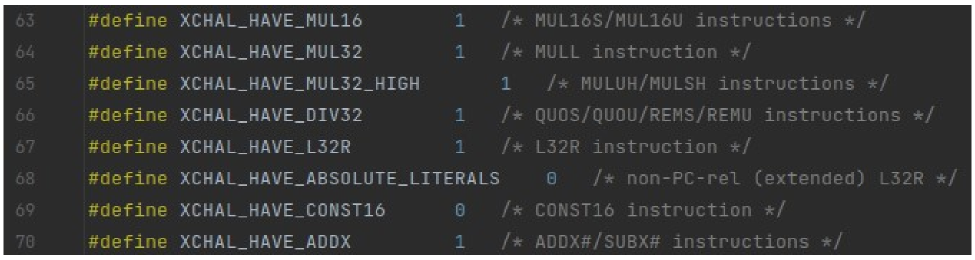


```
#undef XCHAL_MAX_INSTRUCTION_SIZE
#define XCHAL_MAX_INSTRUCTION_SIZE  4

#undef XCHAL_INST_FETCH_WIDTH
#define XCHAL_INST_FETCH_WIDTH      4


#undef XSHAL_ABI
#undef XTHAL_ABI_WINDOWED
#undef XTHAL_ABI_CALL0
#define XSHAL_ABI           XTHAL_ABI_WINDOWED
#define XTHAL_ABI_WINDOWED      0
#define XTHAL_ABI_CALL0         1


#undef XCHAL_M_STAGE
#define XCHAL_M_STAGE           2
```

Figure 5.2: Detail of xtensa-config.h from the overlay files

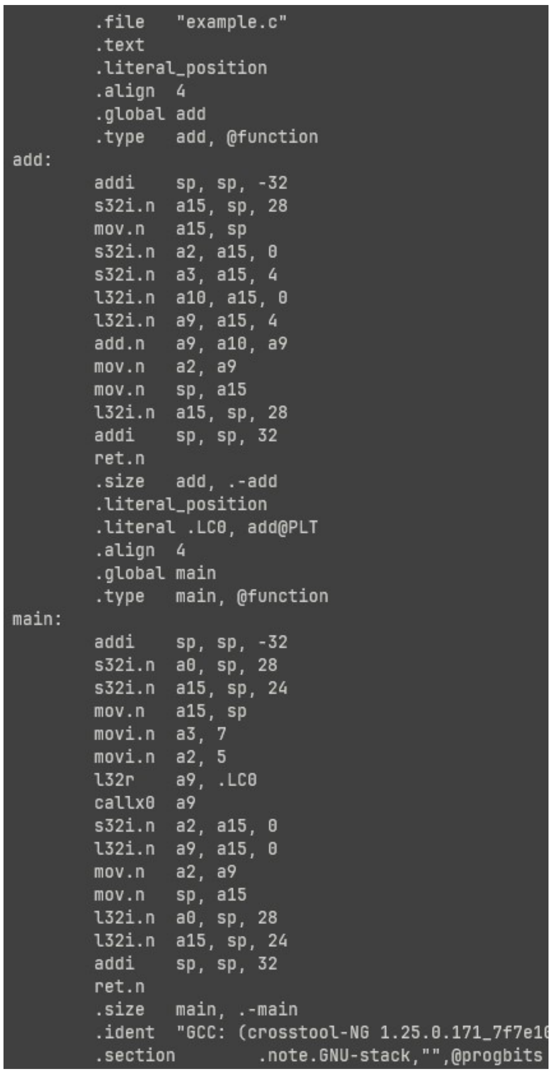
Another point of significance is the ABI utilized. The Xtensa architecture offers two distinct ABIs for processor configuration: the Windowed ABI and the Call0 ABI. The ISA describes 64 general-purpose registers, with 16 being active at any given time. In the Windowed ABI, these registers are rotated to avoid the computational cost of more frequent store and load instructions to memory. Specifically, when a routine calls a subroutine, the window of the visible 16 general-purpose registers shifts, potentially presenting a new set of unused registers for the subroutine to operate on. Upon the subroutine's completion, the window reverses, making the subroutine's results accessible to the initiating routine. Conversely, the Call0 ABI is more straightforward, allowing only the use of a fixed set of 16 general-purpose registers and relying on store and load instructions for subroutine invocations [6, pp. 684–687]. Unfortunately, it proved difficult to make use of the Windowed ABI, hence both kernel and user space in this implementation realize the Call0 ABI [66]. It is important to note that the only modification made to the overlay files occurs in the xtensa-config.h file, seen on line 177 in Figure 5.2, where the ABI definition is set to `#define XSHAL_ABI XTHAL_ABI_CALL0`.

Besides the configuration provided by the xtensa-overlay files, further configuration is given by the specific core definitions, *i.e.*, the LX7 core definitions, found in Espressif's HAL components. These configurations are responsible for correctly configuring the toolchain with more in-depth options, such as definitions of the concrete ISA instructions and their availability in these cores [67]. For instance, Figure 5.3 specifies `XCHAL_HAVE_L32R`, indicating that the core has the `L32R` instruction enabled. This instruction represents a 32-bit load operation relative to the program counter (PC) [6, p. 56]. Enabling this instruction in the toolchain configuration allows the compiler to generate assembly code that utilizes it where applicable. This is evident in the compiled assembly code of a sample example.c file, as shown in Figure 5.4, which employs a subroutine named `add` to sum the immediate values 5 and 7.



```
#define XCHAL_HAVE_MUL16        1   /* MUL16S/MUL16U instructions */
#define XCHAL_HAVE_MUL32        1   /* MULL instruction */
#define XCHAL_HAVE_MUL32_HIGH       1   /* MULUH/MULSH instructions */
#define XCHAL_HAVE_DIV32        1   /* QUOS/QUOU/REMS/REMU instructions */
#define XCHAL_HAVE_L32R         1   /* L32R instruction */
#define XCHAL_HAVE_ABSOLUTE_LITERALS    0   /* non-PC-rel (extended) L32R */
#define XCHAL_HAVE_CONST16      0   /* CONST16 instruction */
#define XCHAL_HAVE_ADDX         1   /* ADDX#/SUBX# instructions */
```

Figure 5.3: Detail of core-isa.h from the HAL



```
        .file   "example.c"
        .text
        .literal_position
        .align  4
        .global add
        .type   add, @function
add:
        addi    sp, sp, -32
        s32i.n  a15, sp, 28
        mov.n   a15, sp
        s32i.n  a2, a15, 0
        s32i.n  a3, a15, 4
        l32i.n  a10, a15, 0
        l32i.n  a9, a15, 4
        add.n   a9, a10, a9
        mov.n   a2, a9
        mov.n   sp, a15
        l32i.n  a15, sp, 28
        addi    sp, sp, 32
        ret.n
        .size   add, .-add
        .literal_position
        .literal .LC0, add@PLT
        .align  4
        .global main
        .type   main, @function
main:
        addi    sp, sp, -32
        s32i.n  a0, sp, 28
        s32i.n  a15, sp, 24
        mov.n   a15, sp
        movi.n  a3, 7
        movi.n  a2, 5
        l32r    a9, .LC0
        callx0  a9
        s32i.n  a2, a15, 0
        l32i.n  a9, a15, 0
        mov.n   a2, a9
        mov.n   sp, a15
        l32i.n  a0, sp, 28
        l32i.n  a15, sp, 24
        addi    sp, sp, 32
        ret.n
        .size   main, .-main
        .ident  "GCC: (crosstool-NG 1.25.0.171_7f7e1
        .section        .note.GNU-stack,"",@progbits
```

Figure 5.4: Assembly of example.c compiled with the built toolchain

In the streamlined implementation repository the overlay files are located in the /linux-esp32s3/configs directory as compressed xtensa_esp32s3-overlay-config.tar.gz file. The xtensa-esp32s3-overlayed default configuration for crosstool-NG resides in the /samples subdirectory and comprises multiple variables. These variables serve various functions, such as specifying the repositories from which GCC, Binutils and uClibc-ng should be obtained, setting C compiler flags for the compilation process and more. With the ct-ng utility at hands, the following commands apply the corresponding crosstool-NG configuration and subsequently build the toolchain accordingly:

```
~/linux-esp32s3/crosstool-NG$ ./ct-ng xtensa-esp32s3-overlayed
~/linux-esp32s3/crosstool-NG$ ./ct-ng build
```

Listing 5.3: Applying overlay crosstool-NG configuration

5.2.3 Dynamic configuration

Even though the dynamically configured toolchain can be later modified during the Buildroot process to accommodate a broader range of processors, some initial configuration is necessary. This configuration informs the GCC compiler *e.g.*, about the ABI it gets configured for [68]. The configuration files that form the basis for the dynamic configuration originate also from the aforementioned xtensa-overlay files and the ESP32-S3 HAL, however, this approach differs in the way those files are being used. In the dynamic case, instead of compiling GCC with the header and source files from the overlays, they are precompiled to a shared object in advance and can then be used as such by GCC. This can be achieved with the provided xtensa-dynconfig tool [69]. Changing into the xtensa-dynconfig directory, the following commands can be issued to generate the needed xtensa_esp32s3-dynconf-config.so shared object:

```
~/linux-esp32s3/xtensa-dynconfig$ make xtensa_esp32s3-dynconf-config.so
~/linux-esp32s3/xtensa-dynconfig$ export XTENSA_GNU_CONFIG=$(pwd)/
    xtensa_esp32s3-dynconf-config.so
```

Listing 5.4: Generating dynamic configuration shared object

The environment variable `XTENSA_GNU_CONFIG` exported on the last line is crucial to both GCC and Buildroot, since if configured for the dynamic case, they can find the needed shared object at the location specified. Hence, to proceed, the same steps are taken as in the overlay-case but with another crosstool-NG configuration:

```
~/linux-esp32s3/crosstool-NG$ ./ct-ng xtensa-esp32s3-dynconf
~/linux-esp32s3/crosstool-NG$ ./ct-ng build
```

Listing 5.5: Applying dynamic crosstool-NG configuration

5.3 Building kernel image and rootfs

The compiled toolchain resides now in the /linux-esp32s3/crosstool-NG/output directory and is labelled xtensa-esp32s3-linux-uclibcfdpic. To proceed with the construction of the kernel image and the rootfs, the working directory must be switched to /linux-esp32s3/buildroot. The following packages must be pre-installed on the host OS additionally to the ones already installed for crosstool-NG:

- which
- sed
- binutils
- build-essential
- diffutils
- bash
- bzip2
- gzip
- perl

- tar
- cpio
- unzip
- file
- bc
- findutils

[40]

Similar to the toolchain configuration, the steps for building the kernel and rootfs will vary based on whether the dynamic configuration is employed or the configuration with overlay files.

5.3.1 Overlay configuration

Following the steps taken in 5.2.2, the following command applies the corresponding configuration for Buildroot:

```
~/linux-esp32s3/buildroot$ make O=${ESP32S3_ROOT_DIR}/buildroot-build
    esp32s3wifi_overlayed_defconfig
```

Listing 5.6: Applying overlay Buildroot configuration

The inlined environment variable `O` serves the purpose of redirecting the build output into /linux-esp32s3/buildroot-built which is not mandatory but advisable, especially since the subsequent implementation steps assume the buildroot-build directory to contain the built kernel image and rootfs.

The esp32s3wifi_overlayed_defconfig configuration resides in the /linux-esp32s3/buildroot /configs directory and specifies various settings, such as where the overlay files can be found, where the toolchain resides, flags for the toolchain's linker, where the rootfs overlay resides (this is the one that also contains the wpa_supplicant.conf file enabling the built kernel to connect to WiFi) and where to obtain the kernel source from, *i.e.*, from which repository. Furthermore does the configuration specify the architecture and a variant; these two options will be picked up by the kernel source in the build process to locate essential header files and its own configuration file. Moreover, the configuration also indicates that the rootfs will be a cramfs and that a tmpfs will be employed, as will be discussed in 6.1.1. If the aforementioned `O` environment variable has been used, the directory needs to be changed to /linux-esp32s3/buildroot-build in order to initiate the build process as follows:

```
~/linux-esp32s3/buildroot-build$ make
```

Listing 5.7: Initiating Buildroot build

5.3.2 Dynamic configuration

For the approach taken in 5.2.3, the corresponding Buildroot configuration is esp32wifi_dynconf_defcon
The application of this configuration and initiation of the build process is the same as in the overlay case:

```
~/linux-esp32s3/buildroot$ make O=${ESP32S3_ROOT_DIR}/buildroot-build
    esp32s3wifi_dynconf_defconfig
~/linux-esp32s3/buildroot$ cd ../buildroot-build
~/linux-esp32s3/buildroot-build$ make
```

Listing 5.8: Applying dynamic Buildroot configuration and build

This configuration differs from the overlay one primarily in the sense that the `BR2_XTENSA_DYNCONFIG` option is enabled and no path to overlay files is specified. However, it is important so start the build process in an environment where the `XTENSA_GNU_CONFIG` has been exported according to the instructions in 5.2.3.

5.4 Flashing kernel image and rootfs

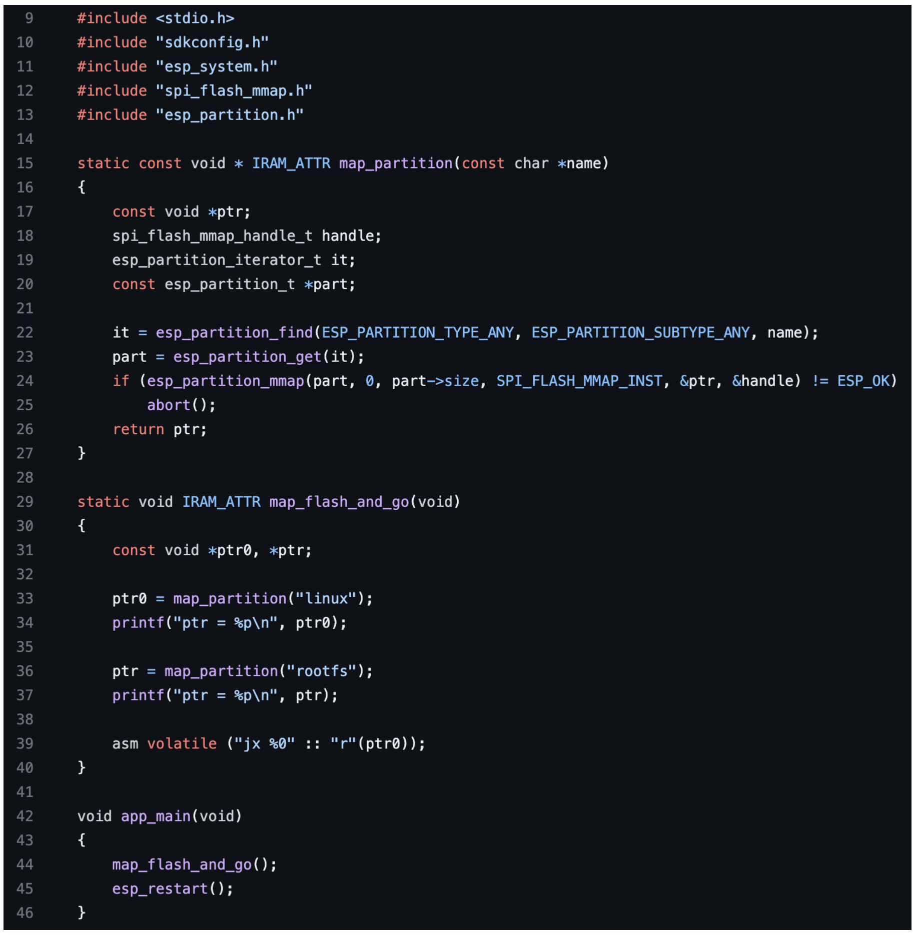
The kernel image built in 5.3 employs an additional driver necessary for the Universal Asynchronous Receiver/Transmitter (UART) on the development board to communicate with the connected host machine [70]. But before this driver can be of any use, kernel image and the rootfs need to be flashed onto the board. This is achieved with a custom bootloader based on the *hello_world* example in the ESP-IDF framework [71].

5.4.1 Early bootloader version

While the bootloader used in this thesis has undergone modifications to incorporate WiFi capabilities, its core functionality remains unchanged compared to its eralier versions. Consequently, for the sake of simplified illustration, one such early version is presented here. Figure 5.5 shows this bootloader's linux_boot_main.c source file.

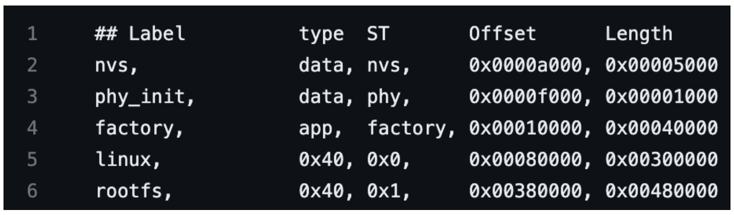
The bootloader comes with several partition table definitions, one of them being a partition table for the ESP32-S3 equipped with 8 MB of flash memory. Its definition is shown in Figure 5.6.

In essence, the partition table, along with the compiled code depicted in Figure 5.5, is flashed onto the board. The partition designated to store the kernel image is labeled as *linux*, while the partition allocated for the rootfs is labeled *rootfs*. The kernel image and rootfs, generated as discussed in Section 5.3, are subsequently flashed onto their corresponding partitions, a process which will be elaborated upon in Section 5.4.2. Upon booting, the bootloader first invokes the `map_flash_and_go` method. This method is



```c
#include <stdio.h>
#include "sdkconfig.h"
#include "esp_system.h"
#include "spi_flash_mmap.h"
#include "esp_partition.h"

static const void * IRAM_ATTR map_partition(const char *name)
{
    const void *ptr;
    spi_flash_mmap_handle_t handle;
    esp_partition_iterator_t it;
    const esp_partition_t *part;

    it = esp_partition_find(ESP_PARTITION_TYPE_ANY, ESP_PARTITION_SUBTYPE_ANY, name);
    part = esp_partition_get(it);
    if (esp_partition_mmap(part, 0, part->size, SPI_FLASH_MMAP_INST, &ptr, &handle) != ESP_OK)
        abort();
    return ptr;
}

static void IRAM_ATTR map_flash_and_go(void)
{
    const void *ptr0, *ptr;

    ptr0 = map_partition("linux");
    printf("ptr = %p\n", ptr0);

    ptr = map_partition("rootfs");
    printf("ptr = %p\n", ptr);

    asm volatile ("jx %0" :: "r"(ptr0));
}

void app_main(void)
{
    map_flash_and_go();
    esp_restart();
}
```

Figure 5.5: Early bootloader version for ESP32-S3



```
## Label         type  ST        Offset      Length
nvs,             data, nvs,      0x0000a000, 0x00005000
phy_init,        data, phy,      0x0000f000, 0x00001000
factory,         app,  factory,  0x00010000, 0x00040000
linux,           0x40, 0x0,      0x00080000, 0x00300000
rootfs,          0x40, 0x1,      0x00380000, 0x00480000
```

Figure 5.6: Partition table for ESP32-S3 with 8 MB flash memory

responsible for mapping both the *linux* and *rootfs* partitions to the virtual address space as instruction memory, utilizing ESP-IDF's `esp_partition_mmap` method [72]. Following this, an inline assembly jump instruction reassigns the PC to the starting address of the mapped *linux* partition. In other words, the PC is now set to point to the kernel's initial location. When the `esp_restart` method is subsequently invoked, the board reboots, initializing from the newly configured PC location, thereby booting the Linux kernel [73]. In the current implementation the kernel runs only on one of the two available cores with the intention to run the ESP-IDF firmware on the other [74].

5.4.2 Flashing procedure

Espressif provides with their ESP-Hosted-NG solution the possibility to enable their SoC's WiFi capabilities [75]. A modified version of this solution is used in this implementation to enable the kernel to use these capabilities [76]. Hence, to proceed with the flashing procedure in the streamlined implementation, the directory needs to be changed to /linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver. This directory contains a modified version of the ESP-IDF as submodule in its /esp-idf subdirectory and the modified bootloader configured with network adapter capabilites in its /network_adapter subdirectory. The subsequent commands are issued to initialize the framework:

```
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver$ cmake .
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver$ source esp-idf/export.
    sh
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver$ cd network_adapter
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ idf.py
    set-target esp32s3
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ cp
    sdkconfig.defaults.esp32s3 sdkconfig
```

Listing 5.9: Initializing the ESP-IDF framework

The first command cleans the ESP-IDF submodule, replaces the esp_wifi library components in the submodule with the ones in the esp_driver directory and executes the install.sh script located in the esp-idf directory. The install.sh script installs several python scripts, such as the idf.py used to flash onto and monitor the board and the parttool.py used to write the kernel image and rootfs onto the board. By sourcing the export.sh script with the second command, environment variables vital for the esp-idf are being exported. The third command changed the directory to the one actually containing the bootloader project to be flashed. The fourth command cleans previous build outputs if any and sets up the framework for the ESP32-S3, generating an sdkconfig file among other things, even though this file will be replaced by a custom configuration with the fifth command. After these steps, the bootloader project can be built and once the board is connected via the UART, flashed onto the board by issuing the following commands [77]:

```
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ idf.py
    build
```

```
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ idf.py
    flash
```

Listing 5.10: Building and flashing the project

The final step is to write the kernel image and the rootfs onto their respective partitions, using the previously mentioned parttool.py script:

```
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ parttool
    .py write_partition --partition-name linux --input ${
    ESP32S3_ROOT_DIR}/buildroot-build/images/xipImage
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ parttool
    .py write_partition --partition-name rootfs --input ${
    ESP32S3_ROOT_DIR}/buildroot-build/images/rootfs.cramfs
```

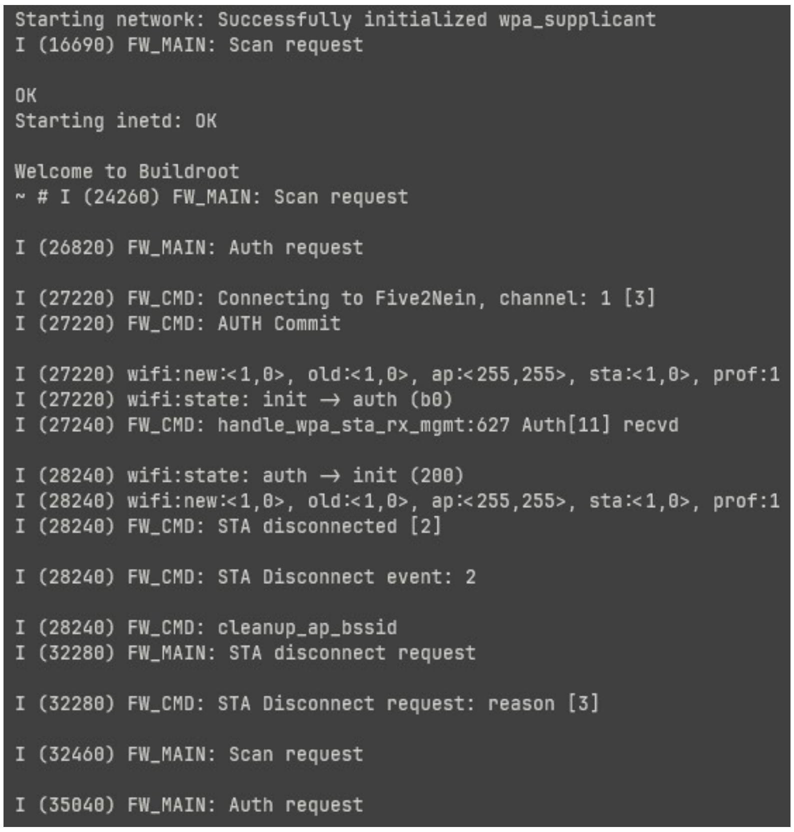
Listing 5.11: Writing kernel image and rootfs onto their partitions

After this step, the bootloader can be triggered and ultimately the kernel be booted by invoking:

```
~/linux-esp32s3/esp-hosted/esp_hosted_ng/esp/esp_driver/network_adapter$ idf.py
    monitor
```

Listing 5.12: Booting Linux via idf.py monitor

Eventually the user is prompted to enter the passphrase, which is defaulted to *root* and if the wpa_supplicant.conf has been configured accordingly as described in 5.1, it will try to connect to the WiFi as can be seen in Figure 5.7.



```
Starting network: Successfully initialized wpa_supplicant
I (16690) FW_MAIN: Scan request

OK
Starting inetd: OK

Welcome to Buildroot
~ # I (24260) FW_MAIN: Scan request

I (26820) FW_MAIN: Auth request

I (27220) FW_CMD: Connecting to Five2Nein, channel: 1 [3]
I (27220) FW_CMD: AUTH Commit

I (27220) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:1
I (27220) wifi:state: init -> auth (b0)
I (27240) FW_CMD: handle_wpa_sta_rx_mgmt:627 Auth[11] recvd

I (28240) wifi:state: auth -> init (200)
I (28240) wifi:new:<1,0>, old:<1,0>, ap:<255,255>, sta:<1,0>, prof:1
I (28240) FW_CMD: STA disconnected [2]

I (28240) FW_CMD: STA Disconnect event: 2

I (28240) FW_CMD: cleanup_ap_bssid
I (32280) FW_MAIN: STA disconnect request

I (32280) FW_CMD: STA Disconnect request: reason [3]

I (32460) FW_MAIN: Scan request

I (35040) FW_MAIN: Auth request
```

Figure 5.7: Linux login on ESP32-S3

Chapter 6

Evaluation

Following the presented implementation, the task shifts to the assessment of the established system's capabilities. Therein, this chapter provides a concise evaluation of the former. Starting with general observations obtained from hands-on interaction with the system, and proceeding through time measurements, the evaluation ultimately culminates in an analysis of the system's WiFi capabilities.

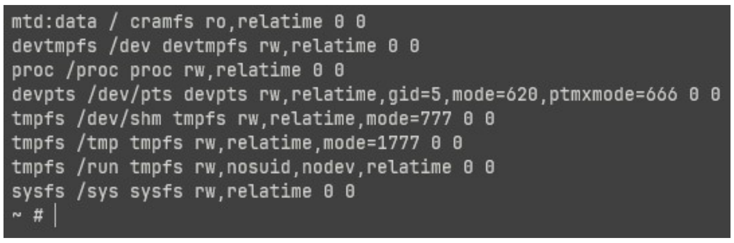
As a useful sidenote, the graphical representations of the Command Line Interface (CLI) in the following sections, which relate to the monitored kernel, may occasionally be misleading. In particular, some characters from an earlier stdout may show up on the current stdin line, even though they are leftovers from the preceding operation and not part of the current input, as can be seen *e.g.*, in Figure 6.7 where `droot login: root` is only a residual.

6.1 General observations

The Linux filesystem employs the concept of the so-called virtual filesystem (VFS), whose role is to provide a software layer that functions as an interface to several different filesystems. One class of filesystems supported by the VFS are the *special filesystems* or *pseudo filesystems*, in which category also the /proc filesystems falls [33, pp. 456–482], [78]. As such, it provides an interface to access kernel data structures, *i.e.*, it can be leveraged to obtain useful system information for evaluation, as will be outlined in this section.

6.1.1 Mounted filesystems

The Linux VFS is hierarchical and consists of various types of filesystems, each serving specific functions and purposes. For instance, the invocation of `cat /proc/mounts` presents a list of the mount_namespaces available to the invoking process, *i.e.*, in this context, to the shell executing the command [78]. Invoked on the provided implementation,



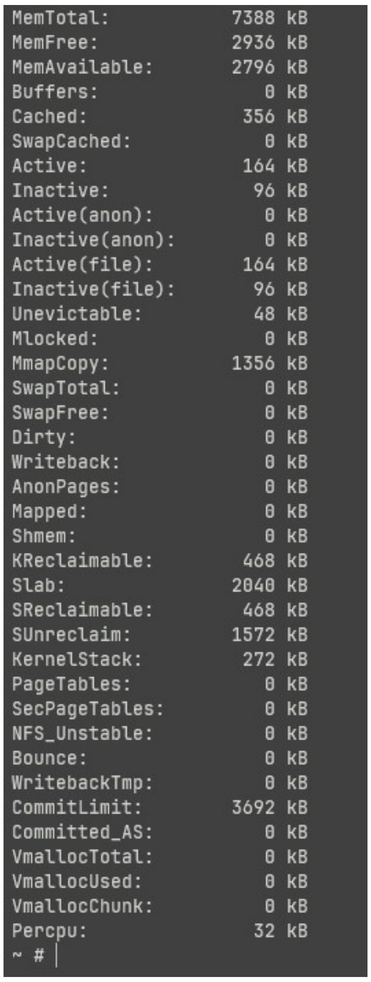
```
mtd:data / cramfs ro,relatime 0 0
devtmpfs /dev devtmpfs rw,relatime 0 0
proc /proc proc rw,relatime 0 0
devpts /dev/pts devpts rw,relatime,gid=5,mode=620,ptmxmode=666 0 0
tmpfs /dev/shm tmpfs rw,relatime,mode=777 0 0
tmpfs /tmp tmpfs rw,relatime,mode=1777 0 0
tmpfs /run tmpfs rw,nosuid,nodev,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
~ # |
```

Figure 6.1: Output of `cat /proc/mounts`

the output presents the list seen in Figure 6.1. The mtd:data filesystem is the root filesystem mounted at / and respresents the read-only compressed ROM filesystem (cramfs) as configured in 5.3. However, the filesystem of particular interest for the subsequent evaluations is the tmpfs mounted at /tmp. This filesystem provides read/write capabilities from the user space, *i.e.*, it enables the user to create, manipulate and delete files located there. Unfortunately, the data represented through the /tmp filesystem resides on volatile memory and as such poses the most significant limitation for real-world use cases: Data whose attributes and volume may be indeterminate at initialization, might require preservation beyond the system's uptime. Nevertheless, the current state of affairs suggests the issue partially being addressed [79].

6.1.2 Memory information

The `cat /proc/meminfo` provides an elaborate snapshot of the system's memory status quo, as shown in Figure 6.2. It is particularly of interest in regards to this implementation, in the MMUless context and the employment of XIP. The MemTotal entry, which specifies the total available memory as 7388 KB, makes efficient memory utilization a high priority, especially considering the incapability of establishing advanced memory handling techniques such as paging.
The MemAvailable metric, showing 2796 KB of estimated readily accessible memory [80], becomes even more significant in light of the XIP implementation as well, since executable code is run directly from flash memory, thereby saving PSRAM that would otherwise be consumed for loading the executable into memory. Furthermore, the presence of a notable Slab memory of 2044 kB highlights how the system uses slab allocation to manage its kernel objects efficiently. Slab allocation becomes increasingly important in a system lacking an MMU because it reduces fragmentation and optimizes memory usage for kernel objects, thus maximizing the efficiency of the available memory space [33, pp. 324–350].



```
MemTotal:           7388 kB
MemFree:            2936 kB
MemAvailable:       2796 kB
Buffers:               0 kB
Cached:              356 kB
SwapCached:            0 kB
Active:              164 kB
Inactive:             96 kB
Active(anon):          0 kB
Inactive(anon):        0 kB
Active(file):        164 kB
Inactive(file):       96 kB
Unevictable:          48 kB
Mlocked:               0 kB
MmapCopy:           1356 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:                 0 kB
Writeback:             0 kB
AnonPages:             0 kB
Mapped:                0 kB
Shmem:                 0 kB
KReclaimable:        468 kB
Slab:               2040 kB
SReclaimable:        468 kB
SUnreclaim:         1572 kB
KernelStack:         272 kB
PageTables:            0 kB
SecPageTables:         0 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:        3692 kB
Committed_AS:          0 kB
VmallocTotal:          0 kB
VmallocUsed:           0 kB
VmallocChunk:          0 kB
Percpu:               32 kB
~ # |
```

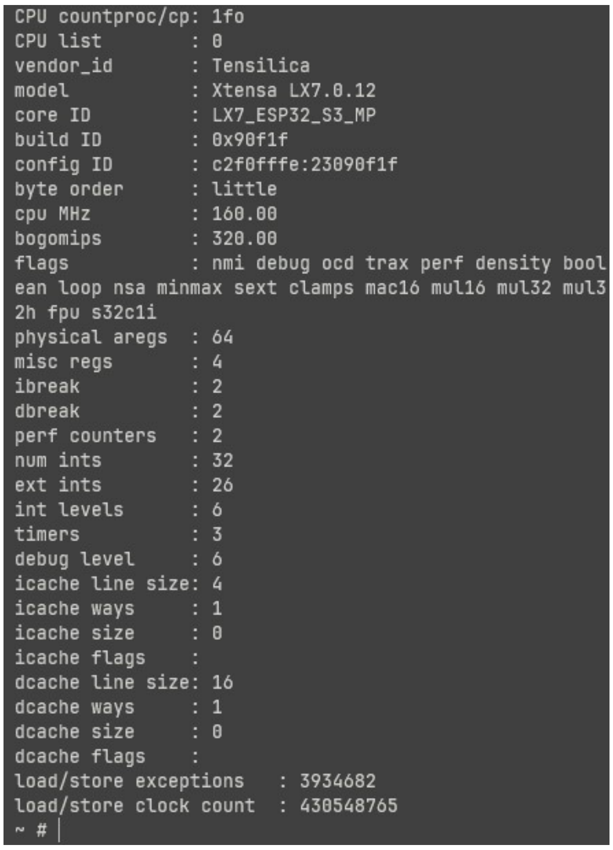
Figure 6.2: Output of `cat /proc/meminfo`

6.1.3 CPU information

Listing the contents of another pseudo file by `cat /proc/cpuinfo`, presents the output shown in Figure 6.3. The first line, as mentioned in the sidenote in 6, is misleading here, since it supposedly aims to display the count of processors the system runs on as `CPU count : 1`, *i.e.*, as explained in 5.4.1, the system makes use of only one of the available two processor cores. Further, besides a bunch of meta information, such as the vendor_id being defined as Tensilica and the core ID as Xtensa LX7.0.12, more insightful information can be gained. Although the ESP32-S3 is designed to provide a CPU frequency of up to 240 MHz in high performance mode [30], here it is only operating at 160 MHz.

One notable entry is the set of CPU flags, which indicate capabilities for which the core has been configured. These align with the core configuration overlay as elaborated in 5.3, *e.g.*, as visibible in Figure 5.3 has the core been configured to support the MUL16, MUL32, MUL32_HIGH instructions that correspond to the flags mul16, mul32, mul32h respectively [6, pp. 324–617].

Another interesting insight presented is the distinction between ICache and DCache related cache entries, which reflect the nature of a Harvard architecture as outlined in 3.1, *i.e.*, the system having separate instruction and data busses connected to different caches.



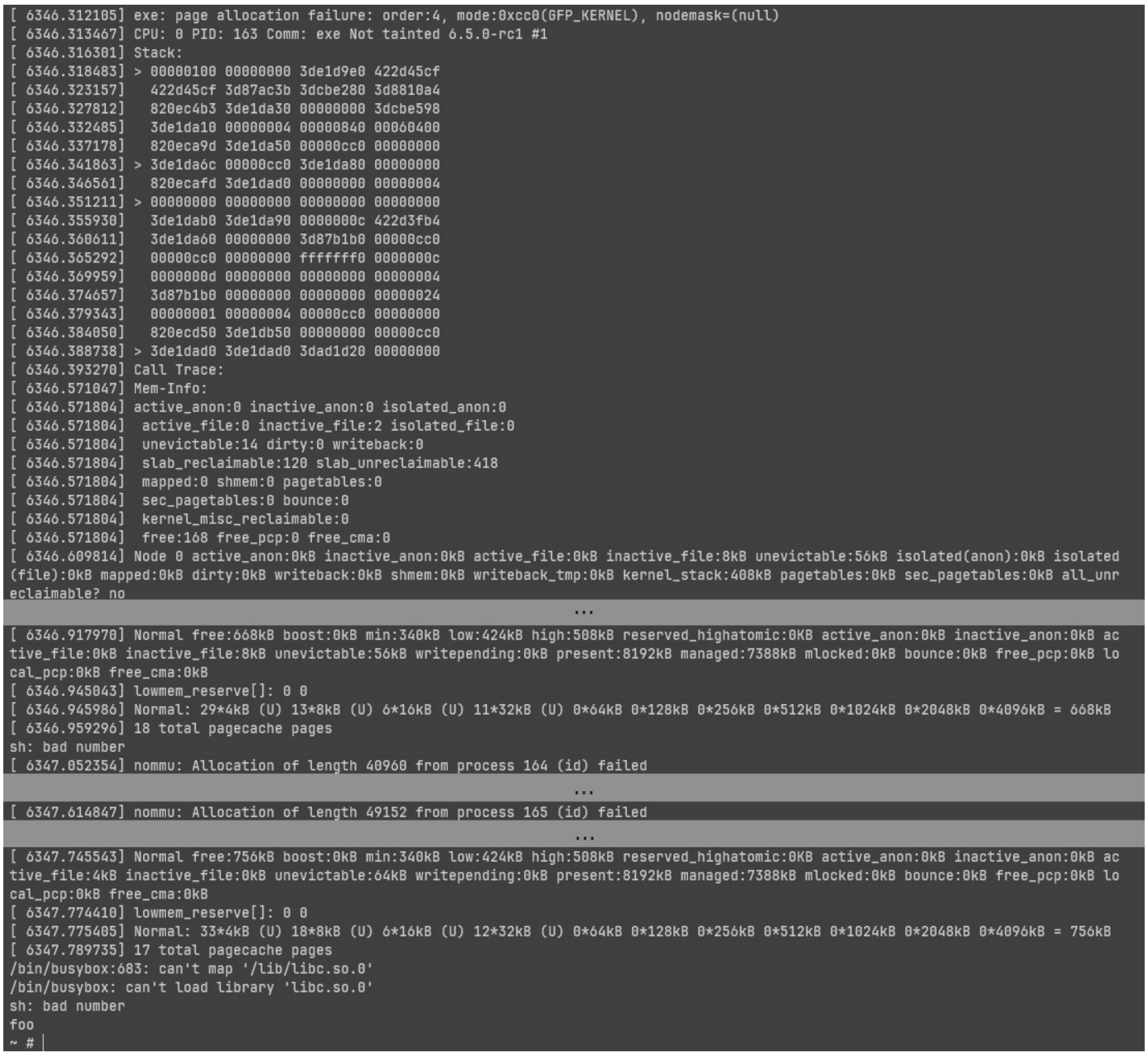
```
CPU countproc/cp: 1fo
CPU list        : 0
vendor_id       : Tensilica
model           : Xtensa LX7.0.12
core ID         : LX7_ESP32_S3_MP
build ID        : 0x90f1f
config ID       : c2f0fffe:23090f1f
byte order      : little
cpu MHz         : 160.00
bogomips        : 320.00
flags           : nmi debug ocd trax perf density bool
ean loop nsa minmax sext clamps mac16 mul16 mul32 mul3
2h fpu s32c1i
physical aregs  : 64
misc regs       : 4
ibreak          : 2
dbreak          : 2
perf counters   : 2
num ints        : 32
ext ints        : 26
int levels      : 6
timers          : 3
debug level     : 6
icache line size: 4
icache ways     : 1
icache size     : 0
icache flags    :
dcache line size: 16
dcache ways     : 1
dcache size     : 0
dcache flags    :
load/store exceptions   : 3934682
load/store clock count  : 430548765
~ # |
```

Figure 6.3: Output of `cat /proc/cpuinfo`

6.1.4 Command Substitution issues

One observation of particular interest and as such one that indicates a flaw in the established system functionality, is the inability for correct Command Substitution. The GNU bash manual [81] describes Command Substitution as the ability to „[...] allow the output of a command to replace the command itself“. In other words, by using the syntax `$(CMD)`, where `CMD` specifies the command being replaced by its own output, one can integrate a subroutine into a parent command. A simple example is `echo $(echo "foo")`. The outer echo command spawns a subshell process to evaluate its argument, since it is a command substitution; this results in executing the inner `echo "foo"` command in said subshell process, substituting its output `foo` as the argument for the outer echo, and thus printing `foo` to the stdout. An excerpt from the output generated by issuing `echo $(echo "foo")` can be seen in Figure 6.4, the full output can be found in Appendix A.

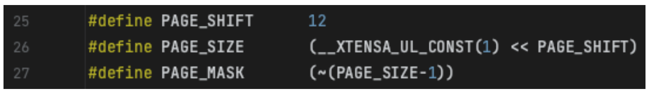
The output indicates a page allocation failure by three involved processes, in the example



```
[ 6346.312105] exe: page allocation failure: order:4, mode:0xcc0(GFP_KERNEL), nodemask=(null)
[ 6346.313467] CPU: 0 PID: 163 Comm: exe Not tainted 6.5.0-rc1 #1
[ 6346.316301] Stack:
[ 6346.318483] > 00000100 00000000 3de1d9e0 422d45cf
[ 6346.323157]   422d45cf 3d87ac3b 3dcbe280 3d8810a4
[ 6346.327812]   820ec4b3 3de1da30 00000000 3dcbe598
[ 6346.332485]   3de1da10 00000004 00000840 00060400
[ 6346.337178]   820eca9d 3de1da50 00000cc0 00000000
[ 6346.341863] > 3de1da6c 00000cc0 3de1da80 00000000
[ 6346.346561]   820ecafd 3de1dad0 00000000 00000004
[ 6346.351211] > 00000000 00000000 00000000 00000000
[ 6346.355930]   3de1dab0 3de1da90 0000000c 422d3fb4
[ 6346.360611]   3de1da60 00000000 3d87b1b0 00000cc0
[ 6346.365292]   00000cc0 00000000 fffffff0 0000000c
[ 6346.369959]   0000000d 00000000 00000000 00000004
[ 6346.374657]   3d87b1b0 00000000 00000000 00000024
[ 6346.379343]   00000001 00000004 00000cc0 00000000
[ 6346.384050]   820ecd50 3de1db50 00000000 00000cc0
[ 6346.388738] > 3de1dad0 3de1dad0 3dad1d20 00000000
[ 6346.393270] Call Trace:
[ 6346.571047] Mem-Info:
[ 6346.571804] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6346.571804]  active_file:0 inactive_file:2 isolated_file:0
[ 6346.571804]  unevictable:14 dirty:0 writeback:0
[ 6346.571804]  slab_reclaimable:120 slab_unreclaimable:418
[ 6346.571804]  mapped:0 shmem:0 pagetables:0
[ 6346.571804]  sec_pagetables:0 bounce:0
[ 6346.571804]  kernel_misc_reclaimable:0
[ 6346.571804]  free:168 free_pcp:0 free_cma:0
[ 6346.609814] Node 0 active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:8kB unevictable:56kB isolated(anon):0kB isolated
(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:408kB pagetables:0kB sec_pagetables:0kB all_unr
eclaimable? no
                                                              ...
[ 6346.917970] Normal free:668kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:0kB inactive_file:8kB unevictable:56kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6346.945043] lowmem_reserve[]: 0 0
[ 6346.945986] Normal: 29*4kB (U) 13*8kB (U) 6*16kB (U) 11*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 668kB
[ 6346.959296] 18 total pagecache pages
sh: bad number
[ 6347.052354] nommu: Allocation of length 40960 from process 164 (id) failed
                                                              ...
[ 6347.614847] nommu: Allocation of length 49152 from process 165 (id) failed
                                                              ...
[ 6347.745543] Normal free:756kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:4kB inactive_file:0kB unevictable:64kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6347.774410] lowmem_reserve[]: 0 0
[ 6347.775405] Normal: 33*4kB (U) 18*8kB (U) 6*16kB (U) 12*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 756kB
[ 6347.789735] 17 total pagecache pages
/bin/busybox:683: can't map '/lib/libc.so.0'
/bin/busybox: can't load library 'libc.so.0'
sh: bad number
foo
~ # |
```

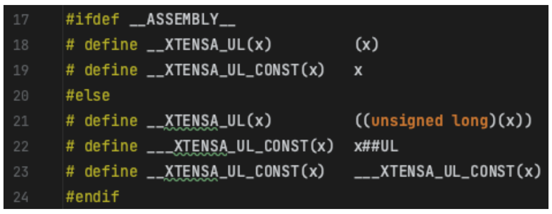
Figure 6.4: Excerpt of `echo $(echo "foo")` output

the ones with process ids (PIDs) 163, 164 and 165, each attempting to allocate memory of length 40960, 40960 and 49152 (in bytes) respectively. Additional processes being spawned by the invokation makes sense, regarding the necessity of spawning a subshell. The page allocation order of 4 on the first line translates to $2^4 = 16$ pages attempted to be allocated [82]. The `PAGE_SIZE` employed in the system is known from the kernel source; more specifically, the header files page.h and types.h located in arch/xtensa/include/uapi /asm of the kernel source define the `PAGE_SIZE` to be 4 KB, as can be seen in Figure 6.5 and 6.6. This yields an attempted allocation of $16 * 4 = 64$ KB $= 65536$ bytes, aligning with the SoCs being designed to map external RAM in blocks of 64 KB via DCache or ICache [30, p. 393] and as such justifies the aforementioned lengths.



```
#define PAGE_SHIFT      12
#define PAGE_SIZE       (__XTENSA_UL_CONST(1) << PAGE_SHIFT)
#define PAGE_MASK       (~(PAGE_SIZE-1))
```

Figure 6.5: Detail from page.h



```
#ifdef __ASSEMBLY__
# define __XTENSA_UL(x)          (x)
# define __XTENSA_UL_CONST(x)    x
#else
# define __XTENSA_UL(x)          ((unsigned long)(x))
# define ___XTENSA_UL_CONST(x)   x##UL
# define __XTENSA_UL_CONST(x)    ___XTENSA_UL_CONST(x)
#endif
```

Figure 6.6: Detail from types.h

Interestingly enough, the intended output `foo` can nevertheless be seen on the last line. While the precise cause of this obscure behaviour has not been conclusively determined, the given Mem-Info indicates sufficient memory of certain types, which might imply none of them being of the type attempted to allocate. Another hypothesis suggests memory fragmentation on the kernel side being responsible, particularly given the absence of a MMU.

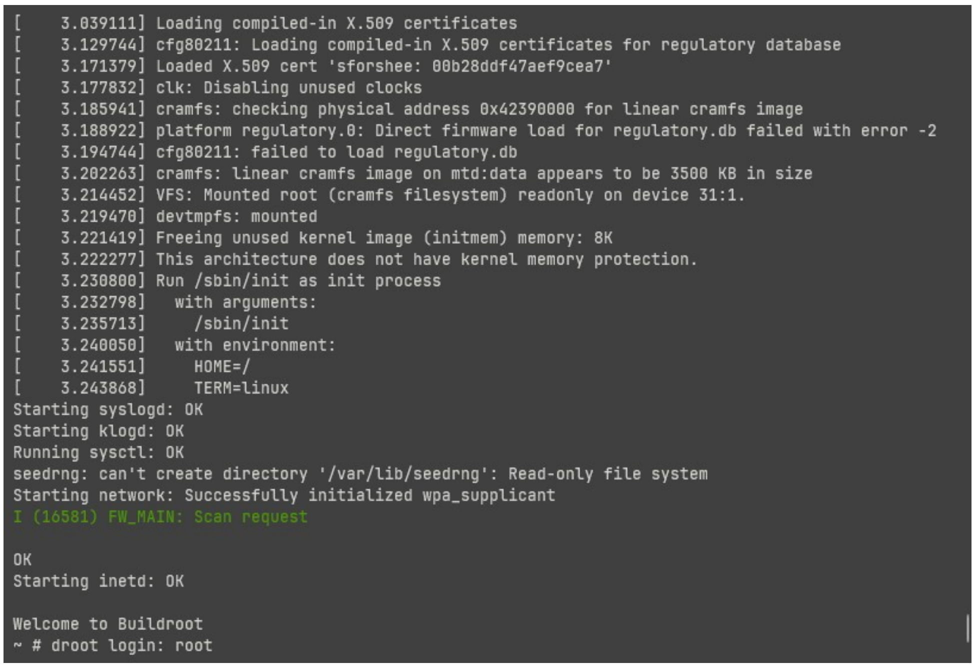
To summarize the general observations, the findings from subsections 6.1.1 through 6.1.3 were to be expected and accurately represent the characteristics of the Xtensa ISA and the ESP32-S3 SoC, along with the specific design choices made. However, the observations in subsection 6.1.4 raise unresolved questions. For practical applications of this implementation, it would be advisable to identify the exact cause of the observed issue to prevent potential critical consequences.

6.2 Time measurements

Timing metrics hold significant value in evaluating the effectiveness of the implemented system. Even if the system were to show flawless functionality, the utility of such performance becomes limited if it fails to operate within acceptable time constraints.

6.2.1 Booting

An initial critical metric to evaluate is the duration required for the system to complete its boot process. The boot time for the implemented system detailed in Chapter 5 is explicitly indicated during the boot process. The numerical value displayed on the left represents the elapsed time in seconds from the initiation of the monitoring process to the point of user space login, *i.e.*, approximately 3.2 seconds in Figure 6.7. Given its brevity, this boot time presents no significant issues and is comfortably acceptable for practical applications.



```
[    3.039111] Loading compiled-in X.509 certificates
[    3.129744] cfg80211: Loading compiled-in X.509 certificates for regulatory database
[    3.171379] Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[    3.177832] clk: Disabling unused clocks
[    3.185941] cramfs: checking physical address 0x42390000 for linear cramfs image
[    3.188922] platform regulatory.0: Direct firmware load for regulatory.db failed with error -2
[    3.194744] cfg80211: failed to load regulatory.db
[    3.202263] cramfs: linear cramfs image on mtd:data appears to be 3500 KB in size
[    3.214452] VFS: Mounted root (cramfs filesystem) readonly on device 31:1.
[    3.219470] devtmpfs: mounted
[    3.221419] Freeing unused kernel image (initmem) memory: 8K
[    3.222277] This architecture does not have kernel memory protection.
[    3.230800] Run /sbin/init as init process
[    3.232798]   with arguments:
[    3.235713]     /sbin/init
[    3.240050]   with environment:
[    3.241551]     HOME=/
[    3.243868]     TERM=linux
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
seedrng: can't create directory '/var/lib/seedrng': Read-only file system
Starting network: Successfully initialized wpa_supplicant
I (16581) FW_MAIN: Scan request

OK
Starting inetd: OK

Welcome to Buildroot
~ # droot login: root
```

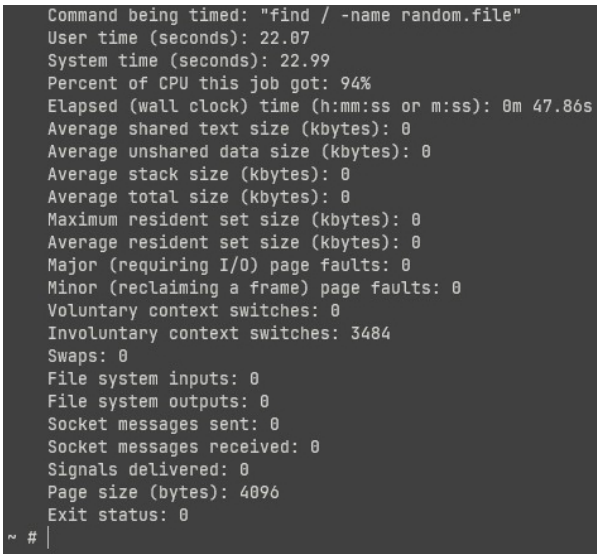
Figure 6.7: Kernel image boot time

6.2.2 Measuring execution time of utilities

Another relevant aspect to evaluate is the execution time of specific binary utilities intended for practical use. The subsequent section provides measurements obtained using the `time` utility from BusyBox. In its standard mode of operation, `time` measures three key parameters: user time, which represents the CPU time spent in user mode; system time, denoting the CPU time spent in kernel mode; and overall elapsed time, also known as wall clock time, which accounts for the total duration from the moment the measured binary is invoked to its ultimate termination. When invoked with the `-v` flag for verbose output, it provides additional metrics such as CPU usage percentages and context switches [41], [83].

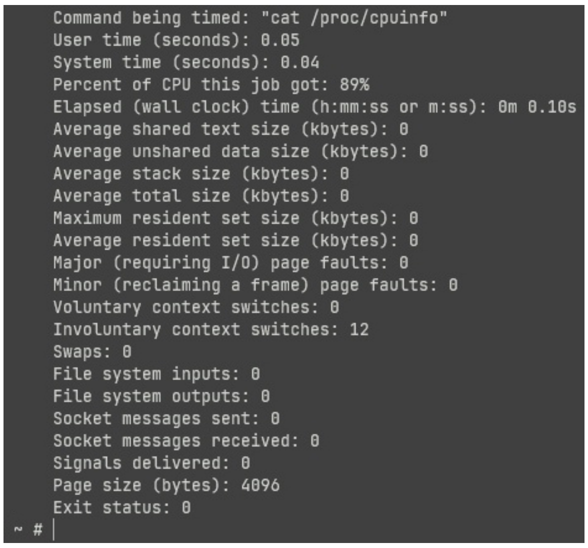
Measuring `find`

For the first example an empty sample file is created in `/tmp/random.file`. In this example it is measured how long it takes for the `find` utility to locate the file, starting from the filesystem's root directory, *i.e.*, the following command is issued: `time -v find / -name random.file`. The output reveals a user time of 22.07 seconds and a system time of 22.99 seconds, signifying a balanced workload between user and kernel modes. This finding is supported by the CPU usage of 94%, which is expected given that the operation is the only process actively demanding CPU resources. The wall clock time is registered at 47.86 seconds, aligning with the aforementioned metrics, since $(usertime + kerneltime)/wallclock \approx CPUusage$ holds true. The exit status of 0 also indicates a successful return from the method. The result can be seen in Figure 6.8.



```
    Command being timed: "find / -name random.file"
    User time (seconds): 22.07
    System time (seconds): 22.99
    Percent of CPU this job got: 94%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0m 47.86s
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 0
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 0
    Voluntary context switches: 0
    Involuntary context switches: 3484
    Swaps: 0
    File system inputs: 0
    File system outputs: 0
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
~ # |
```

Figure 6.8: Measuring `find`



```
    Command being timed: "cat /proc/cpuinfo"
    User time (seconds): 0.05
    System time (seconds): 0.04
    Percent of CPU this job got: 89%
    Elapsed (wall clock) time (h:mm:ss or m:ss): 0m 0.10s
    Average shared text size (kbytes): 0
    Average unshared data size (kbytes): 0
    Average stack size (kbytes): 0
    Average total size (kbytes): 0
    Maximum resident set size (kbytes): 0
    Average resident set size (kbytes): 0
    Major (requiring I/O) page faults: 0
    Minor (reclaiming a frame) page faults: 0
    Voluntary context switches: 0
    Involuntary context switches: 12
    Swaps: 0
    File system inputs: 0
    File system outputs: 0
    Socket messages sent: 0
    Socket messages received: 0
    Signals delivered: 0
    Page size (bytes): 4096
    Exit status: 0
~ # |
```

Figure 6.9: Measuring `cat`

Measuring `cat`

In this second example, we measure the `cat` command. This command concatenates the contents of files and prints the result to stdout, however, by specifying only a single file, only that file's content gets printed. Hence, for the measurement we will print the contents of the previously presented cpuinfo file located in the /proc pseudo filesystem by issuing `time -v cat /proc/cpuinfo`. The output records also here an almost equal amount of 0.05 seconds user time and 0.04 seconds kernel time. The CPU usage is measured at 89% much like the previous `find` example, but only for a very short duration of 0.10 seconds. Here as well does the approximate equation $(usertime + kerneltime)/wallclock \approx CPUusage$ hold true. Having an exit status of 0, also this command returns successfully. The result can be seen in Figure 6.9.

Measuring `tar` with `gzip`

[41] The `tar` utility serves to create an uncompressed archive file from one or more individual files, whilst the `gzip` utility specializes in single-file compression. These utilities are often used in conjunction to overcome `gzip`'s limitation of handling only single files, as `tar` enables the archiving of multiple files into a single uncompressed file suitable for compression. In standard Linux systems, the `tar` utility can directly archive and compress files into `gzip` format using the `-z` flag. However, this implementation employs limited versions of both utilities, derived from BusyBox, which do not natively support this functionality. To circumvent this limitation, the pipe functionality can be employed, which serves the purpose of redirecting one command's output to another command's stdin instead of its own stdout [84].

For demonstration, a 1 MB sample file filled with zeros is created in the /tmp directory using the command `dd if=/dev/zero of=/tmp/zeros.file bs=1024 count=1024`. This consumes a significant portion of the remaining writable memory, leaving approximately

1.8 MB available. Upon navigating to the /tmp directory, the archived and compressed file is generated with `tar -cf - zeros.file | gzip -9 > zeros.file.tar.gz`. In this command, the `-cf` flags instruct `tar` to *create* a *file*, but using – as the filename redirects the output to stdout, however, intercepted by the pipe operator |, hence channeling the output to `gzip`'s stdin instead. The `-9` flag specifies the highest level of compression, and the output is finally redirected via `>` to the file `zeros.file.tar.gz`. The measurement can then be executed with `time -v tar -cf - | gzip -9 > zeros.file.tar.gz`. The output can be seen in 6.10; for the sake of completeness, a subsequent – yet not measured – invocation of `du -ha .` has been issued to present the file sizes of the /tmp directory's contents. Concerning the ratio between user time and kernel time, the observations align with those made in the previous examples. The noteworthy divergence is the relative increase in wall clock time of 0.75 seconds, coupled with a reduced CPU utilization of 19%. This discrepancy may be attributed to increased read and write durations, allowing the CPU more idle time. Without further investigation of the internal specifics, may be speculated about a possibly large number of necessary instruction memory reads by either of `tar` or `gzip`; assuming this being the case would imply, as outlined in 4.2, slower reads from flash memory instead of PSRAM due to the nature of XIP. Here as well does the exit status of 0 indicate a successful return.

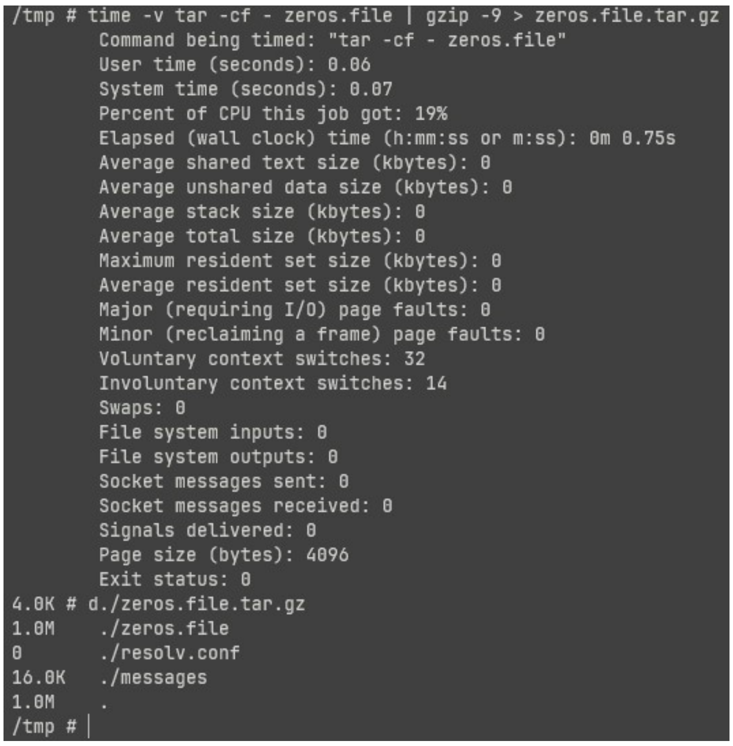

```
/tmp # time -v tar -cf - zeros.file | gzip -9 > zeros.file.tar.gz
        Command being timed: "tar -cf - zeros.file"
        User time (seconds): 0.06
        System time (seconds): 0.07
        Percent of CPU this job got: 19%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0m 0.75s
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 0
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 0
        Voluntary context switches: 32
        Involuntary context switches: 14
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
4.0K # d./zeros.file.tar.gz
1.0M    ./zeros.file
0       ./resolv.conf
16.0K   ./messages
1.0M    .
/tmp # |
```

Figure 6.10: Measuring `tar` with `gzip`

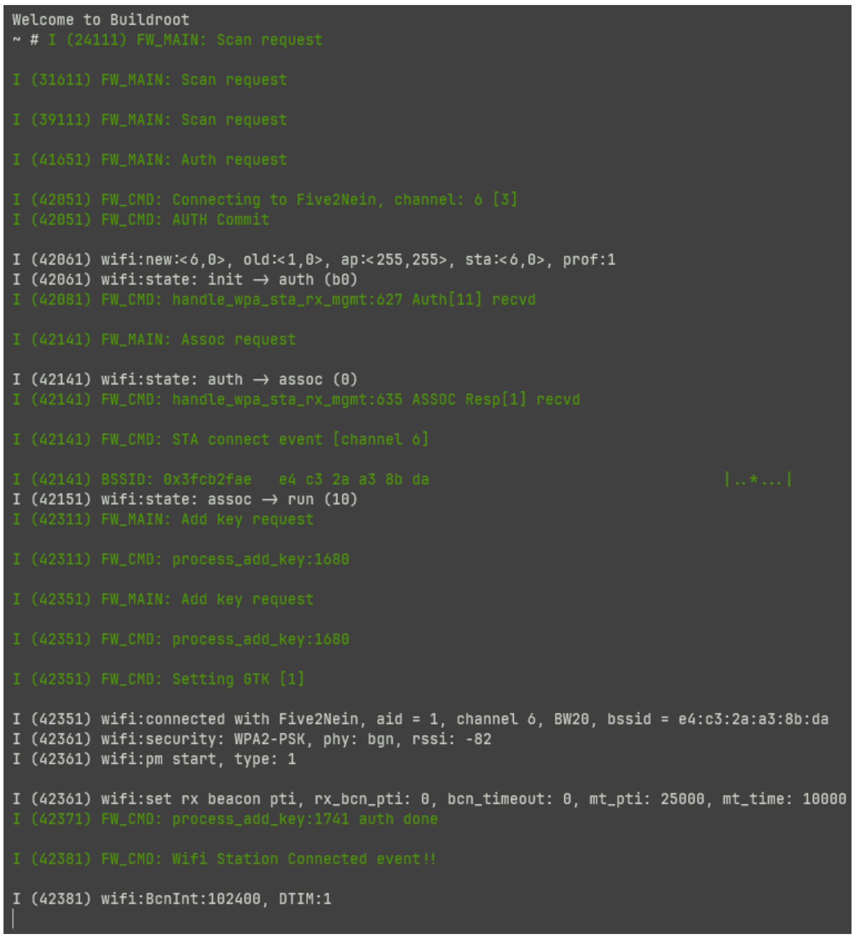
To conclude the evaluation of the timing metrics, the observations through these experiments offer valuable insights that may be applicable to a wide range of real-world scenarios. However, it is imperative to recognize the limitations of the current implementation, particularly in contexts requiring high temporal precision. For instance, employing this MCU for applications such as capturing time-sensitive data in downhill ski races, where even hundredths of a second are crucial, would likely be unsuitable. Therefore, while the measurements present a functional foundation, they highlight the necessity for specialized adjustments when high-performance or time-sensitive operations are involved.

6.3 WiFi capabilities

This section serves the purpose of assessing the WiFi capabilities inherent to the implemented system. As outlined in 5.1, the wpa_supplicant is configured to facilitate connectivity with a WPA2 network.

6.3.1 Connectivity behaviour

During the boot sequence, observations confirm that the wpa_supplicant autonomously initiates attempts to establish a network connection. While it generally succeeds in authenticating and connecting, the process occasionally necessitates multiple attempts, as indicated by the recurring `FW_MAIN: Scan request` messages displayed in Figure 6.11. Moreover, it has been observed that the network connection is subject to occasional dis-



```
Welcome to Buildroot
~ # I (24111) FW_MAIN: Scan request

I (31611) FW_MAIN: Scan request

I (39111) FW_MAIN: Scan request

I (41651) FW_MAIN: Auth request

I (42051) FW_CMD: Connecting to Five2Nein, channel: 6 [3]
I (42051) FW_CMD: AUTH Commit

I (42061) wifi:new:<6,0>, old:<1,0>, ap:<255,255>, sta:<6,0>, prof:1
I (42061) wifi:state: init -> auth (b0)
I (42081) FW_CMD: handle_wpa_sta_rx_mgmt:627 Auth[11] recvd

I (42141) FW_MAIN: Assoc request

I (42141) wifi:state: auth -> assoc (0)
I (42141) FW_CMD: handle_wpa_sta_rx_mgmt:635 ASSOC Resp[1] recvd

I (42141) FW_CMD: STA connect event [channel 6]

I (42141) BSSID: 0x3fcb2fae   e4 c3 2a a3 8b da                              |..*...|
I (42151) wifi:state: assoc -> run (10)
I (42311) FW_MAIN: Add key request

I (42311) FW_CMD: process_add_key:1680

I (42351) FW_MAIN: Add key request

I (42351) FW_CMD: process_add_key:1680

I (42351) FW_CMD: Setting GTK [1]

I (42351) wifi:connected with Five2Nein, aid = 1, channel 6, BW20, bssid = e4:c3:2a:a3:8b:da
I (42361) wifi:security: WPA2-PSK, phy: bgn, rssi: -82
I (42361) wifi:pm start, type: 1

I (42361) wifi:set rx beacon pti, rx_bcn_pti: 0, bcn_timeout: 0, mt_pti: 25000, mt_time: 10000
I (42371) FW_CMD: process_add_key:1741 auth done

I (42381) FW_CMD: Wifi Station Connected event!!

I (42381) wifi:BcnInt:102400, DTIM:1
```

Figure 6.11: Successful WiFi connection

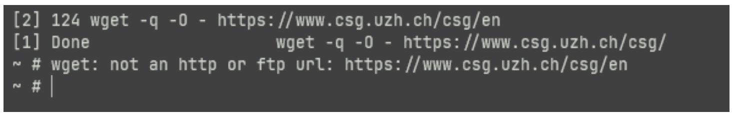
ruptions. Although, a plausible explanation for this behaviour could simply be the limited perimeter within which the integrated network adapter can effectively operate. This hypothesis is supported by the lower frequency of disconnection events when the device is positioned in closer proximity to the WiFi's access point.

6.3.2 Communicating with a RESTful API

The presented implementation features only the `wget` utility for making REST requests, more precisely, the BusyBox variant of wget [41]. Therefore, in the following, several different applications of the `wget` utility will be evaluated. In each case the expected API response is redirected to stdout by specifying `-O -`. To ensure that `wget`'s own output to stdout does not overwrite this API response, the `-q` (quiet) flag is enabled as well.

Assessment of HTTPS support

In the initial experiment, a GET request via HTTPS is made to a web page as a proof-of-concept to assess the availability of HTTPS functionality. Unfortunately, as can be seen in 6.12, `wget` yields an error, indicating support only for HTTP or FTP. Upon review-



```
[2] 124 wget -q -O - https://www.csg.uzh.ch/csg/en
[1] Done                    wget -q -O - https://www.csg.uzh.ch/csg/
~ # wget: not an http or ftp url: https://www.csg.uzh.ch/csg/en
~ # |
```

Figure 6.12: GET request via HTTPS unsuccessful

ing the suite of installed packages within the implementation, it became apparent that Certification Authority (CA) certificates were absent, a crucial component for enabling HTTPS [85]. A subsequent attempt to rebuild the rootfs with integrated CA certificates led to a size constraint issue. The newly compiled rootfs expanded from a compressed size of 3.5 MB, as mentioned in 4.2, to 6.7 MB, thereby exceeding the available storage and preventing further exploration of HTTPS support.

Extent of `wget` capabilities

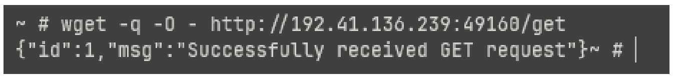
To examine the performance of the `wget` utility in the absence of HTTPS support, a rudimentary RESTful API is set up on a remote server to facilitate communication via HTTP. This API offers support for the following endpoints:

- GET method to /get
- POST method to /post, accepting data in JSON format
- PATCH method to /patch/:id, accepting data in JSON format
- DELETE method to /delete/:id

The GET request successfully executes, as shown in Figure 6.13. The same holds true for a simple POST request, whose output can be seen in Figure 6.14. The data to send can be specified with the `--post-data` option, enabling sending of JSON data. To ensure clarity, since the command issued for the POST request becomes overwritten by the API response, Listing 6.1 shows the full command.

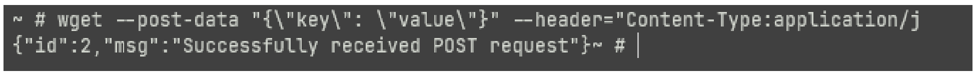
```
wget --post-data "{\"key\": \"value\"}"
   --header "Content-Type:application/json"
   -q -O - http://192.41.136.239:49160/post
```

Listing 6.1: Sending simple POST request



```
~ # wget -q -O - http://192.41.136.239:49160/get
{"id":1,"msg":"Successfully received GET request"}~ # |
```

Figure 6.13: Simple GET request



```
~ # wget --post-data "{\"key\": \"value\"}" --header="Content-Type:application/j
{"id":2,"msg":"Successfully received POST request"}~ # |
```

Figure 6.14: Simple POST request

It appears that the `--method` option – intended to specify the REST method – is not supported by the limited BusyBox variant of `wget`, thereby restricting the available REST methods to only GET and POST.

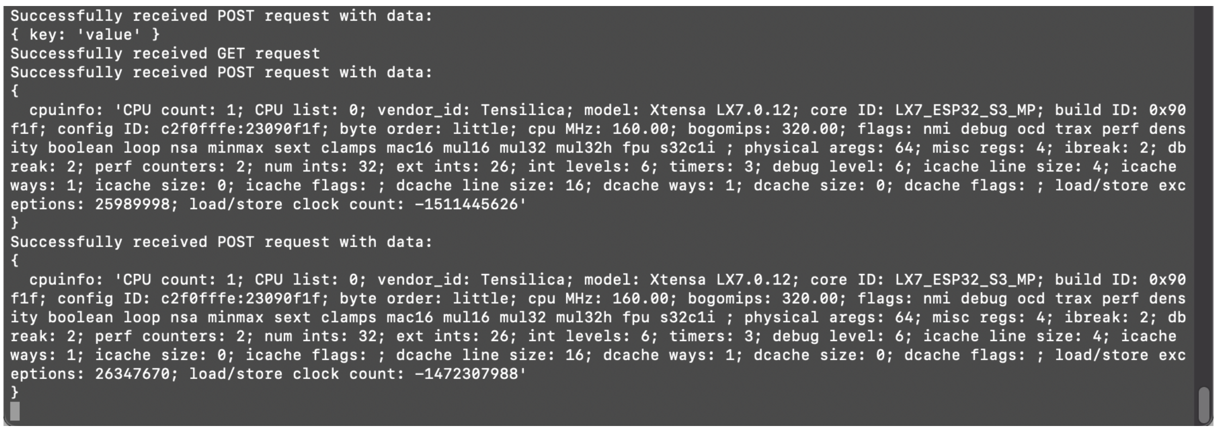
Complex POST request

In the preceding example of a POST request, the submitted data comprised a simple JSON object, specified directly on the command line. While this serves as a useful demonstration, it is not necessarily reflective of typical, real-world applications. In a more realistic setting, one might employ a shell script to dynamically populate the data for the POST request via a variable. Unfortunately, as discussed in 6.1.4, Command Substitution shows malfunctioning in the current implementation, hence, restricting this particular approach. Nonetheless, with a similar technique as the one presented in 6.2.2, where data was piped between the `tar` and `gzip` utilities, a more sophisticated POST request can be devised. Listing 6.2 shows the command issued to transmit the well-known output of `cat /proc/cpuinfo` from 6.1.3.

```
cat /proc/cpuinfo |
   sed ':x;N;$!bx;s/\n/; /g;s/\t//g' |
   awk '{printf "'\''{\"cpuinfo\":\"%s\"}'\''\n", $0}' |
   awk '{printf "wget --post-data %s --header \"Content-Type:
      application/json\" -q -O - http://192.41.136.239:49160/post\n",
       $0}' | sh
```

Listing 6.2: Sending output of `cat /proc/cpuinfo` via POST request

To break it down, the `cat /proc/cpuinfo` produces multiple lines of output, therefore using `sed ':x;N;$!bx;s/\n/; /g;s/\t//g'` combines them into a single line of output, separating them with '; '. Additionally, the `sed` command removes all tab characters. The next command – `awk '{printf "'\''{\"cpuinfo\":\"%s\"}'\''\n", $0}'` – formats the output from `sed` into JSON format. This output becomes the input to another `awk` command; `awk '{printf "wget --post-data %s --header \"Content-Type:application/json\" -q -O - http://192.41.136.239:49160/post\n", $0}'` produces a string representing the full wget command with the formatted POST data JSON inserted as an actual substring instead of being accessed as a variable. At last, the shell binary `sh` executes the constructed string as its input, essentially making the POST request. The data received on the server-side for this and also the preceding examples can be seen in Figure 6.15.



```
Successfully received POST request with data:
{ key: 'value' }
Successfully received GET request
Successfully received POST request with data:
{
  cpuinfo: 'CPU count: 1; CPU list: 0; vendor_id: Tensilica; model: Xtensa LX7.0.12; core ID: LX7_ESP32_S3_MP; build ID: 0x90
f1f; config ID: c2f0fffe:23090f1f; byte order: little; cpu MHz: 160.00; bogomips: 320.00; flags: nmi debug ocd trax perf dens
ity boolean loop nsa minmax sext clamps mac16 mul16 mul32 mul32h fpu s32c1i ; physical aregs: 64; misc regs: 4; ibreak: 2; db
reak: 2; perf counters: 2; num ints: 32; ext ints: 26; int levels: 6; timers: 3; debug level: 6; icache line size: 4; icache
ways: 1; icache size: 0; icache flags: ; dcache line size: 16; dcache ways: 1; dcache size: 0; dcache flags: ; load/store exc
eptions: 25989998; load/store clock count: -1511445626'
}
Successfully received POST request with data:
{
  cpuinfo: 'CPU count: 1; CPU list: 0; vendor_id: Tensilica; model: Xtensa LX7.0.12; core ID: LX7_ESP32_S3_MP; build ID: 0x90
f1f; config ID: c2f0fffe:23090f1f; byte order: little; cpu MHz: 160.00; bogomips: 320.00; flags: nmi debug ocd trax perf dens
ity boolean loop nsa minmax sext clamps mac16 mul16 mul32 mul32h fpu s32c1i ; physical aregs: 64; misc regs: 4; ibreak: 2; db
reak: 2; perf counters: 2; num ints: 32; ext ints: 26; int levels: 6; timers: 3; debug level: 6; icache line size: 4; icache
ways: 1; icache size: 0; icache flags: ; dcache line size: 16; dcache ways: 1; dcache size: 0; dcache flags: ; load/store exc
eptions: 26347670; load/store clock count: -1472307988'
}
```

Figure 6.15: Server logs of the API

In summary, the evaluation of the WiFi capabilities reveals functional yet limited networking features. The first noteworthy constraint is the restriction to only GET and POST requests. However, this may not pose a significant challenge depending on the application context, as other REST methods like PATCH or DELETE can be emulated by the means of GET and POST. The far more pressing issue is the absence of HTTPS support, which substantially compromises the security of communications. Moreover, this evaluation poses an additional incentive to investigate the issues discussed in 6.1.4. Although a workaround could be constructed to submit a more complex POST request, conventional methods to do so are more desirable.

In regards to the specific application scenario discussed in 1.3, concerning secure and real-time tracking of artworks, a conclusive assessment of compatibility between the WiFi capabilities of the ESP32 device running Linux and this use case remains open. Nevertheless, the findings of this work present opportunities for such future implementations in the realm of IoT tracking devices. To move from potential to practice, several key issues identified in this evaluation must first be addressed. The absence of HTTPS support stands as the most pressing concern, given its imperative role in ensuring secure communications. Beyond this, expanding the MCU's sensory capabilities stands as a further necessity to facilitate its applicability as an artwork tracking device.

Chapter 7

Conclusion & Future Work

The overarching objective of this thesis was focused on investigating the feasibility of building and operating a minimal Linux kernel, in alignment with the Xtensa ISA, on microcontrollers of the ESP32 family of MCUs, facilitating a possible integration into the IoT. Through meticulous examination of key tools and valuable contributions from external sources, a functional prototype was successfully constructed. This accomplishment enabled the culminating endeavour of conducting a comprehensive evaluation of the prototype's practical utility and limitations.

As the IoT continues to expand and shape various aspects of modern life, the role of cost-efficient and adaptable hardware becomes increasingly significant. This thesis, therefore, primarily explored the prospects of employing cost-effective microcontrollers, specifically those from the ESP32 family, to run a Linux kernel. In the concomitant study of these microcontrollers, challenges ranging from resource-constraints to ones arising from their architectural paradigms, led to distinct design choices for the kernel, ultimately providing a tailored solution for a niche architectural constitution. Such an approach not only has the potential to accelerate the growth of the IoT landscape but also hints at the possibility for some degree of standardization through the utilization of a widely known and well documented OS like Linux.

Traditionally, porting of Linux to constrained platforms, such as the one at hand, has often been a sidelining effort. By presenting the viability of Linux on cost-effective ESP32 MCUs, deviating from the norm already due to the less commonly used Xtensa ISA, not only does this thesis present an economical incentive, but even more broadens the catalogue of Linux-compatible IoT hardware. Moreover, as a consequent effect, this thesis thus also challenges prevailing notions about the hardware requisites to support the means of Linux. As a result, this thesis presents a potential enrichment to the embedded world as a whole but even more, possibly to the vast IoT jungle.

However, taking into account the current stage at which the advancements presented in this thesis find themselves, the discussed potential enrichment remains largely hypothetical. This is especially the case considering the illustrated shortcomings of the implementation, both in terms of partially flawed isolated system behavior and its yet to be optimized capabilities for secure interaction with external environments. Despite these limitations, this thesis presents opportunities, inviting for further exploration into them and the associated prospective utility gain by broadening the horizons of IoT technology. Thereby, opening the doors of innovation wider than before, even if only by a certain amount.

Subsequent research focusing on the matter of WiFi capabilities may investigate the options available for increased flash storage, thereby potentially enabling a prototype to fulfill the requirements for secure network communication. Further, creating a stable system without memory allocation deficiencies and making it a priority to ensure absence of potentially arising collateral errors and ones that might not yet have caught the eye.

Bibliography

[1] Cisco Systems, "Cisco Annual Internet Report (2018–2023)", Mar. 10, 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visited on 08/21/2023).

[2] S. Kumar, P. Tiwari, and M. Zymbler, "Internet of Things is a revolutionary approach for future technology enhancement: A review", *Journal of Big Data*, vol. 6, no. 1, p. 111, Dec. 2019. DOI: 10.1186/s40537-019-0268-2.

[3] Rachit, S. Bhatt, and P. R. Ragiri, "Security trends in Internet of Things: A survey", *SN Applied Sciences*, vol. 3, no. 1, p. 121, Jan. 2021. DOI: 10.1007/s42452-021-04156-9.

[4] U. Saeed, M. A. Khuhro, M. Waqas, and N. Mirbahar, "Comparative analysis of different Operating systems for Raspberry Pi in terms of scheduling, synchronization, and memory management", *Mehran University Research Journal of Engineering and Technology*, vol. 41, no. 3, pp. 113–119, Jul. 1, 2022. DOI: 10.22581/muet1982.2203.11.

[5] "Raspberry Pi 3 B 3 B 1 GB Prozessor: BCM2837". (), [Online]. Available: https://de.rs-online.com/web/p/raspberry-pi/1826547?gb=b (visited on 08/22/2023).

[6] Cadence Design Systems, *Xtensa Instruction Set Architecture (ISA) Summary for all Xtensa LX Processors*, 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/ip/tensilica-ip.html (visited on 08/19/2023).

[7] *I386*, in *Wikipedia*, Jul. 31, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=I386&oldid=1168007310 (visited on 08/22/2023).

[8] *Linux*, in *Wikipedia*, Jul. 28, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Linux&oldid=1167562094#cite_note-25 (visited on 08/22/2023).

[9] Espressif Systems, *ESP32-WROVER-IE Datasheet*, Feb. 9, 2023. [Online]. Available: https://www.espressif.com/en/support/documents/technical-documents (visited on 08/23/2023).

[10] "Welcome to uClibc-ng! - Embedded C library". (), [Online]. Available: https://uclibc-ng.org/ (visited on 08/22/2023).

[11] "SC0022 Raspberry Pi 3B", Mouser Electronics. (), [Online]. Available: https://www.mouser.ch/ProductDetail/358-SC0022 (visited on 08/22/2023).

[12] "Search results for: ESP32-DevKitC", Mouser Electronics. (), [Online]. Available: `https://www.mouser.ch/c/?q=ESP32-DevKitC&sort=pricing` (visited on 08/22/2023).

[13] "ESP32-WROVER-IE-N8R8 Espressif Systems", Mouser Electronics. (), [Online]. Available: `https://www.mouser.ch/ProductDetail/356-ESP32WRVIE6464UC` (visited on 08/22/2023).

[14] "Zero W & Zero WH - Raspberry Pi", Mouser Electronics. (), [Online]. Available: `https://www.mouser.ch/new/raspberry-pi/raspberry-pi-zero-w-zero-wh/` (visited on 08/22/2023).

[15] "ESP32-S3-WROOM-1-N8R8 Espressif Systems", Mouser Electronics. (), [Online]. Available: `https://www.mouser.ch/ProductDetail/356-EP32S3WROOM1N8R8` (visited on 08/22/2023).

[16] M. F. Mecklenburg, *Art in Transit: Studies in the Transport of Paintings.* Sep. 1991.

[17] J. Küffer, "Art Tracking with IoT and Blockchains", University of Zurich, 2023, in preparation.

[18] V. A. Arowoiya, A. E. Oke, C. O. Aigbavboa, and J. Aliu, "An appraisal of the adoption internet of things (IoT) elements for sustainable construction", *Journal of Engineering, Design and Technology*, vol. 18, no. 5, pp. 1193–1208, Jan. 1, 2020. DOI: `10.1108/JEDT-10-2019-0270`.

[19] D. Shah and V. haradi, "IoT Based Biometrics Implementation on Raspberry Pi", *Procedia Computer Science*, Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016, vol. 79, pp. 328–336, Jan. 1, 2016. DOI: `10.1016/j.procs.2016.03.043`.

[20] A. Milinković, S. Milinković, and L. Lazic, "Choosing the right RTOS for IoT platform", Mar. 22, 2015.

[21] J. Hu and G.-b. Zhang, "Research Transplanting Method of Embedded Linux Kernel Based on ARM Platform", *2010 International Conference of Information Science and Management Engineering*, pp. 35–38, Aug. 2010. DOI: `10.1109/ISME.2010.191`.

[22] C. Sabri, L. Kriaa, and S. L. Azzouz, "Comparison of IoT Constrained Devices Operating Systems: A Survey", in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, Oct. 2017, pp. 369–375. DOI: `10.1109/AICCSA.2017.187`.

[23] W. Wu, L. Shen, Z. Zhao, M. Li, and G. Q. Huang, "Industrial IoT and Long Short-Term Memory Network-Enabled Genetic Indoor-Tracking for Factory Logistics", *IEEE Transactions on Industrial Informatics*, vol. 18, no. 11, pp. 7537–7548, Nov. 2022. DOI: `10.1109/TII.2022.3146598`.

[24] R. I. Pereira, I. M. Dupont, P. C. Carvalho, and S. C. Jucá, "IoT embedded linux system based on Raspberry Pi applied to real-time cloud monitoring of a decentralized photovoltaic plant", *Measurement*, vol. 114, pp. 286–297, Jan. 2018. DOI: `10.1016/j.measurement.2017.09.033`.

[25] K. Khanchuea and R. Siripokarpirom, "A Multi-Protocol IoT Gateway and WiFi/BLE Sensor Nodes for Smart Home and Building Automation: Design and Implementation", in *2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*, Mar. 2019, pp. 1–6. DOI: `10.1109/ICTEmSys.2019.8695968`.

[26] M. Filippov. "Linux on esp32s3". (May 20, 2023), [Online]. Available: `https://habr.com/ru/articles/736408/` (visited on 08/26/2023).

[27] M. Babiuch, P. Foltýnek, and P. Smutný, "Using the ESP32 Microcontroller for Data Processing", in *2019 20th International Carpathian Control Conference (ICCC)*, May 2019, pp. 1–6. DOI: `10.1109/CarpathianCC.2019.8765944`.

[28] A. Sharp and Y. Vagapov, "Comparative analysis and practical implementation of the ESP32 microcontroller module for the Internet of Things", [Online]. Available: `https://core.ac.uk/outputs/287589325` (visited on 09/04/2023).

[29] R. Pawson, "The Myth of the Harvard Architecture", *IEEE Annals of the History of Computing*, vol. 44, no. 3, pp. 59–69, Jul. 1, 2022. [Online]. Available: `https://ieeexplore.ieee.org/document/9779481/` (visited on 08/31/2023).

[30] Espressif Systems, *ESP32-S3 Technical Reference Manual*, Jul. 4, 2023. [Online]. Available: `https://www.espressif.com/en/support/documents/technical-documents` (visited on 08/23/2023).

[31] F. Vasquez and C. Simmonds, *Mastering Embedded Linux Programming*, Third edition. Packt Publishing, 2021.

[32] "Cross-compilation toolchains for Linux - Home". (), [Online]. Available: `https://toolchains.bootlin.com/` (visited on 08/23/2023).

[33] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd. O'Reilly Media, Nov. 1, 2005.

[34] M. Filippov. "Linux-xtensa". (), [Online]. Available: `https://github.com/jcmvbkbc/linux-xtensa` (visited on 08/31/2023).

[35] Free Software Foundation. "GNU make". (), [Online]. Available: `https://www.gnu.org/software/make/manual/make.html` (visited on 08/29/2023).

[36] G. Kroah-Hartmann, *Linux Kernel in a Nutshell*, 1st edition. O'Reilly Media, Inc., Dec. 1, 2006.

[37] Wikipedia. "uClinux", Wikipedia. (), [Online]. Available: `https://en.wikipedia.org/wiki/%CE%9CClinux` (visited on 09/05/2023).

[38] M. Gillham. "uClinux and Linux set to merge", Linux.com. (Nov. 19, 2002), [Online]. Available: `https://www.linux.com/news/uclinux-and-linux-set-merge/` (visited on 08/28/2023).

[39] uClinux.org. "uClinux™ - Embedded Linux Microcontroller Project". (Nov. 13, 2018), [Online]. Available: `https://web.archive.org/web/20181113230737/http://www.uclinux.org/` (visited on 09/05/2023).

[40] The Buildroot developers. "The Buildroot user manual". (), [Online]. Available: `https://buildroot.org/downloads/manual/manual.html` (visited on 08/31/2023).

[41] D. Vlasenko, R. Landley, and B. Reutner-Fischer. “BusyBox Package List”. (), [Online]. Available: `https://busybox.net/downloads/BusyBox.html` (visited on 08/31/2023).

[42] BusyBox Developers. “Mirror/busybox”, GitHub. (), [Online]. Available: `https://github.com/mirror/busybox` (visited on 09/04/2023).

[43] Espressif Systems. “Espressif IoT Development Framework”, GitHub. (), [Online]. Available: `https://github.com/espressif/esp-idf` (visited on 09/05/2023).

[44] Espressif Systems. “FreeRTOS (Overview) - ESP32-S3 - ESP-IDF”. (), [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/system/freertos.html` (visited on 09/05/2023).

[45] Espressif Systems, *ESP32-S3-WROOM-1 Datasheet*, ESP32-S3-WROOM-1-N8R8 datasheet, Mar. 7, 2023. [Online]. Available: `https://www.espressif.com/en/support/documents/technical-documents` (visited on 08/23/2023).

[46] Espressif Systems. “Partitions API - ESP32 - ESP-IDF”. (), [Online]. Available: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/partition.html` (visited on 08/23/2023).

[47] GitHub user OtherCrashOverride. “Esp_partition_mmap fails with partitions larger than 3MB (IDFGH-107) · Issue #1184 · espressif/esp-idf”, GitHub. (Oct. 28, 2017), [Online]. Available: `https://github.com/espressif/esp-idf/issues/1184` (visited on 08/23/2023).

[48] Espressif Systems. “ESP Product Selector”. (), [Online]. Available: `https://products.espressif.com/#/product-selector` (visited on 08/22/2023).

[49] D. E. Comer, *Essentials of Computer Architecture*, 1st edition. Pearson Education, Aug. 23, 2004.

[50] S. Wellhöfer, *Application eXecute-In-Place (XIP) With Linux and AXFS*, Sep. 17, 2009. [Online]. Available: `https://www.scribd.com/document/19855245/Application-eXecute-In-Place-XIP-with-Linux-and-AXFS` (visited on 08/21/2023).

[51] M. Filippov. “Xtensa: Add XIP kernel support”. (Nov. 26, 2019), [Online]. Available: `https://github.com/torvalds/linux/commit/7af710d988775aadf440222ecbe0c10eecf3eb54` (visited on 08/26/2023).

[52] T. Benavides, J. Treon, J. Hulbert, and W. Chang, “The Enabling of an Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Boot Time”, *Journal of Computers*, vol. 3, no. 1, pp. 79–89, Jan. 1, 2008. [Online]. Available: `http://academypublisher.com/ojs/index.php/jcp/article/view/382` (visited on 09/05/2023).

[53] D. Harris and S. Harris, *Digital Design and Computer Architecture*, 2nd ed. Morgan Kaufmann Publishers Inc., Jul. 2012, 712 pp.

[54] J. R. Levine, *Linkers & Loaders*, 1st edition. Morgan Kaufmann, Oct. 11, 1999.

[55] The Santa Cruz Operation, *System V Application Binary Interface*, Mar. 18, 1997. [Online]. Available: `https://refspecs.linuxbase.org/` (visited on 08/27/2023).

[56] uclinux-dev. “Elf2flt/README.md”, GitHub. (), [Online]. Available: `https://github.com/uclinux-dev/elf2flt/blob/main/README.md` (visited on 09/05/2023).

[57] M. Guene. "Fdpic_doc/abi.txt", GitHub. (), [Online]. Available: https://github.com/mickael-guene/fdpic_doc/blob/master/abi.txt (visited on 09/05/2023).

[58] G. Frommelt. "fromGreg/linux-esp32s3". (Aug. 29, 2023), [Online]. Available: https://github.com/fromGreg/linux-esp32s3 (visited on 08/29/2023).

[59] M. Filippov. "Esp32s3 linux rebuild scripts". (), [Online]. Available: https://gist.github.com/jcmvbkbc/316e6da728021c8ff670a24e674a35e6 (visited on 08/30/2023).

[60] M. Filippov. "[RFC 0/5] xtensa: Support dynamic configuration". (May 22, 2017), [Online]. Available: https://gcc.gnu.org/pipermail/gcc-patches/2017-May/475109.html (visited on 08/29/2023).

[61] crosstool-NG community. "Setting up host OS". (), [Online]. Available: https://crosstool-ng.github.io/docs/os-setup/ (visited on 08/29/2023).

[62] Free Software Foundation. "GNU Autoconf". (), [Online]. Available: https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.71/autoconf.html (visited on 08/29/2023).

[63] Free Software Foundation. "GNU Automake". (), [Online]. Available: https://www.gnu.org/software/automake/manual/automake.html (visited on 08/29/2023).

[64] crosstool-NG community. "Installing crosstool-NG". (), [Online]. Available: https://crosstool-ng.github.io/docs/install/ (visited on 08/29/2023).

[65] Espressif Systems. "Xtensa-overlays". (), [Online]. Available: https://github.com/espressif/xtensa-overlays (visited on 08/30/2023).

[66] M. Filippov. "WIP: Xtensa: Change default abi to windowed abi". (Jul. 11, 2023), [Online]. Available: https://github.com/jcmvbkbc/linux-xtensa/commit/9c0802afef8797c7a343fc6b3c00abf9299898e4 (visited on 08/30/2023).

[67] Espressif Systems. "Esp-hal-components". (), [Online]. Available: https://github.com/espressif/esp-hal-components (visited on 08/30/2023).

[68] M. Filippov. "WIP: Gcc: Xtensa: Implement FDPIC support". (Apr. 29, 2023), [Online]. Available: https://github.com/jcmvbkbc/gcc-xtensa/commit/4b95a691656adf2f59786733b36cd8a352f3881a (visited on 08/30/2023).

[69] M. Filippov. "Xtensa-dynconfig". (), [Online]. Available: https://github.com/jcmvbkbc/xtensa-dynconfig/tree/original (visited on 08/30/2023).

[70] M. Filippov. "WIP: Drivers/tty/serial: Add driver for ESP32 UART". (), [Online]. Available: https://github.com/jcmvbkbc/linux-xtensa/commit/88ec52cf55d1fbcbdd78ef3bad5c963b70686883 (visited on 08/30/2023).

[71] M. Filippov. "Esp-idf/examples/get-started/linux_boot". (), [Online]. Available: https://github.com/jcmvbkbc/esp-idf/tree/linux-5.0.1/examples/get-started/linux_boot (visited on 08/30/2023).

[72] Espressif Systems. "Partitions API - ESP32-S3 - ESP-IDF". (), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/storage/partition.html?highlight=esp_partition_t# (visited on 08/23/2023).

[73] Espressif Systems. “Miscellaneous System APIs - ESP32-S3 - ESP-IDF”. (), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/api-reference/system/misc_system_api.html?highlight=esp_restart#_CPPv411esp_restartv (visited on 08/30/2023).

[74] M. Filippov. “Misc/esp32-ipc: Introduce generic IPC for ESP32”. (Aug. 30, 2023), [Online]. Available: https://github.com/jcmvbkbc/linux-xtensa/commit/e4dbf9ea0a4ecaa2b6aef8d6e5fb345d36c55643 (visited on 09/02/2023).

[75] Espressif Systems. “ESP-Hosted-NG”. (), [Online]. Available: https://github.com/espressif/esp-hosted/tree/master/esp_hosted_ng (visited on 08/30/2023).

[76] M. Filippov. “fromGreg/esp-hosted at shmem-frozen”. (), [Online]. Available: https://github.com/fromGreg/esp-hosted/tree/shmem-frozen (visited on 08/30/2023).

[77] Espressif Systems. “Getting started - ESP32-S3 - ESP-IDF”. (), [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/get-started/linux-macos-setup.html (visited on 08/30/2023).

[78] M. Kerrisk. “Proc(5) - Linux manual page”. (), [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 09/01/2023).

[79] M. Filippov. “Drivers/mtd: Add esp32 MTD support”. (Aug. 23, 2023), [Online]. Available: https://github.com/jcmvbkbc/linux-xtensa/commit/1c26c02f2daae58007ffa831735fb134b8e36b6c (visited on 09/01/2023).

[80] R. van Riel and L. Torvalds. “/proc/meminfo: Provide estimated available memory”. (Jan. 22, 2014), [Online]. Available: https://github.com/torvalds/linux/commit/34e431b0ae398fc54ea69ff85ec700722c9da773 (visited on 09/02/2023).

[81] Free Software Foundation. “Command Substitution (Bash Reference Manual)”. (), [Online]. Available: https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html (visited on 09/02/2023).

[82] M. Gorman, *Understanding the Linux® Virtual Memory Manager*, 1st Edition. Pearson Education, Apr. 2004.

[83] M. Kerrisk. “Time(1) - Linux manual page”. (), [Online]. Available: https://man7.org/linux/man-pages/man1/time.1.html (visited on 08/31/2023).

[84] Free Software Foundation. “Pipelines (Bash Reference Manual)”. (), [Online]. Available: https://www.gnu.org/software/bash/manual/html_node/Pipelines.html (visited on 09/01/2023).

[85] J. Viega, M. Messier, and P. Chandra, *Network Security with OpenSSL*, 1st Edition. O’Reilly Media, Jun. 2002.

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
ARM	Advanced RISC Machines
bFLT	Binary Flat Format
CA	Certification Authority
CISC	Complex Instruction Set Computer
CLI	Command Line Interface
CPU	Central Processing Unit
ELF	Executable and Linkable Format
FDPIC	Function Descriptor Position-Independent Code
GCC	GNU Compiler Collection
GDB	GNU Debugger
GPL	GNU General Public License
HAL	Hardware Abstraction Layer
HCL	Harvard Computing Laboratory
HDD	Hard Disk Drive
IoT	Internet-of-Things
ISA	Instruction Set Architecture
KB	Kilobyte
MB	Megabyte
MCU	Microcontroller Unit
MMU	Memory Management Unit
NFT	Non-Fungible Token
OS	Operating System
PC	Program Counter
PIC	Position-Independent Code
PID	Process Identifier
PSRAM	Pseudo-Static Random Access Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RPI	Raspberry Pi
RTOS	Run-Time Operating Systemn
SoC	System-on-a-Chip
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
VFS	Virtual File System

WiFi	Wireless Fidelity
XIP	eXecute-In-Place

Glossary

Board A Board refers to a PCB board, also called PWB board, is used to integrate or wire a MCU and peripherals onto.

Espressif Espressif refers to Espressif Systems, a semiconductor company originating from China.

Cramfs The cramfs stands for Compressed ROM filesystem. A virtual, read-only, compressed file system used in Linux, primarily for embedded systems with memory constraints.

Host (machine) In the context of embedded design, a host (machine) refers to the machine on which resource-intensive compilations take place to produce software that can be flashed onto an embedded device.

Microcontroller A microcontroller describes a processor on a single integrated circuit. A microcontroller contains one or more processor cores, memory and programmable input/output peripherals.

Rootfs The rootfs stands for Root filesystem. The virtual base filesystem in Linux mounted at / where other file systems are mounted, serving as the starting point for the filesystem hierarchy.

Stdin The stdin refers to Standard Input. A standard stream for input data, commonly used for receiving data from the user in a Linux environment.

Stdout The stdout refers to Standard Output. A standard stream where a program writes its output data, typically displayed in the terminal in a Linux system..

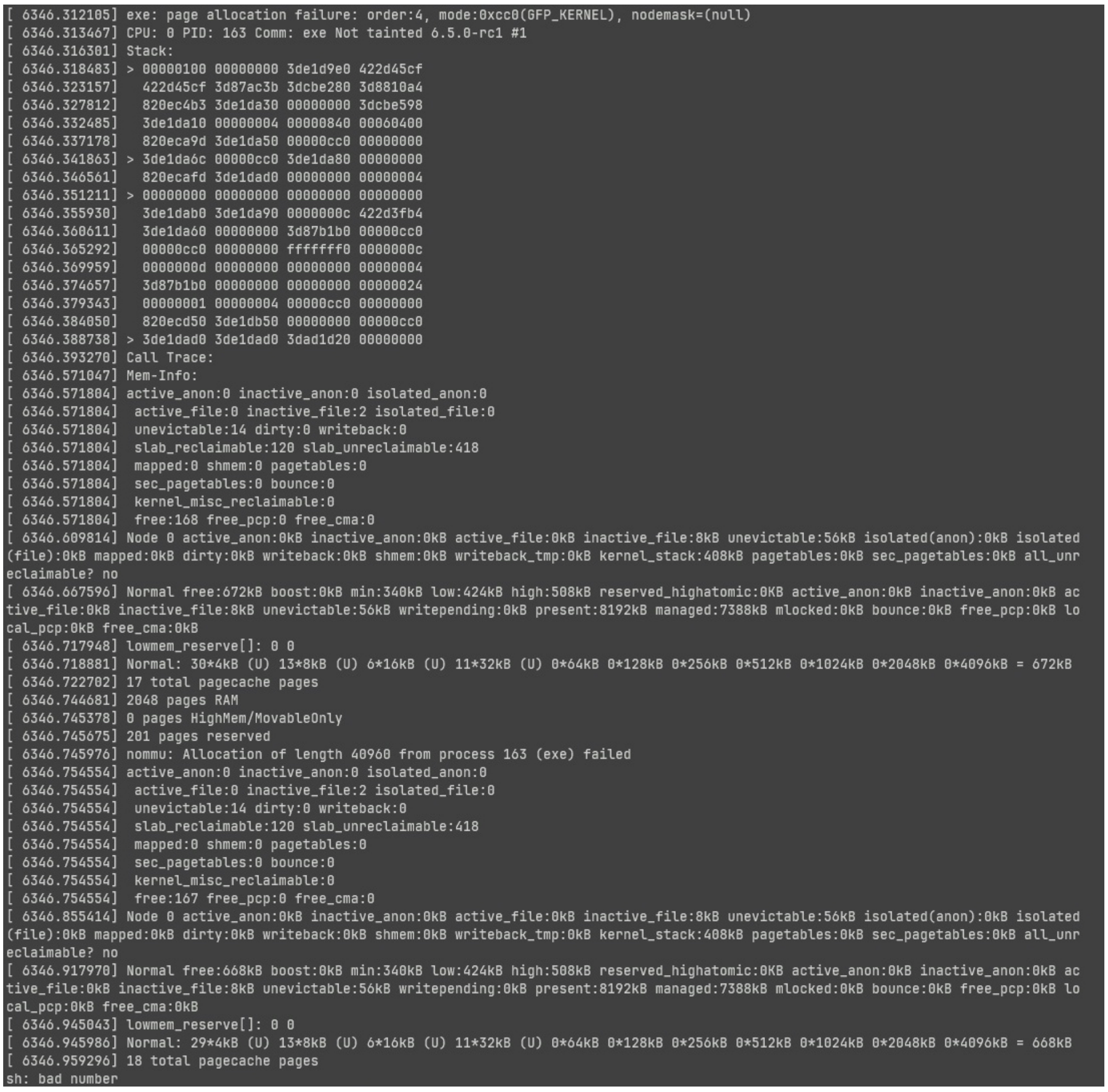
Target (device) In the context of embedded design, a target (device) refers to the embedded device, which is being targeted for development.

Tmpfs The tmpfs stands for Temporary filesystem. A virtual, RAM-based filesystem in Linux, commonly used for storing temporary files that do not require persistence across reboots.

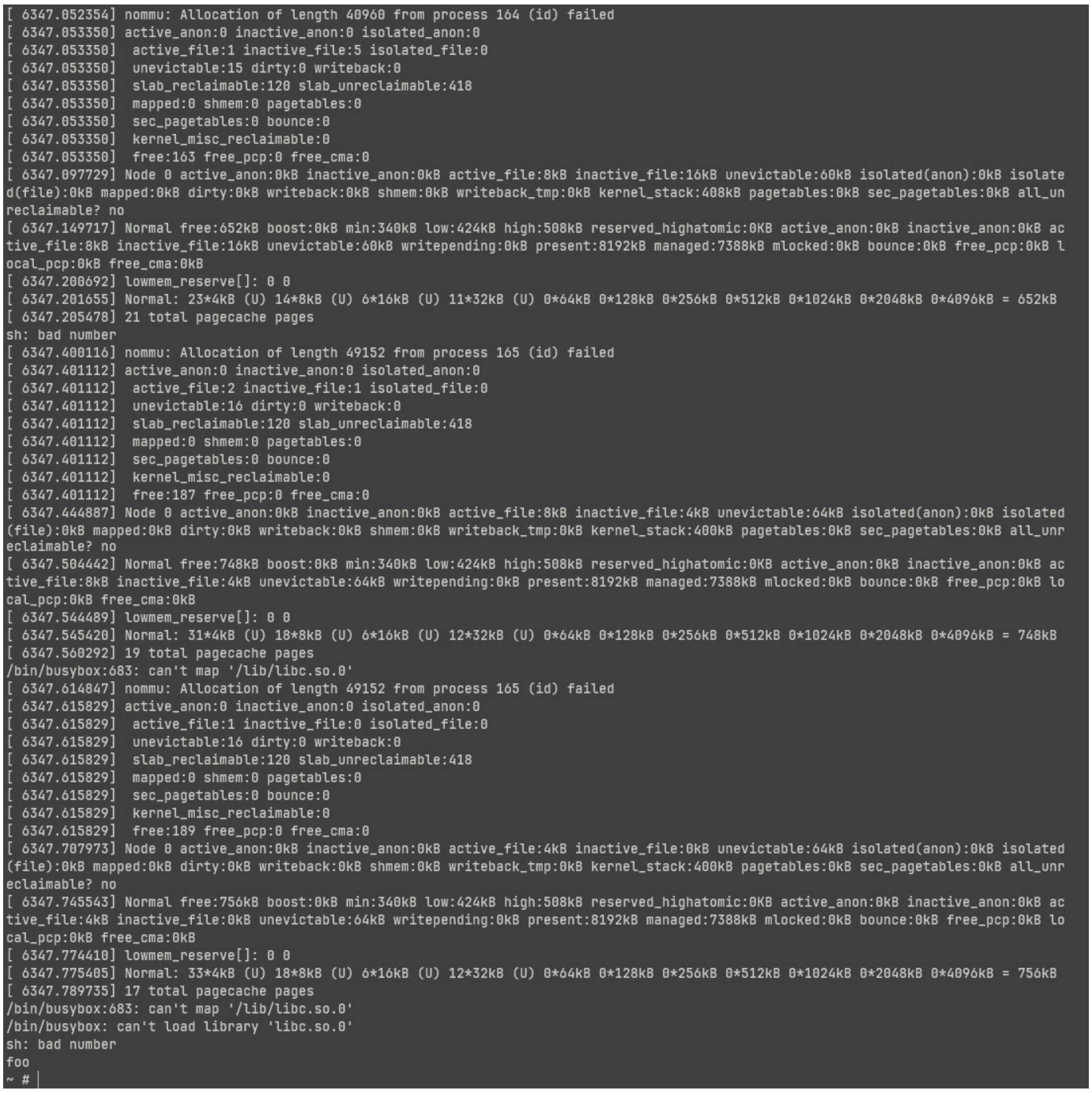
List of Figures

Appendix A

Command Substitution Output



```
[ 6346.312105] exe: page allocation failure: order:4, mode:0xcc0(GFP_KERNEL), nodemask=(null)
[ 6346.313467] CPU: 0 PID: 163 Comm: exe Not tainted 6.5.0-rc1 #1
[ 6346.316301] Stack:
[ 6346.318483] > 00000100 00000000 3de1d9e0 422d45cf
[ 6346.323157]   422d45cf 3d87ac3b 3dcbe280 3d8810a4
[ 6346.327812]   820ec4b3 3de1da30 00000000 3dcbe598
[ 6346.332485]   3de1da10 00000004 00000840 00060400
[ 6346.337178]   820eca9d 3de1da50 00000cc0 00000000
[ 6346.341863] > 3de1da6c 00000cc0 3de1da80 00000000
[ 6346.346561]   820ecafd 3de1dad0 00000000 00000004
[ 6346.351211] > 00000000 00000000 00000000 00000000
[ 6346.355930]   3de1dab0 3de1da90 0000000c 422d3fb4
[ 6346.360611]   3de1da60 00000000 3d87b1b0 00000cc0
[ 6346.365292]   00000cc0 00000000 fffffff0 0000000c
[ 6346.369959]   0000000d 00000000 00000000 00000004
[ 6346.374657]   3d87b1b0 00000000 00000000 00000024
[ 6346.379343]   00000001 00000004 00000cc0 00000000
[ 6346.384050]   820ecd50 3de1db50 00000000 00000cc0
[ 6346.388738] > 3de1dad0 3de1dad0 3dad1d20 00000000
[ 6346.393270] Call Trace:
[ 6346.571047] Mem-Info:
[ 6346.571804] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6346.571804]  active_file:0 inactive_file:2 isolated_file:0
[ 6346.571804]  unevictable:14 dirty:0 writeback:0
[ 6346.571804]  slab_reclaimable:120 slab_unreclaimable:418
[ 6346.571804]  mapped:0 shmem:0 pagetables:0
[ 6346.571804]  sec_pagetables:0 bounce:0
[ 6346.571804]  kernel_misc_reclaimable:0
[ 6346.571804]  free:168 free_pcp:0 free_cma:0
[ 6346.609814] Node 0 active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:8kB unevictable:56kB isolated(anon):0kB isolated
(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:408kB pagetables:0kB sec_pagetables:0kB all_unr
eclaimable? no
[ 6346.667596] Normal free:672kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:0kB inactive_file:8kB unevictable:56kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6346.717948] lowmem_reserve[]: 0 0
[ 6346.718881] Normal: 30*4kB (U) 13*8kB (U) 6*16kB (U) 11*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 672kB
[ 6346.722702] 17 total pagecache pages
[ 6346.744681] 2048 pages RAM
[ 6346.745378] 0 pages HighMem/MovableOnly
[ 6346.745675] 201 pages reserved
[ 6346.745976] nommu: Allocation of length 40960 from process 163 (exe) failed
[ 6346.754554] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6346.754554]  active_file:0 inactive_file:2 isolated_file:0
[ 6346.754554]  unevictable:14 dirty:0 writeback:0
[ 6346.754554]  slab_reclaimable:120 slab_unreclaimable:418
[ 6346.754554]  mapped:0 shmem:0 pagetables:0
[ 6346.754554]  sec_pagetables:0 bounce:0
[ 6346.754554]  kernel_misc_reclaimable:0
[ 6346.754554]  free:167 free_pcp:0 free_cma:0
[ 6346.855414] Node 0 active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:8kB unevictable:56kB isolated(anon):0kB isolated
(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:408kB pagetables:0kB sec_pagetables:0kB all_unr
eclaimable? no
[ 6346.917970] Normal free:668kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:0kB inactive_file:8kB unevictable:56kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6346.945043] lowmem_reserve[]: 0 0
[ 6346.945986] Normal: 29*4kB (U) 13*8kB (U) 6*16kB (U) 11*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 668kB
[ 6346.959296] 18 total pagecache pages
sh: bad number
```



```
[ 6347.052354] nommu: Allocation of length 40960 from process 164 (id) failed
[ 6347.053350] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6347.053350]  active_file:1 inactive_file:5 isolated_file:0
[ 6347.053350]  unevictable:15 dirty:0 writeback:0
[ 6347.053350]  slab_reclaimable:120 slab_unreclaimable:418
[ 6347.053350]  mapped:0 shmem:0 pagetables:0
[ 6347.053350]  sec_pagetables:0 bounce:0
[ 6347.053350]  kernel_misc_reclaimable:0
[ 6347.053350]  free:163 free_pcp:0 free_cma:0
[ 6347.097729] Node 0 active_anon:0kB inactive_anon:0kB active_file:8kB inactive_file:16kB unevictable:60kB isolated(anon):0kB isolate
d(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:408kB pagetables:0kB sec_pagetables:0kB all_un
reclaimable? no
[ 6347.149717] Normal free:652kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:8kB inactive_file:16kB unevictable:60kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB l
ocal_pcp:0kB free_cma:0kB
[ 6347.200692] lowmem_reserve[]: 0 0
[ 6347.201655] Normal: 23*4kB (U) 14*8kB (U) 6*16kB (U) 11*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 652kB
[ 6347.205478] 21 total pagecache pages
sh: bad number
[ 6347.400116] nommu: Allocation of length 49152 from process 165 (id) failed
[ 6347.401112] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6347.401112]  active_file:2 inactive_file:1 isolated_file:0
[ 6347.401112]  unevictable:16 dirty:0 writeback:0
[ 6347.401112]  slab_reclaimable:120 slab_unreclaimable:418
[ 6347.401112]  mapped:0 shmem:0 pagetables:0
[ 6347.401112]  sec_pagetables:0 bounce:0
[ 6347.401112]  kernel_misc_reclaimable:0
[ 6347.401112]  free:187 free_pcp:0 free_cma:0
[ 6347.444887] Node 0 active_anon:0kB inactive_anon:0kB active_file:8kB inactive_file:4kB unevictable:64kB isolated(anon):0kB isolated
(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:400kB pagetables:0kB sec_pagetables:0kB all_unr
eclaimable? no
[ 6347.504442] Normal free:748kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:8kB inactive_file:4kB unevictable:64kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6347.544489] lowmem_reserve[]: 0 0
[ 6347.545420] Normal: 31*4kB (U) 18*8kB (U) 6*16kB (U) 12*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 748kB
[ 6347.560292] 19 total pagecache pages
/bin/busybox:683: can't map '/lib/libc.so.0'
[ 6347.614847] nommu: Allocation of length 49152 from process 165 (id) failed
[ 6347.615829] active_anon:0 inactive_anon:0 isolated_anon:0
[ 6347.615829]  active_file:1 inactive_file:0 isolated_file:0
[ 6347.615829]  unevictable:16 dirty:0 writeback:0
[ 6347.615829]  slab_reclaimable:120 slab_unreclaimable:418
[ 6347.615829]  mapped:0 shmem:0 pagetables:0
[ 6347.615829]  sec_pagetables:0 bounce:0
[ 6347.615829]  kernel_misc_reclaimable:0
[ 6347.615829]  free:189 free_pcp:0 free_cma:0
[ 6347.707973] Node 0 active_anon:0kB inactive_anon:0kB active_file:4kB inactive_file:0kB unevictable:64kB isolated(anon):0kB isolated
(file):0kB mapped:0kB dirty:0kB writeback:0kB shmem:0kB writeback_tmp:0kB kernel_stack:400kB pagetables:0kB sec_pagetables:0kB all_unr
eclaimable? no
[ 6347.745543] Normal free:756kB boost:0kB min:340kB low:424kB high:508kB reserved_highatomic:0KB active_anon:0kB inactive_anon:0kB ac
tive_file:4kB inactive_file:0kB unevictable:64kB writepending:0kB present:8192kB managed:7388kB mlocked:0kB bounce:0kB free_pcp:0kB lo
cal_pcp:0kB free_cma:0kB
[ 6347.774410] lowmem_reserve[]: 0 0
[ 6347.775405] Normal: 33*4kB (U) 18*8kB (U) 6*16kB (U) 12*32kB (U) 0*64kB 0*128kB 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 756kB
[ 6347.789735] 17 total pagecache pages
/bin/busybox:683: can't map '/lib/libc.so.0'
/bin/busybox: can't load library 'libc.so.0'
sh: bad number
foo
~ #
```